Contents

Chapter 1

Relevant Mathematics

We will start by looking at existential and universal quantifiers; the time (and space) complexity of algorithms; BigOh and Theta notations; summations; and recurrence relations.

1.1 Existential and Universal Quantifiers

Existential and universal quantifiers provide an extremely useful language for making formal statements. You must understand them. A game between a prover and a verifier is a level of abstraction within which it is easy to understand and prove such statements.

The *Loves* Example: Suppose the relation (predicate) $Loves(b, g)$ means that the boy b loves the girl g.

Expression	**Meaning**
$\exists g\ Loves(Sam, g)$	"Sam loves some girl".
$\forall g\ Loves(Sam, g)$	"Sam loves every girl".
$\exists b \forall g\ Loves(b, g)$	"Some boy loves every girl".
$\forall b \exists g\ Loves(b, g)$	"Every boy loves some girl".
$\exists g \forall b\ Loves(b, g)$	"Some girl is loved by every boy".

Definition of Relation: A relation like $Loves(b, g)$ states for every pair of objects $b = Sam$ and $g = Mary$ that the relation either holds between them or does not. Though we will use the word *relation*, $Loves(b, g)$ is also considered to be a *predicate*. The difference is that a predicate takes only one argument and hence focuses on whether the property is $true$ or $false$ about the given tuple $\langle b, g\rangle = \langle Sam, Mary\rangle$.

Representations: Relations (predicates) can be represented in a number of ways.

Functions: A relation can be viewed as a function mapping tuples of objects either to $true$ to $false$, namely $Loves : \{b \mid b \text{ is a boy }\} \times \{g \mid g \text{ is a girl }\} \Rightarrow \{true, false\}$ or more generally $Loves : \{p_1 \mid p_1 \text{ is a person }\} \times \{p_2 \mid p_2 \text{ is a person }\} \Rightarrow \{true, false\}$.

Set of Tuples: Alternatively, it can be viewed as a set containing the tuples for which it is true, namely $Loves = \{\langle Sam, Mary\rangle, \langle Sam, Ann\rangle, \langle Bob, Ann\rangle, \ldots\}$. $\langle Sam, Mary\rangle \in Loves$ iff $Loves(Sam, Mary)$ is true.

Directed Graph Representation: If the relation only has two arguments, it can be represented by a directed graph. The nodes consist of that objects in the domain. We place a directed edge $\langle g, b\rangle$ between pairs for which the relation is true. If the domains for the first and second objects are disjoint, then the graph is bipartite. Of course, the *Loves* relation could be defined to include $Loves(Sam, Bob)$. One would then need to also consider $Loves(Sam, Sam)$. See Figure 1.1 below.

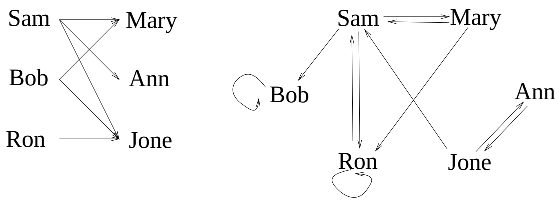


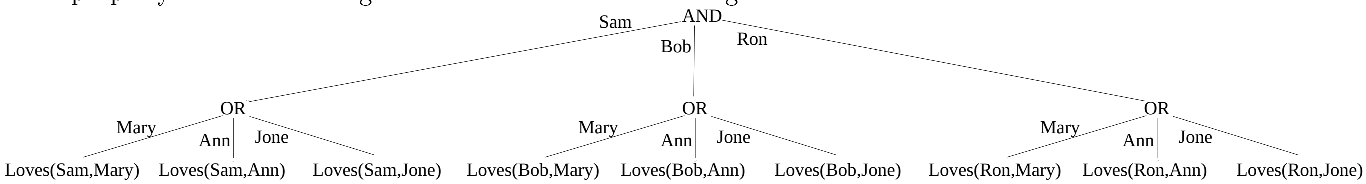
Figure 1.1: A directed graph representation of the *Loves* relation.

Quantifiers: You will be using the following quantifiers and properties.

The Existence Quantifier: The exists quantifier $\exists$ means that there is at least one object in the domain with the property. This quantifier relates the boolean operator OR. For example, $\exists g\ Loves(Sam, g) \equiv [Loves(Sam, Mary)\ OR\ Loves(Sam, Ann)\ OR\ Loves(Sam, Jone)\ OR\ \ldots]$.

The Universal Quantifier: The universal quantifier $\forall$ means that all of the object in the domain have the property. It relates the boolean operator AND. For example, $\forall g\ Loves(Sam, g) \equiv [Loves(Sam, Mary)\ AND\ Loves(Sam, Ann)\ AND\ Loves(Sam, Jone)\ AND\ \ldots]$.

Combining Quantifiers: Quantifiers can be combined. The order of operations is such that $\forall b \exists g\ Loves(b, g)$ is understood to be bracketed as $\forall b\, [\exists g\ Loves(b, g)]$, namely "Every boy has the property "he loves some girl"". It relates to the following boolean formula.



Order of Quantifiers: The order of the quantifiers matters. For example, $\forall b \exists g\ Loves(b, g)$ and $\exists g \forall b\ Loves(b, g)$ mean different things. The second one states that "The same girl is loved by every boy". To be true, there needs to be a Marilyn Monroe sort of girl that all the boys love. The first statement says that "Every boy loves some girl". A Marilyn Monroe sort of girl will make this statement true. However, it is also true in a monogamous situation in which every boy loves a different girl. Hence, the first statement can be true in more different ways than the second one. In fact, the second statement implies the first one, but not vice versa.

Definition of Free and Bound Variables: As I said, the statement $\exists g\ Loves(Sam, g)$ means "Sam loves some girl". This is a statement about Sam. Similarly, the statement $\exists g\ Loves(b, g)$ means "b loves some girl". This is a statement about the boy b. Whether the statement is true depends on which boy b is referring to. The statement is *not* about the girl g. The variable g is used as a local variable (similar to $for(i = 1; i <= 10; i++)$) to express "some girl". In this expression, we say that the variable g is *bound*, while b is *free*, because g has a quantifier and b does not.

Defining Other Relations: You can define other relations by giving an expression with free variables. For example, you can define the relations $LovesSomeOne(b) \equiv \exists g\ Loves(b, g)$.

Building Expressions: Suppose you wanted to state that "Every girl has been cheated on" using the *loves* relation. It may be helpful to break the problem into three steps.

Step 1) Assuming Other Relations: Suppose you have the relation $cheats(sam, mary)$, indicating that "Sam cheats on Mary". How would you express the fact that "Every girl has been cheated on"? The advantage of using this function is that we can focus on this one part of the statement. (Note that we are not claiming that every boy cheats. For example, one boy may have broken every girl's heart. Nor are we claiming that any girl has been cheated on in every relationship she has had.)

Given this, the answer is $\forall g \exists b\ cheats(b, g)$.

Step 2) Constructing the Other Predicate: Here we do not have a *cheats* function. Hence, we must construct a sentence from the *loves* function stating that "Sam cheats on Mary".

Clearly, there must be another girl involved besides Mary, so let's start with $\exists g'$. Now, in order for cheating to occur, who to love whom? (For simplicity's sake, let's assume that cheating means loving more than one person at the same time.) Certainly, Sam must love the other girl. He must also love Mary. If he did not love her, then he would not be cheating on her. Must Mary love Sam? No. If Sam tells Mary he loves her dearly and then a moment later he tells Sue he loves *her* dearly, then he has cheated on Mary regardless of how Mary feels about him. Therefore, Mary does not have to love Sam. In conclusion, we might define $cheat(sam, mary) \equiv \exists g' \ \big(\ Loves(sam, mary)$ and $Loves(sam, g') \ \big)$.

However, we have made a mistake here. In our example, the other girl and Mary cannot be the same person. Hence, we must define the relation as $cheat(sam, mary) \equiv \exists g' \ \big(\ Loves(sam, mary)$ and $Loves(sam, g')$ and $g' \neq mary \ \big)$.

Step 3) Combining the Parts: Combining the two relations together gives you $\forall g \exists b \exists g' \ \big(\ Loves(b, g)$ and $Loves(b, g')$ and $g' \neq g \ \big)$. This statement expresses that "Every girl has been cheated on". See Figure 1.2.

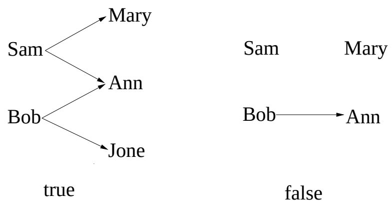


Figure 1.2: Consider the statement $\forall g \exists b \exists g'\ (Loves(b, g)$ and $Loves(b, g')$ and $g \neq g')$, "Every girl has been cheated on". On the left is an example of a situation in which the statement is true, and on the right is one in which it is false.

The Domain of a Variable: Whenever you state $\exists g$ or $\forall g$, there must be an understood set of values that the variable g might take on. This set is called the *domain* of the variable. It might be explicitly given or implied, but it must be understood. Here the domain is "the" set of girls. You must make clear whether this means all girls in the room, all the girls currently in the world, or all girls that have ever existed.

The statements $\exists g\ P(g)$ and $\forall g\ P(g)$ do not say anything about a particular girl. Instead, they say something about the domain of girls: $\exists g\ P(g)$ states that at least one girl from the domain that has the property and $\forall g\ P(g)$ states that all girls from the domain have the property. Whether the statement is true depends on the domain. For example,

$$\forall x \exists y\ x \times y = 1$$

asks whether every value has an inverse. Whether this is true depends on the domain. It is certainly not true of the domain of integers. For example, two does not have an integer inverse. It seems to be true of the domain of reals. Be careful, however; zero does not have an inverse. It would be better to write.

$$\forall x \neq 0,\ \exists y\ x \times y = 1$$

or equivalently

$$\forall x \exists y\ \ (x \times y = 1\ OR\ x = 0)$$

The Negation of a Statement: The negation of a statement is formed by putting a negation on the left-hand side. (Brackets sometimes help.) A negated statement, however, is best understood by moving the negation as deep (as far right) into the statement as possible. This is done as follows.

Negating AND and OR: A negation on the outside of an AND or an OR statement can be moved deeper into the statement using De Morgan's law. Recall that the AND is replaced by an OR and the OR is replaced with an AND.

$\neg\big(\ Loves(Sam, Mary)\ AND\ Loves(Sam, Ann)\ \big)$ iff $\neg Loves(Sam, Mary)\ OR\ \neg Loves(Sam, Ann)$: The negation of "Sam loves Mary and Ann" is "Either Sam does not love Mary or he does not love Ann". He can love one of the girls, but not both.
A common mistake is to make the negation be $\neg Loves(Sam, Mary)\ AND\ \neg Loves(Sam, Ann)$. However, this says that "Sam loves neither Mary nor Ann".

$\neg\big(\ Loves(Sam, Mary)\ OR\ Loves(Sam, Ann)\ \big)$ iff $\neg Loves(Sam, Mary)\ AND\ \neg Loves(Sam, Ann)$: The negation of "Sam either loves Mary or he loves Ann" is "Sam does not love Mary and he does not love Ann."

Negating Quantifiers: Similarly, a negation can be move past one or more quantifiers either to the right or to the left. However, you must then change these quantifiers from existential to universal and vice versa.

$\neg\big(\ \exists g\ Loves(Sam, g)\ \big)$ iff $\forall g\ \neg Loves(Sam, g)$: The negation of "There is a girl that Sam loves" is "There are no girls that Sam loves" or "Every girl is not loved by Sam." A common mistake is to state the negation as $\exists g\ \neg Loves(Sam, g)$. However, this says that "There is a girl that is not loved by Sam".

$\neg\big(\ \forall g\ Loves(Sam, g)\ \big)$ iff $\exists g\ \neg Loves(Sam, g)$: The negation of "Sam loves every girl" is "There is a girl that Sam does not love."

$\neg\big(\ \exists b \forall g\ Loves(b, g)\ \big)$ iff $\forall b\ \neg\big(\ \forall g\ Loves(b, g)\ \big)$ iff $\forall b \exists g\ \neg Loves(b, g)$: The negation of "There is a boy that loves every girl" is "There are no boys that love every girl" or "For every boy, it is not the case that he loves every girl" or "For every boy, there is a girl that he does not love".

$\neg\big(\ \exists g_1 \exists g_2\ Loves(Sam, g_1)\ AND\ Loves(Sam, g_2)\ AND\ g_1 \neq g_2\ \big)$
iff $\forall g_1 \forall g_2\ \neg\big(\ Loves(Sam, g_1)\ AND\ Loves(Sam, g_2)\ AND\ g_1 \neq g_2\ \big)$:
iff $\forall g_1 \forall g_2\ \neg Loves(Sam, g_1)\ OR\ \neg Loves(Sam, g_2)\ OR\ g_1 = g_2$
The negation of "There are two (distinct) girls that Sam loves" is "Given any pair of (distinct) girls, Sam does not love both" or "Given any pair of girls, either Sam does not love the first or he does not love the second, or you gave me the same girl twice".

The Domain Does Not Change: The negation of $\exists x \geq 5,\ x + 2 = 4$ is $\forall x \geq 5,\ x + 2 \neq 4$. The negation is NOT $\exists x < 5\ \ldots$. The reason is that both the statement and its negation are asking a question about numbers greater than 5. Is there or is there not a number with the property such that $x + 2 = 4$?

Proving a Statement True: There are a number of seemingly different techniques for proving that an existential or universal statement is true. The core of all these techniques, however, is the same. Personally, I like to view the proof as a strategy for winning a game against an adversary.

Techniques for Proving $\exists g\ Loves(Sam, g)$:

Proof by Example or by Construction: The classic technique to prove that something with a given property exists is by example. You either directly provide an example, or you describe how to construct such an object. Then you prove that your example has the property. For the above statement, the proof would state "Let g' be the girl Mary" and then would to prove that "Sam loves Mary".

Proof by Adversarial Game: Suppose you claim to an adversary that "There is a girl that Sam loves". What will the adversary say? Clearly he challenges, "Oh, yeah! Who?". You then meet the challenge by producing a specific girl g' and proving that $Loves(Sam, g')$, that is that Sam loves g'. The statement is true if you have a strategy guaranteed to beat any adversary in this game.

- If the statement is true, then you can produce such a girl g'.
- If the statement is false, then you will not be able to.

Techniques for Proving $\forall g$ $Loves(Sam, g)$:

Proof by Example Does NOT Work: Proving that Sam loves Mary is interesting, but it does not prove that he loves all girls.

Proof by Case Analysis: The laborious way of proving that Sam loves all girls is to consider each and every girl, one at a time, and prove that Sam loves her.
This method is impossible if the domain of girls is infinite.

Proof by "Arbitrary" Example: The classic technique to prove that every object from some domain has a given property is to let some symbol represent an arbitrary object from the domain and then to prove that that object has the property. Here the proof would begin "Let g' be any arbitrary girl". Because we don't actually know which girl g' is, we must either prove $Loves(Sam, g')$ (1) simply from the properties that g' has *because* she is a girl or (2) go back to doing a case analysis, considering each girl g' separately.

Proof by Adversarial Game: Suppose you claim to an adversary that "Sam loves every girl". What will the adversary say? Clearly he challenges, "Oh, yeah! What about Mary?" You meet the challenge by proving that Sam loves Mary. In other words, the adversary provides a girl g'. You win if you can prove that $Loves(Sam, g')$.
The only difference between this game and the one for existential quantifiers is who provides the example. Interestingly, the game only has one round. The adversary is only given one opportunity to challenge you.
A proof of the statement $\forall g$ $Loves(Sam, g)$ consists of a strategy for winning the game. Such a strategy takes an arbitrary girl g', provided by the adversary, and proves that "Sam loves g'". Again, because we don't actually know *which* girl g' is, we must either prove (1) that $Loves(Sam, g')$ simply from the properties that g' has because she is a girl or (2) go back to doing a case analysis, considering each girl g' separately.

- If the statement $\forall g$ $Loves(Sam, g)$ is true, then you have a strategy. No matter how the adversary plays, no matter which girl g' he gives you, Sam loves her. Hence, you can win the game by proving that $Loves(Sam, g')$.
- If the statement is false, then there is a girl g' that Sam does not love. Any true adversary (and not just a friend) will produce this girl and you will lose the game. Hence, you cannot have a winning strategy.

Proof by Contradiction: A classic technique for proving the statement $\forall g$ $Loves(Sam, g)$ is proof by contradiction. Except in the way that it is expressed, it is exactly the same as the proof by an adversary game.

> By way of contradiction (BWOC) assume that the statement is false, i.e., $\exists g$ $\neg Loves(Sam, g)$ is true. Let g' be some such girl that Sam does not love. Then you must prove that in fact Sam *does* love g'. This contradicts the statement that Sam does not love g'. Hence, the initial assumption is false and $\forall g$ $Loves(Sam, g)$ is true.

Proof by Adversarial Game for More Complex Statements: As I said, I like viewing the proof that an existential or universal statement is true as a strategy for a game between a prover and an adversary. The advantage to this technique is that it generalizes into a nice game for arbitrarily long statements.

The Steps of Game:

Left to Right: The game moves from left to right, providing an object for each quantifier.

Prover Provides $\exists b$: You, as the prover, must provide any existential objects.

Adversary Provides $\forall g$: The adversary provides any universal objects.

To Win, Prove the Relation $\boldsymbol{Loves(b', g')}$: Once all the objects have been provided, you (the prover) must prove that the innermost relation is in fact true. If you can, then you win. Otherwise, you lose.

Proof Is a Strategy: A proof of the statement consists of a strategy such that you win the game no matter how the adversary plays. For each possible move that the adversary takes, such a strategy must specify what move you will counter with.

Negations in Front: To prove a statement with a negation in the front of it, first put the statement into "standard" form with the negation moved to the right. Then prove the statement in the same way.

Examples:

$\boldsymbol{\exists b \forall g\ Loves(b, g)}$: To prove that "There is a boy that loves every girl", you must produce a specific boy b'. Then the adversary, knowing your boy b', tries to prove that $\forall g\ Loves(b', g)$ is false. He does this by providing an arbitrary girl g' that he hopes b' does not love. You must prove that "b' loves g'".

$\boldsymbol{\neg \left(\exists b \forall g\ Loves(b, g) \right) \text{ iff } \forall b \exists g\ \neg Loves(b, g)}$: With the negation moved to the right, the first quantifier is universal. Hence, the adversary first produces a boy b'. Then, knowing the adversary's boy, you produce a girl g'. Finally, you prove that $\neg Loves(b', g')$.
Your proof of the statement could be viewed as a function G that takes as input the boy b' given by the adversary and outputs the girl $g' = G(b')$ countered by you. Here, $g' = G(b')$ is an example of a girl that boy b' does not love. The proof must prove that $\forall b \neg Loves(b, G(b))$

1.2 Logarithms and Exponentials

Logarithms $\log_2(n)$ and exponentials 2^n arise often in this course and should be understood.

Uses: There are three main reasons for these to arise.

Divide Logarithmic Number of Times: Many algorithms repeatedly cut the input instance in half. A classic example is binary search. If you take some thing of size n and you cut it in half; then you cut one of these halves in half; and one of these in half; and so on. Even for a very large initial object, it does not take very long until you get a piece of size below 1. The number of times that you need to cut it is denoted by $\log_2(n)$. Here the base 2 is because you are cutting them in half. If you were to cut them into thirds, then the number of times to cut is denoted by $\log_3(n)$.

A Logarithmic Number of Digits: Logarithms are also useful because writing down a given integer value n requires $\lceil \log_{10}(n+1) \rceil$ decimal digits. For example, suppose that $n = 1{,}000{,}000 = 10^6$. You would have to divide this number by 10 six times to get to 1. Hence, by our previous definition, $\log_{10}(n) = 6$. This, however, is the number of zeros, not the number of digits. We forgot the leading digit 1. The formula $\lceil \log_{10}(n+1) \rceil = 7$ does the trick. For the value $n = 6{,}372{,}845$, the number of digits is given by $\log_{10}(6{,}372{,}846) = 6.804333$, rounded up is 7. Being in computer science, we store our values using bits. Similar arguments give that $\lceil \log_2(n+1) \rceil$ is the number of bits needed.

Exponential Search: Suppose a solution to your problem is represented by n digits. There are 10^n such strings of n digits. One way to count them is by computing the number of choices needed to choose one. There are 10 choices for the first digit. For each of those, there are 10 choices for the second. This gives $10 \times 10 = 100 choices$. Then for each of these 100 choices, there are 10 choices for the third, and so on. Another way to count them is that there are $1{,}000{,}000 = 10^6$ integers represented by 6 digits, namely $000{,}000$ up to $999{,}9999$. The difficulty in there being so many potential solutions to your problem, is that doing a blind search through them all to find your favorite one will take about 10^n time. For any reasonable n, this is a huge amount of time.

Rules: There are lots of rules about logs and exponential that one might learn. Personally, I like to minimize it to the following.

$b^n = (\overbrace{b \times b \times b \times \ldots \times b}^{n})$: This is the definition of exponentiation. b^n is n b's multiplied together.

$b^n \times b^m = b^{n+m}$: This is simply by counting the number of b's being multiplied.

$$(\overbrace{b \times b \times b \times \ldots \times b}^{n}) \times (\overbrace{b \times b \times b \times \ldots \times b}^{m}) = \overbrace{b \times b \times b \times \ldots \times b}^{n+m}.$$

$b^0 = 1$: One might guess that zero b's multiplied together is zero, but it needs to be one. One argument for this is as follows. $b^n = b^{0+n} = b^0 \times b^n$. For this to be true, b^0 must be one.

$b^{-n} = \frac{1}{b^n}$: The fact that this needs to be true can be argued in a similar way. $1 = b^{n+(-n)} = b^n \times b^{-n}$. For this to be true, b^{-n} must be $\frac{1}{b^n}$.

$(b^n)^m = b^{n \times m}$: Again we count the number of b's.

$$\overbrace{(\overbrace{b \times b \times b \times \ldots \times b}^{n}) \times (\overbrace{b \times b \times b \times \ldots \times b}^{n}) \times \ldots \times (\overbrace{b \times b \times b \times \ldots \times b}^{n})}^{m} = \overbrace{b \times b \times b \times \ldots \times b}^{n \times m}.$$

If $x = \log_b(n)$ then $n = b^x$: This is the definition of logarithms.

$\log_b(1) = 0$: This follow from $b^0 = 1$.

$\log_b(b^x) = x$ and $b^{\log_b(n)} = n$: Substituting $n = b^x$ into $x = \log_b(n)$ gives the first and substituting $x = \log_b(n)$ into $n = b^x$ gives the second.

$\log_b(n \times m) = \log_b(n) + \log_b(m)$: The number of digits to write down the product of two integers is the number to write down each of them separately (modulo rounding errors). We prove it by applying the definition of logarithms and the above rules. $b^{\log_b(n \times m)} = n \times m = b^{\log_b(n)} \times b^{\log_b(m)} = b^{\log_b(n) + \log_b(m)}$. It follows that $\log_b(n \times m) = \log_b(n) + \log_b(m)$.

$\log_b(n^d) = d \times \log_b(n)$: This is an extension of the above rule.

$\log_b(n) - \log_b(m) = \log_b(n) + \log_b(\frac{1}{m}) = \log_b(\frac{n}{m})$: This is another extension of the above rule.

$d^{c \log_2(n)} = n^{c \log_2(d)}$: This rule states that you can move things between the base to the exponent as long as you add or remove a log. The proof is as follows. $d^{c \log_2(n)} = \left(2^{\log_2(d)}\right)^{c \log_2(n)} = 2^{\log_2(d) \times c \log_2(n)} = 2^{\log_2(n) \times c \log_2(d)} = \left(2^{\log_2(n)}\right)^{c \log_2(d)} = n^{c \log_2(d)}$.

$\log_2(n) = 3.32.. \times \log_{10}(n)$: The number of bits needed to express an value integer n is 3.32.. times the number of decimal digits needed. This can be seen as follows. Suppose $x = \log_2 n$. Then $n = 2^x$, giving $\log_{10} n = \log_{10}(2^x) = x \cdot \log_{10} 2$. Finally, $x = \frac{1}{\log_{10} 2} \log_{10}(n) = 3.32.. \log_{10} n$.

Which Base: We will write $\Theta(\log(n))$ with out giving an explicit base. A high school student might use base 10 as the default, a scientist base $e = 2.718..$, and computer scientists base 2. My philosophy is that I exclude the base when it does not matter. As seen above, $\log_{10}(n)$, $\log_2(n)$, and $\log_e(n)$, differ only by a multiplicative constant. In general, we will be ignoring multiplicative constants, and hence which base used is irrelevant. I will only include the base when the base matters. For example, 2^n and 10^n differ by much more than a multiplicative constant.

The Ratio $\frac{\log a}{\log b}$: When computing the ratio between two logarithms, the base used does not matter because changing the base will introduce the same constant both on the top and the bottom, which will cancel. Hence, when computing such a ratio, you can choose which ever base makes the calculation the easiest. For example, to compute $\frac{\log 16}{\log 8}$, the obvious base to use is 2, because $\frac{\log_2 16}{\log_2 8} = \frac{4}{3}$. On the other hand, to compute $\frac{\log 9}{\log 27}$, the obvious base to use is 3, because $\frac{\log_3 9}{\log_3 27} = \frac{2}{3}$.

1.3 The Time (and Space) Complexity of an Algorithm

It is important to classify algorithms based on their time and space complexities.

Purpose:

Estimate Duration: To estimate how long a program will run.

Estimate Input Size: To estimate the largest input that can reasonably be given to the program.

Compare Algorithms: To compare the efficiency of different algorithms for solving the same problem.

Parts of Code: To help you focus your attention on the parts of code that are executed the largest number of times. This is the code you need to improve to reduce running time.

Choose Algorithm: To choose an algorithm for an application.

- If the input won't be larger than six, don't waste your time writing an extremely efficient algorithm.
- If the input size is a thousand, then be sure the program runs in polynomial not exponential time.
- If you are working on the Gnome project and the input size is a billion, then be sure the program runs in linear time.

Time and Space Complexities are Functions, $T(n)$ and $S(n)$: The time complexity of an algorithm is not a single number but is a function indicating how the running time depends on the *size* of the input. We often denote this by $T(n)$, giving the number of "operations" executed on the worst case input instance of "size" n. An example would be $T(n) = 3n^2 + 7n + 23$. Similarly, $S(n)$ gives the "size" of the rewritable memory the algorithm requires.

Ignoring Details, $\Theta(T(n))$ and $\mathcal{O}(T(n))$: Generally, we ignore the low order terms in the function $T(n)$ and the multiplicative constant in front. We also ignore the function for small values of n and focus the *asymptotic* behavior as n becomes very large. Some of the reasons are the following.

Model Dependent: The multiplicative constant in front of the time depends on how fast the computer is and on the precise definition of "size" and "operation".

Too Much Work: Counting every operation that the algorithm executes in precise detail is more work than it is worth.

Not Significant: It is much more significant whether the time complexity is $T(n) = n^2$ or $T(n) = n^3$ than whether it is $T(n) = n^2$ or $T(n) = 3n^2$.

Large n Matter: One might say that we only consider large input instance in our analysis, because the running time of an algorithm only become an issue when the input is large. However, the running time of some algorithms on small input instances is quite critical. In fact, the size n of a realistic input instance depends on both on the problem and on the application. The choice was made to consider only large n in order to provide a clean and consistent mathematical definition.

See Theta and Big Oh notations in Section 1.4.

1.3.1 Different Models of Time and Space Complexity

A model of time complexity is defined by what is considered to be the "size" of an input instance and what is considered to be a single algorithmic "operation". Different models are used for different applications depending on what we want to learn about the algorithm and at what level of detail.

Typically, the definitions of "size" and an "operation" are tied together. A basic unit is defined and then the size of an input instance is defined to be the number of these units needed to express the input and a single operation is able to do basic operations on one or two of these units. The following are the classic models used.

Ignoring Multiplicative Constants: The following models are equivalent up to some multiplicative constant.

Intuitive: Seconds. The obvious unit to measure *time* complexity is in seconds. In fact, a very intuitive definition of time complexity $T(n)$, is that an adversary is allowed to send you a hard input instance along a narrow channel and $T(n)$ is the number of seconds that it takes you to solve the problem as a function of the number of seconds it takes the adversary to send the instance. The problem with this definition is that it is very dependent on the speed of communication channel and of the computer used. However, this difference is only up to some multiplicative constant.

An algorithm is thought to be slow if an adversary can quickly give you an instance to the problem that takes the algorithm a huge amount of time to solve, perhaps even the age of the universe. In contrast, the algorithm is considered fast, if in order to force it to work hard, the adversary must also work hard communicating the instance.

Intuitive: The Size of Your Paper. Similarly, the size of an input instance can be defined to be the number of square inches of paper used by the adversary to write down the instance given a fixed symbol size. Correspondingly, as single operation would be to read, write, or manipulate one or two of these symbols. Similarly, the space requirements of the algorithm is the number of square inches of paper, with eraser, used by the algorithm to solve the given input instance.

Formal: Number of Bits. In theoretical computer science, the formal definition of the size of an instance is the number of binary bits required to encode it. For example, if the input is 101001_2, then $n = 6$. The only allowable operations are AND, OR and NOT of pairs of bits. The algorithm, in this context, can be viewed as some form of a *Turing Machine* or a circuit of AND, OR and NOT gates. Most people get nervous around this definition. Thats ok. We have others.

Practical: Number of Digits, Characters, or Words. It is easier to think of integers written in decimal notation and to think of other things as written in sentences composed of the letters A-Z. A useful definition of the size of an instance is the number of digits or characters required to encode it. You may argue that an integer can be stored as a single variable, but for a really big value, your algorithm will need a data structure consisting of a number of 32-bit words to store it. In this case, the size of an instance would be the number of these words needed to represent it.

For each of these definitions of size, the definition of an operation would be a single action on a pair of digits, characters, or 32-bit words.

Within a Constant: These definitions of size are equivalent within a multiplicative constant. Given a value N, the number of bits to represent it will be $\lceil \log_2(N+1) \rceil$, the number of digits will be $\lceil \log_{10}(N+1) \rceil = \lceil \frac{1}{3.32} \log_2(N+1) \rceil$, and the number of 32-bit words will be $\lceil \log_{(2^{32})}(N+1) \rceil = \lceil \frac{1}{32} \log_2(N+1) \rceil$. See the rules about logarithms in Section 1.2.

Ignoring Logarithmic Multiplicative Factors: The following model can be practically easier to handle then the bit model and it gives the same measure of time complexity except for being a few logarithmic factors more pessimistic. This is not usually a problem, because the only time when this text cares about these factors is when differentiating between $T(n) = n$ time and $T(n) = n \log_2 n$ time, particularly when regarding algorithms for sorting.

Practical: Number of Bounded Elements. Suppose the input consists of an array, tree, graph, or strange data structure consisting of n integers (or more complex objects), each in the range $[1..n^5]$. A reasonable definition of size would be n, the number of integers. A reasonable definition of an operation would something like adding or multiplying two such integers or indexing into array using one integer in order to retrieve another.

Within a Log: Surprisingly perhaps, the time complexity of an algorithm is more pessimistic under this model than under the bit model. Suppose, for example, that under this model the algorithm requires $T(n) = n^3$ integer operations. Each integer in the range $[1..n^5]$ requires $\log_2 n^5 = 5 \log_2 n$ bits. Hence, the size of the entire input instance is $n_{bit} = n \times 5 \log_2 n$ bits. Adding two n_{bit} bit integers requires some constant times n_{bit} bit operations. To be concrete, let us say $3n_{bit}$ bit operations. This gives that the total number of bit operations required by the algorithm is $T_{bit} = (15 \log_2 n) \cdot T(n) = 15n^3 \log_2 n$. Rearranging $n_{bit} = n \times 5 \log_2 n$ gives $n = \frac{n_{bit}}{5 \log_2 n}$ and plugging this into $T_{bit} = 15n^3 \log_2 n$ gives $T_{bit}(n_{bit}) = 15 \left(\frac{n_{bit}}{5 \log_2 n}\right)^3 \log_2 n = \frac{0.12}{(\log_2 n)^2}(n_{bit})^3$ bit operations. This gives the time complexity of the algorithm in the bit model. Note that this is two logarithm factors, $\frac{0.12}{(\log_2 n)^2}$, better than time complexity $T(n) = n^3$ within the element model.

An Even Stronger Complexity Measure:

Practical: Number of Unbounded Elements. Again suppose that the input consists of an array, tree, graph, or strange data structure consisting of n integers (or more complex objects), but now there is no bound on how large each can be. Again we will define the size of an input instance to be the number of integers and a single operation to be some operation on these integers, no matter how big these integers are. Being an even more pessimistic measure of time complexity, having an algorithm that is faster according to this measure is a much stronger statement. The reason is that the algorithm must be able to solve the problem in the same $T(n)$ integer operations independent of how big the integers are that the adversary gives it.

Too Strong A Complexity Measure:

Incorrect: The Value of an Integer. When the input is an integer, it is tempting to denote its value with n and use this as the size of the input. DO NOT DO THIS! The value or magnitude of an integer is very very different than the number of bits needed to write it down. For example, $N = 10000000000000000000000000000000000000000000000000$ is a very big number. It is bigger then the number of atoms in the universe. However, we can write it on 4 inches of paper, using 50 digits. In binary, it would require 166 bits.

Any of the above models of time complexity are reasonable except for the last one.

Which Input of Which Size:

Input Size n Is Assumed to Be Very Large: Suppose one program takes $100n$ time and another takes n^2 time. Which is faster depends on the input size n. The first is slower for $n < 100$, and the second for larger inputs. It is true that some applications deal with smaller inputs, but in order to have a consistent way to compare running times, we assume that the input is very large. This is done (1) by considering only inputs whose size n is greater than some constant n_0 or (2) by taking the limit as n goes to infinity.

Which Input Do We Use To Determine Time Complexity?: $T(n)$ is the time required to execute the given algorithm on an input of size n. However, which input instance of this size do you give the program to measure its complexity? After all, there are 2^n inputs instances with n bits. Here are three possibilities:

A Typical Input: The problem with this is that it is not clear who decides what a "typical" input is. Different applications will have very different typical inputs. It is valid to design your algorithm to work well for a particular application, but if you do, this needs to be clearly stated.

Average Case or Expected: Average case analysis considers time averaged over all input instances of size n. This is equivalent to the expected time for a "random" input instance of size n. The problem with this definition is that it assumes that all instances are equally likely to occur. One might just as well consider a different probability distribution on the inputs.

Worst Case: The usual measure is to consider the instance of size n on which the given algorithm is the slowest. Namely, $T(n) = \max_{I \in \{I \mid |I|=n\}} Time(I)$. This measure provides a nice clean mathematical definition and is the most easy to analyze. The only problem is that sometimes the algorithm does much better than the worst case, because the worst case is not a reasonable input. One such example, we will see is Quick Sort.

1.3.2 Examples

Example 1: Suppose program P_1 requires $T_1(n) = n^4$ operations and P_2 requires $T_2(n) = 2^n$. Suppose that your machine executes 10^6 operations per second.

1. If $n = 1,000$, what is the running time of these programs?
   - Answer:
     (a) $T_1(n) = \left(10^3\right)^4 = 10^{12}$ operations, $= 10^6$ seconds, $= 11.6$ days.
     (b) $T_2(n) = 2^{\left(10^3\right)}$ operations. The number of years is $\frac{2^{\left(10^3\right)}}{10^6 \cdot 60 \cdot 60 \cdot 356}$. This is too big for my calculator. The log of this number is $\frac{10^3}{\log_2(10)} - \log_{10} 10^6 - \log_{10}(60 \cdot 60 \cdot 356) = 301.03 - 6 - 6.12 = 288.91$. Therefore, the number of years is $10^{288.91}$. Don't wait for it.
2. If you want to run your program in 24 hrs, how big can your input be?
   - Answer: The number of operations is $T = 24 * 60 * 60 * 10^6 = 8.64 \cdot 10^{10}$. $n_1 = T^{1/4} = 542$. $n_2 = \log_2 T = \frac{\log T}{\log 2} = 36$.
3. Approximately, for which input size, do the program have the same running times?
   - Answer: Setting $n = 16$ gives $n^4 = (16)^4 = \left(2^4\right)^4 = 2^{4 \cdot 4} = 2^{16} = 2^n$.

Example 2: Two simple algorithms, summation and factoring.

The Problems/Algorithms:

Summation: The task is to sum the N entries of an array, i.e., $A(1) + A(2) + A(3) + \ldots + A(N)$.

Factoring: The task is to find divisors of an integer N. For example, on input $N = 5917$ we output that $N = 97 \times 61$. (This problem is central to cryptography.) The algorithm checks whether N is divisible by 2, by 3, by 4, ... by N.

Time: Both algorithms require $T = N$ operations (additions or divisions).

How Hard? The Summing algorithm is considered to be very fast, while the factoring algorithm is considered to be very time consuming. However, both algorithms take $T = N$ time to complete. The time complexity of these algorithms will explain why.

Typical Values of N: In practice, the N for Factoring is much larger than that for Summation. Even if you sum all the entries in the entire 8G hard drive, then N is still only $N \approx 10^{10}$. On the other hand, the military wants to factor integers $N \approx 10^{100}$.

However, assessing the complexity of an algorithm should not be based on how it happens to be used in practice.

Size of the Input: The input for Summation is $n \approx 32N$ bits.

The input for Factoring is $n = \log_2 N$ bits. Therefore, with a few hundred bits you can write down a difficult factoring instance that is seemingly impossible to solve.

Time Complexity: The running time of Summation is $T(n) = N = \frac{1}{32}n$, which is linear in its input size.

The running time of Factoring is $T(N) = N = 2^n$, which is exponential in its input size.

This is why the Summation algorithm is considered to be *feasible*, while the Factoring algorithm is considered to be *infeasible*.

1.4 Asymptotic Notations and Their Properties

BigOh and Theta are a notation for classifying functions.

Purpose:

Time Complexity: Generally for this course, the functions that we will be classifying with be the time complexities of programs. On the other hand, these ideas can also be used to classify any function.

Functions Growth: To give an idea of how fast a function grows without going into too much detail.

Example: While on a walk in the woods, someone asks me, "What is that?"

I could answer:

"It is an adult male red-bellied sapsucker of the American northeast variety."

or I could say:

"It's a bird."

Which is better? It depends on how much detail is needed.

Classes of Functions: To classify functions into different classes.

Classifying: We classify the set of all vertebrate animals into mammal, bird, reptile, and so on; we further classify mammals into humans, cats, dogs, and so on; finally, among others, the class of humans will contain me. Similarly, we will classify functions into constants $\Theta(1)$, poly-logarithms $\log^{\Theta(1)}(n)$, polynomial $n^{\Theta(1)}$, exponentials $2^{\Theta(n)}$, double exponentials $2^{2^{\Theta(n)}}$, and so on; we further classify the polynomials $n^{\Theta(1)}$ into linear functions $\Theta(n)$, the time for sorting $\Theta(n\log n)$, quadratics $\Theta(n^2)$, cubics $\Theta(n^3)$, and so on; finally, among others, the class of linear functions $\Theta(n)$ will contain $f(n) = 2n + 3\log^2 n + 8$.

Asymptotic Growth Rate: When classifying animals, Darwin decided to consider not whether or not it sleeps during the day, but whether it has hair. When classifying functions, complexity theorists like my father decided not to consider its behavior on small values of n or even whether it is monotone increasing, but how quickly it grows when its input n grows really big. This is referred to as the *asymptotics* of the function. Here are some examples of different growth rates.

Function	Approximate value of $T(n)$ for $n =$				
$T(n)$	10	100	1,000	10,000	
$\log n$	3	6	9	13	amoeba
$\sqrt{n}$	3	10	31	100	bird
n	10	100	1,000	10,000	human
$n\log n$	30	600	9,000	130,000	giant
n^2	100	10,000	10^6	10^8	elephant
n^3	1,000	10^6	10^9	10^{12}	dinosaur
2^n	1,024	10^{30}	10^{300}	10^{3000}	the universe

Note: The universe contains approximately 10^{80} particles.

1.4.1 The Classic Classifications of Functions

The following are the main classifications of functions that will arise in this course as the running times of an algorithm or as the amount of space that it uses.

Functions with A Basic Form: Most functions considered will have the basic form $f(n) = c \cdot b^{an} \cdot n^d \cdot \log^e n$, where a, b, c, d, and e are real constants. Such functions are classified as follows.

Constants:	$\Theta(1)$	$f(n) = c$,	where $c > 0$.
Logarithms:	$\Theta(\log(n))$	$f(n) = c \cdot \log n$,	where $c > 0$.
Poly-Logarithms:	$\log^{\Theta(1)}(n)$	$f(n) = c \cdot \log^e n$,	where $c > 0$, and $e > 0$.
Linear Functions:	$\Theta(n)$	$f(n) = c \cdot n$,	where $c > 0$.
Time for Sorting:	$\Theta(n\log n)$	$f(n) = c \cdot n\log n$,	where $c > 0$.
Quadratics:	$\Theta(n^2)$	$f(n) = c \cdot n^2$,	where $c > 0$.
Cubics:	$\Theta(n^3)$	$f(n) = c \cdot n^3$,	where $c > 0$.
Polynomial:	$n^{\Theta(1)}$	$f(n) = c \cdot n^d \cdot \log^e n$,	where $c > 0$, $d > 0$, $e \in (-\infty, \infty)$.
Exponentials:	$2^{\Theta(n)}$	$f(n) = c \cdot b^{an} \cdot n^d \cdot \log^e n$,	where $c > 0$, $a > 0$, $b > 1$, $d, e \in (-\infty, \infty)$.
Double Exp.:	$2^{2^{\Theta(n)}}$		

These classes will now get discussed in more detail.

Constants $\Theta(1)$: A constant function is one whose output does not depend on its input. For example, $f(n) = 7$. One algorithm for which this function arises is popping an element off a stack that is stored as a linked list. This takes maybe 7 "operations" independent of the number n of elements in the stack. Someone else's implementation may require 8 "operations". $\Theta(1)$ is a class of functions that include all such functions. We say $\Theta(1)$ instead of 7 when we do not care what the actually constant is. Determining whether it is 7, 9, or 8.829 may be more work than it is really worth. A function like $f(n) = 8 + \sin(n)$, on the other hand, continually changes between 7 and 9 and $f(n) = 8 + \frac{1}{n}$ continually changes approaching 8. However, if we are not caring whether it is 7, 9, or 8.829, why should we care if it is changing between them? Hence, both of these functions are included in $\Theta(1)$. On the other hand, in most applications being negative $f(n) = -1$ or zero $f(n) = 0$ would be quite a different matter. Hence, these are excluded. Similarly, the function $f(n) = \frac{1}{n}$ is not included, because the only constant that it is bounded below by is zero and the zero function is not included.

Examples Included:

- $f(n) = 7$ and $f(n) = 8.829$
- $f(n) = 8 + \sin(n)$
- $f(n) = 8 + \frac{1}{n}$
- $f(n) = \{$ 1 if n is odd, 2 if n is even $\}$

Examples Not Included:

- $f(n) = -1$ and $f(n) = 0$ (fails $c > 0$)
- $f(n) = \sin(n)$ (fails $c > 0$)
- $f(n) = \frac{1}{n}$ (too small)
- $f(n) = \log_2 n$ (too big)

Logarithms $\Theta(\log(n))$: See Section 1.4 for how logarithmic functions like $\log_2(n)$ arise in this course and for some of their rules. The class of functions $\Theta(\log(n))$ consists of those functions $f(n)$ that are "some constant" times this initial function $\log_2(n)$. For example, $f(n) = 2\log_2(n)$ is included.

Slowly Growing: The functions $f(n) = \log_2(n)$ and $f(n) = \log_{10}(n)$ grow as n gets large, but very slowly. For example, even if n is 10^{80}, which is roughly the number of particles in the universe, still $f(n) = \log_{10}(n)$ is only 80. Because it grows with n, this function certainly is not included in $\Theta(1)$. However, it is much much smaller than the linear function $f(n) = n$.

Which Base: We write $\Theta(\log(n))$ with out giving an explicit base. As shown in the list of rules about logarithms, $\log_{10}(n)$, $\log_2(n)$, and $\log_e(n)$, differ only by a multiplicative constant. Because we are ignoring multiplicative constants anyway, which base is used is irrelevant. The rules also indicate that $8 \cdot \log_2(n^5)$ also differs only by a multiplicative constant. All of these functions are include in $\Theta(\log(n))$. We will only include the base when the base matters. For example, 2^n and 10^n differ by much more than a multiplicative constant.

Some Constant: See $\Theta(1)$ above for what we mean by the "some constant" that can be multiplied by the $\log n$.

Examples Included:

- $f(n) = 7\log_2 n$ and $f(n) = 8.829\log_{10}(n)$
- $f(n) = \log(n^8)$ and $f(n) = \log(n^8 + 2n^3 - 10)$
- $f(n) = (8 + \sin(n))\log_2(n)$, $f(n) = 8\log_2 n + \frac{1}{n}$, and $f(n) = 3\log_2 n - 1,000,000$

Examples Not Included:

- $f(n) = -\log_2 n$ (fails $c > 0$)
- $f(n) = 5$ (too small)
- $f(n) = (\log_2 n)^2$ (too big)

Poly-Logarithms $\log^{\Theta(1)}(n)$: Powers of logs like $(\log n)^3$ are referred to as *poly-log*. These are often written as $\log^3 n = (\log n)^3$. Note that this is different than $\log(n^3) = 3\log n$. The class is denoted $\log^{\Theta(1)}(n)$. It is a super set of $\Theta(\log(n))$, because every function in $\Theta(\log(n))$ is also in $\log^{\Theta(1)}(n)$, but $f(n) = \log^3(n)$ is included in the later but not the first.

Examples Included:

- $f(n) = 7(\log_2 n)^5$, $f(n) = 7\log_2 n$, and $f(n) = 7\sqrt{\log_2 n}$
- $f(n) = 7(\log_2 n)^5 + 6(\log_2 n)^3 - 19 + 7(\log_2 n)^2/n$

Examples Not Included:

- $f(n) = n$ (too big)

Linear Functions $\Theta(n)$: Given an input of n items, it takes $\Theta(n)$ time simply to look at the input. Looping over the items and doing constant amount of work for each takes another $\Theta(n)$ time. Say, $t_1(n) = 2n$ and $t_2(n) = 4n$ for a total of $6n$ time. Now if you do not want to do more than linear time, are you allowed to do any more work? Sure. You can do something that takes $t_3(n) = 3n$ time and something else that takes $t_4(n) = 5n$ time. You are even allowed to do a few things that take a constant amount of time, totaling to say $t_5(n) = 13$. The entire algorithm then takes the sum of these $t(n) = 14n + 13$ time. This is still considered to be linear time.

Examples Included:

- $f(n) = 7n$ and $f(n) = 8.829n$
- $f(n) = (8 + \sin(n))n$ and $f(n) = 8n + \log^{10} n + \frac{1}{n} - 1,000,000$

Examples Not Included:

- $f(n) = -n$ and $f(n) = 0n$ (fails $c > 0$)
- $f(n) = \frac{n}{\log_2 n}$ (too small)
- $f(n) = n\log_2 n$ (too big)

Time for Sorting $\Theta(n\log n)$: Another running time that arises often in algorithms is $\Theta(n\log n)$. For example, this is the number of comparisons needed to sort n elements. What should be noted is that the function $f(n) = n\log n$ is just slightly too big to be in the linear class of functions $\Theta(n)$. This is because $n\log n$ is $\log n$ times n and $\log n$ is not constant.

Examples Included:

- $f(n) = 7n\log_2(n)$ and $f(n) = 8.829n\log_{10}(n) + 3n - 10$

Examples Not Included:

- $f(n) = n$ (too small)
- $f(n) = n\log^2 n$ (too big)

Time for Sorting $\Theta(n\log n)$: Another running time that

Quadratic Functions $\Theta(n^2)$: Two nested loops from 1 to n takes $\Theta(n^2)$ time if each inner iteration takes a constant amount of time. An $n \times n$ matrix requires $\Theta(n^2)$ space if each element takes constant space.

Examples Included:

- $f(n) = 7n^2 - 8n\log n + 2n - 17$

Examples Not Included:

- $f(n) = \frac{n^2}{\log_2 n}$ and $f(n) = n^{1.9}$ (too small)
- $f(n) = n^2\log_2 n$ and $f(n) = n^{2.1}$ (too big)

Cubic Functions $\Theta(n^3)$: Similarly, three nested loops from 1 to n and a $n \times n \times n$ cube matrix requires $\Theta(n^3)$ time and space.

$\tilde{\Theta}(n)$, $\tilde{\Theta}(n^2)$, $\tilde{\Theta}(n^3)$, and so on: Stating that the running time of an algorithm is $\Theta(n^2)$ allows us to ignore the multiplicative constant in front. Often, however, we will in fact ignore logarithmic factors. These factors might either arise because we say that the "size" of the input is n when in fact it consists of n elements each of $\Theta(\log n)$ bits or because we say that the algorithm does $\Theta(n^2)$ "operations" when in fact each operation consists of adding together $\Theta(\log n)$ bit values. The only place that this book cares about these factors is when it is determining whether the running time of sorting algorithms is $\Theta(n)$ or $\Theta(n \log n)$. When one does want to ignore these logarithmic factors, a notation sometimes used is $\tilde{\Theta}(n^2)$. It is a short form for $\log^{\pm\Theta(1)}(n) \times \Theta(n^2)$. Though sometimes maybe we should, we do not use this notation.

Examples Included:

- $f(n) = 7n^2 \log^7 n + 2n - 17$
- $f(n) = 7\frac{n^2}{\log^7 n} + 2n - 17$

Examples Not Included:

- $f(n) = n^{2.1}$ (too big)

Polynomial $n^{\Theta(1)}$: As the notation $n^{\Theta(1)}$ implies, the class of polynomials consists of functions $f(n) = n^c$ where $c > 0$ is "some constant" from $\Theta(1)$. For example, $\sqrt{n}$, n, n^2, n^3, but not $n^0 = 1$ or $n^{-1} = \frac{1}{n}$. By closing the class of functions under additions, this class includes functions like $f(n) = 4n^3 + 7n^2 - 2n + 8$. What about the function $f(n) = n \log n$? If the context was the study of polynomials, I would not consider this to be one. However, what we care about here is grow rate. Though $f(n) = n \log n$ belongs neither to the class $\Theta(n)$ nor to the class $\Theta(n^2)$, it is bounded below by the function $f_1(n) = n$ and above by $f_2(n) = n^2$. Both of these are included in the class $n^{\Theta(1)}$. Hence, $f(n) = n \log n$ must also have the right kind of grow rate and hence it will be included too. In contrast, the function $f(n) = \log n$ is not included, because to make $n^c \leq \log n$, c would have to be zero and $n^0 = 1$ is not included.

Definition of a Feasible Algorithm: An algorithm is considered to be *feasible* if it runs in polynomial time.

Examples Included:

- $f(n) = 7n^2 - 8n \log n + 2n - 17$
- $f(n) = n^2 \log_2 n$ and $f(n) = \frac{n^2}{\log_2 n}$
- $f(n) = \sqrt{n}$ and $f(n) = n^{3.1}$
- $f(n) = 7n^3 \log^7 n - 8n^2 \log n + 2n - 17$

Examples Not Included:

- $f(n) = \log n$ (too small)
- $f(n) = n^{\log n}$ and $f(n) = 2^n$ (too big)

Exponentials $2^{\Theta(n)}$: As the notation $2^{\Theta(n)}$ implies, the class of exponentials consists of functions $f(n) = 2^{cn}$ where $c > 0$ is "some constant" from $\Theta(1)$. For example, $2^{0.001n}$, 2^n, and 2^{2n}. Again, $2^{0n} = 1$ and $2^{-n} = \frac{1}{2^n}$ are not included.

Definition of an Infeasible Algorithm: An algorithm is considered to be *infeasible* if it runs in exponential time.

The Base: At first it looks like the function 4^n should not be in $2^{\Theta(n)}$, because the base is 4 not 2. However, $4^n = 2^{2n}$ and hence it is included. In general, the base of the exponential does not matter when determining whether it is exponential, as long as it is bigger then one. This is because for any $b > 1$, the function $f(n) = b^n = 2^{(\log_2 b)n}$ is included in $2^{\Theta(n)}$, because $(\log_2 b) > 0$ is a constant. (See the logarithmic rules above.) In contrast, 4^n is far to big to be in $\Theta(2^n)$ because $\frac{4^n}{2^n} >> \Theta(1)$.

Incorrect Notation $(\Theta(1))^n$: One might think that we would denoted this class of functions as $(\Theta(1))^n$. However, this would imply $f(n) \approx c^n$ for some constant $c > 0$. This would include $\left(\frac{1}{2}\right)^n$ which decreases with n.

Examples Included:

- $f(n) = 2^n$ and $f(n) = 3^{5n}$
- $f(n) = 2^n \cdot n^2 \log_2 n - 7n^8$ and $f(n) = \frac{2^n}{n^2}$

Examples Not Included:

- $f(n) = n^{1,000,000}$ (too small)
- $f(n) = n! \approx n^n = 2^{n \log_2 n}$ and $f(n) = 2^{n^2}$ (too big)

Double Exponentials $2^{2^{\Theta(n)}}$: These function grow very fast. The only algorithm that runs this slowly considered in this text is Ackermann's function. See Section 11.2.3.

1.4.2 Comparing The Classes of Functions

These classes, except when one is a subset of the other, have very different grow rates. Look at the chart above.

Linear vs Quadratic vs Cubic: Whether the running time is $T(n) = 2n^2$ or $T(n) = 4n^2$, though a doubling in time, will likely not change whether or not the algorithm is practical or not. On the other hand, $T_1(n) = n$ is a whole n times better than $T_2(n) = n^2$. Now, one might argue that an algorithm whose running time is $T_1(n) = 1,000,000n$ is worse than one whose running time is $T_2(n) = n^2$. However, when $n = 1,000,000$, the first one becomes faster. In our analysis, we will consider large values of n. Hence, we will consider $T_1(n) = 1,000,000n$ to be better than $T_2(n) = n^2$. Similarly, this is better than $T_3(n) = n^3$.

Poly-Logarithms vs Polynomials: For sufficiently large n, polynomials are always asymptotically bigger than any poly-logarithm. For example, $(\log n)^{1,000,000}$ is very big for a poly-logarithmic function, but it is considered to be much smaller than $n^{0.00001}$, which in turn is very small for a polynomial. Similarly, $n^4 \log^7 n$ is between n^4 and $n^{4+\epsilon}$ even for $\epsilon = 0.00000001$.

Polynomials vs Exponentials: For sufficiently large n, polynomials are always asymptotically smaller than any exponential. For example, $n^{1,000,000}$ is very big for a polynomial function, but is considered to be much smaller than $2^{0.00001n}$, which is very small for an exponential function.

When BigOh Notation Can Be Misleading:

- Sometimes constant factors matter. For example, if an algorithm requires $1,000,000,000 \cdot n$ time, then saying that it is $\mathcal{O}(n)$ is true but misleading. In practice, constants are generally small.
- One algorithm may only be better for unrealistically large input sizes. For example, $(\log n)^{100}$ has a lower complexity than n. However, n only gets bigger when n is approximately 2^{1024}. $(\log n)^{100} = (\log 2^{1024})^{100} = 1024^{100} = 2^{(10 \cdot 100)} < 2^{1024} = n$.

1.4.3 Other Useful Notations

The following are different notations for comparing the asymptotic grow of a function $f(n)$ to some "constant" times the function $g(n)$.

Greek Letter	Standard Notation	My Notation	Meaning
Theta	$f(n) = \Theta(g(n))$	$f(n) \in \Theta(g(n))$	$f(n) \approx c \cdot g(n)$
BigOh	$f(n) = \mathcal{O}(g(n))$	$f(n) \leq \mathcal{O}(g(n))$	$f(n) \leq c \cdot g(n)$
Omega	$f(n) = \Omega(g(n))$	$f(n) \geq \Omega(g(n))$	$f(n) \geq c \cdot g(n)$
Little Oh	$f(n) = o(g(n))$	$f(n) << o(g(n))$	$f(n) << g(n)$
Little Omega	$f(n) = \omega(g(n))$	$f(n) >> \omega(g(n))$	$f(n) >> g(n)$

Examples:

Same: $7 \cdot n^3$ is within a constant of n^3. Hence, it is in $\Theta(n^3)$, $\mathcal{O}(n^3)$, and $\Omega(n^3)$. However, because it is not much smaller than n^3, it not in $o(n^3)$ and because it is not much bigger, it not in $\omega(n^3)$.

Smaller: $7 \cdot n^3$ is asymptotically much smaller than n^4. Hence, it is in $\mathcal{O}(n^4)$ and in $o(n^4)$, but it is not in $\Theta(n^4)$, $\Omega(n^4)$, or $\omega(n^4)$.

Bigger: $7 \cdot n^3$ is asymptotically much bigger than n^2. Hence, it is in $\Omega(n^2)$ and in $\omega(n^2)$, but it is not in $\Theta(n^2)$, $\mathcal{O}(n^2)$, or in $o(n^2)$

You should always give the smallest possible approximation. Even when someone asks for BigOh notation, what they really want is Theta notation.

Notation Considerations:

"$\in$" vs "$=$": I consider $\Theta(n)$ to be a class of functions. Given this one should use the set notation, $f(n) \in \Theta(g(n))$, to denote membership.

On the other hand, the meaning is that, ignoring constant multiplicative factors $f(n)$ has the same asymptotic growth as $g(n)$. Given this, the notation $7n = \Theta(n)$ makes sense. This notation is standard.

Even the statements $3n^2+5n-7 = n^{\Theta(1)}$ and $2^{3n} = 2^{\Theta(n)}$ make better sense when when you think of the symbol Θ to mean "some constant." However, be sure to remember that $4^n \cdot n^2 = 2^{\Theta(n)}$ is also true.

"$=$" vs "$\leq$": $7n = \mathcal{O}(n^2)$ is also standard notation. This makes less sense to me. Because it means that $7n$ is at most some constant times n^2, a better notation would be $7n \leq \mathcal{O}(n^2)$. The standard notation is even more awkward, because $\mathcal{O}(n) = \mathcal{O}(n^2)$ should be true, but $\mathcal{O}(n^2) = \mathcal{O}(n)$ should be false. What sense does this make?

Running Time of an Algorithm:

Θ: "My algorithm runs in $\Theta(n^2)$ time" means that it runs in $c \times n^2$ time for some constant c.

$\mathcal{O}$: "My algorithm runs in $\mathcal{O}(n^2)$ time" means that it was difficult to determine exactly how long the algorithm will run. You have managed to prove that for every input, it does not take any more than this much time. However, it might actually be the case that for every input it runs faster than this.

Ω: "My algorithm runs in $\Omega(n^2)$ time" means that you have found an input for which your algorithm takes more than some constant c times n^2 time. It may take even longer.

o: Suppose that you are proving that your algorithm runs in $\Theta(n^3)$ time, and there is a subroutine that gets called once that requires only $\Theta(n^2)$ time. Because $n^2 = o(n^3)$, its running time is insignificant to the over all time. Hence, you would say, "Don't worry about the subroutine, because it runs in $o(n^3)$ time."

Time Complexity of a Problem: The time complexity of a computational problem is the time of the fastest algorithm to solve the problem.

$\mathcal{O}$: You say a problem has time complexity $\mathcal{O}(n^2)$ when you have a $\Theta(n^2)$ algorithm for it, because there may be a faster algorithm to solve the problem that you do not know about.

Ω: To say that the problem has time complexity $\Omega(n^2)$ is a big statement. It not only means that nobody knows a faster algorithm; it also means that no matter how smart you are, it is impossible to come up with a faster algorithm, because such an algorithm simply does not exist.

Θ: "The time complexity of the problem is $\Theta(n^2)$" means that you have an upper bound and a lower bound on the time which match. The upper bound provides an algorithm, and the lower bound proves that no algorithm could do better.

1.4.4 Different Levels of Detail When Classifying a Function

One can decide how much information about a function they want to reveal. We have already discussed how we could reviel only that the function $f(n) = 7n^2 + 5n$ is a polynomial by stating that it is in $n^{\Theta(1)}$ or reviel more that it is a quadratic by stating it is in $\Theta(n^2)$. However, there are even more options. We could give more detail by saying that $f(n) = (7 + o(1))n^2$. This is a way of writing $7n^2 + o(n^2)$. It reviels that the constant in front of the high-order term is 7. We are told that there may be low-order terms, but not what they are. Even more details would be given by saying that $f(n) = 7n^2 + \Theta(n)$.

As another example, let $f(n) = 7 \cdot 2^{3n^5} + 8n^4 \log^7 n$. The following classifications of functions are sorted from the least inclusive to the most inclusive, meaning that every function included in the first is also included in the second, but not vice versa. They all contain f, giving more and more details about it.

$$\begin{aligned}
f(n) &= 7 \cdot 2^{3n^5} + 8n^4 \log^7 n \\
&\in \; 7 \cdot 2^{3n^5} + \Theta(n^4 \log^7 n) \\
&\subseteq \; 7 \cdot 2^{3n^5} + \tilde{\Theta}(n^4) \\
&\subseteq \; 7 \cdot 2^{3n^5} + n^{\Theta(1)} \\
&\subseteq \; (7 + o(1))2^{3n^5} \\
&\subseteq \; \Theta(2^{3n^5}) \\
&\subseteq \; 2^{\Theta(n^5)} \\
&\subseteq \; 2^{n^{\Theta(1)}}
\end{aligned}$$

Exercise 1.4.1 *(See solution in Section 20) For each of these classes of functions, give a function that is in it but not in the next smaller class.*

1.4.5 Constructing and Deconstructing the Classes of Functions

We will understand better which functions are in each of these classes, by carefully considering how both the classes and the functions within them are constructed and conversely, deconstructed.

The Class of Functions $\Theta(g(n))$: Above we discussed the classes of functions $\Theta(1)$, $\Theta(\log n)$, $\Theta(n)$, $\Theta(n \log n)$, $\Theta(n^2)$, and $\Theta(n^3)$. Of course, we could similarly define $\Theta(n^4)$. How about $\Theta(2^n \cdot n^2 \cdot \log^3 n)$? Yes, this is a class of functions too. For every function $g(n)$, $\Theta(g(n))$ is the class of functions that are similar within a constant factor in asymptotic growth rate to $g(n)$. In fact, one way to think of the class $\Theta(g(n))$ is $\Theta(1) \cdot g(n)$. Note that the classes $\log^{\Theta(1)}(n)$, $n^{\Theta(1)}$, and $2^{\Theta(n)}$ do not have this form. We will cover them next.

Constructing The Functions in The Class: A class of functions like $\Theta(n^2)$ is constructed by starting with some function $g(n) = n^2$ and from it constructing other functions by repeatedly doing one of the following:

Closed Under Scalar Multiplication: You can multiply any function $f(n)$ in the class by a strictly positive constant $c > 0$. This gives us that $7n^2$ and $9n^2$ are in $\Theta(n^2)$.

Closed Under Addition: The class of functions $\Theta(g(n))$ is closed under addition. What this means is that if $f_1(n)$ and $f_2(n)$ are both functions in the class, then their sum $f_1(n) + f_2(n)$ is also in it. More over, if $f_2(n)$ grows too slowly to be in the class then $f_1(n) + f_2(n)$ is still in it and so is $f_1(n) - f_2(n)$. For example, $7n^2$ being in $\Theta(n^2)$ and $3n + 2$ growing smaller gives that $7n^2 + 3n + 2$ and $7n^2 - 3n - 2$ are in $\Theta(n^2)$.

Bounded Between: You can also throw in any function that is bounded below and above by functions that you have constructed before. $7n^2$ and $9n^2$ being in $\Theta(n^2)$ gives that $(8 + sin(n))n^2$ is in $\Theta(n^2)$. Note that $7n^2 + 100n$ is also in $\Theta(n^2)$ because it is bounded below by $7n^2$ and above by $8n^2 = 7n^2 + 1n^2$, because the $1n^2$ eventually dominates the $100n$.

Determining Which Class $\Theta(g(n))$ of Functions $f(n)$ is in: One task that you will often be asked to do is given a function $f(n)$ determine which "Θ" class it is in. This amounts to finding a function $g(n)$ that is as simple as possible and for which $f(n)$ is contained in $\Theta(g(n))$. This can be done by deconstructing $f(n)$ in the opposite way that we constructed functions to put into the class of functions $\Theta(g(n))$.

Drop Low Order Terms: If $f(n)$ is a number of things added or subtracted together, then each of these things is called at *term*. Determine which of the terms grows the fastest. The slower growing terms are referred to as *low-order terms*. Drop them.

Drop Multiplicative Constant: Drop the multiplicative constant in front of the largest term.

Examples:

- Given $3n^3 \log n - 1000n^2 + n - 29$, the terms are $3n^3 \log n$, $1000n^2$, n, and 29. Dropping the low order terms gives $3n^3 \log n$. Dropping the multiplicative constant 3, gives $n^3 \log n$. Hence, $3n^3 \log n - 1000n^2 + n - 29$ is in the class $\Theta(n^3 \log n)$.
- Given $7 \cdot 4^n \cdot n^2 / \log^3 n + 8 \cdot 2^n + 17 \cdot n^2 + 1000 \cdot n$, the terms are $7 \cdot 4^n \cdot n^2 / \log^3 n$, $8 \cdot 2^n$, $17 \cdot n^2$, and $1000 \cdot n$. Dropping the low order terms gives $7 \cdot 4^n \cdot n^2 / \log^3 n$. Dropping the multiplicative constant 7, gives $4^n \cdot n^2 / \log^3 n$. Hence, this function $7 \cdot 4^n \cdot n^2 / \log^3 n + 8 \cdot 2^n + 17 \cdot n^2 + 1000 \cdot n$ is in the class $\Theta(4^n \cdot n^2 / \log^3 n)$.
- $\frac{1}{n} + 18$ is in the class $\Theta(1)$. Since $\frac{1}{n}$ is a lower-order term than 18, it is dropped.
- $\frac{1}{n^2} + \frac{1}{n}$ is in the class $\Theta(\frac{1}{n})$ because $\frac{1}{n^2}$ is a smaller term.

The Class of Functions $(g(n))^{\Theta(1)}$: Above we discussed the classes of functions $\log^{\Theta(1)}(n)$, $n^{\Theta(1)}$, and $2^{\Theta(n)}$. Instead of forming the class $\Theta(g(n))$ by multiplying some function $g(n)$ by a constant from $\Theta(1)$, the form of these classes is $(g(n))^{\Theta(1)}$ and hence are formed by raising some function $g(n)$ to the power of some constant from $\Theta(1)$.

Constructing The Functions in The Class: The poly-logarithms $\log^{\Theta(1)}(n)$, polynomial $n^{\Theta(1)}$, and exponential $2^{\Theta(n)}$ are constructed by starting respectively with the functions $g(n) = \log n$, $g(n) = n$, and $g(n) = 2^n$ and from them constructing other functions by repeatedly doing one of the following:

Closed Under Scalar Multiplication and Additions: As with $\Theta(g(n))$, if $f_1(n)$ is in the class and $f_2(n)$ is either in it or grows too slowly to be in it, then $cf_1(n)$, $f_1(n) + f_2(n)$, and $f_1(n) - f_2(n)$ are also in it. For example, n^2 and n being in $n^{\Theta(1)}$ and $\log n$ growing smaller gives that $7n^2 - 3n + 8 \log n$ is in it as well.

Closed Under Multiplication: Unlike $\Theta(g(n))$, these classes are closed under multiplication as well. Hence, $f_1(n) \cdot f_2(n)$ is in it as well, and if $f_2(n)$ is sufficiently smaller that everything does not cancel, then so is $\frac{f_1(n)}{f_2(n)}$. For example, n^2 and n being in $n^{\Theta(1)}$ and $\log n$ growing smaller gives that $n^2 \cdot n = n^3$, $\frac{n^2}{n} = n$, $n^2 \cdot \log n$, and $\frac{n^2}{\log n}$ are in it as well.

Closed Under Raising to Constant $c > 0$ Powers: Functions in these classes can be raised to any constant $c > 0$. This is why $\log^c(n)$ is in $\log^{\Theta(1)}(n)$, n^c is in $n^{\Theta(1)}$, and $(2^n)^c = 2^{cn}$ is in $2^{\Theta(n)}$.

Different Constant $b > 1$ Bases: As shown in Section 1.2, $b^n = 2^{\log_2(b)n}$. Hence, as long as $b > 1$, it does not matter what the base is.

Bounded Between: As with $\Theta(g(n))$, you can also throw in any function that is bounded below and above by functions that you have constructed before. For example, $n^2 \cdot \log n$ is in $n^{\Theta(1)}$ because it is bounded between n^2 and n^3.

Determining Whether $f(n)$ is in the Class: You can determine whether $f(n)$ is in $\log^{\Theta(1)}(n)$, $n^{\Theta(1)}$, or $2^{\Theta(n)}$ by deconstructing it.

Drop Low Order Terms: As with $\Theta(g(n))$, the first step is to drop the low order terms.

Drop Low Order Multiplicative Factors: Now instead of just multiplicative constants in front of the largest term, we are able to drop low order multiplicative factors.

Drop Constants $c > 0$ in the Exponent: Drop the 3 in $\log^3 n$, n^3, and 2^{3n}, but not the -1 in 2^{-n}.

Change Constants $b > 1$ in the Base to 2: Change 3^n to 2^n, but not $(\frac{1}{2})^n$.

Examples:

- Given $3n^3 \log n - 1000n^2 + n - 29$, we drop the low order terms to get $3n^3 \log n$. The factors of this term are 3, n^3, and $\log n$. Dropping the low order ones gives n^3. This is n to the power of a constant. Hence, $3n^3 \log n - 1000n^2 + n - 29$ is in the class $n^{\Theta(1)}$.
- Given $7 \cdot 4^n \cdot n^2/\log^3 n + 8 \cdot 2^n + 17 \cdot n^2 + 1000 \cdot n$, we drop the low order terms to get $7 \cdot 4^n \cdot n^2/\log^3 n$. The factors of this term are 7, 4^n, n^2, and $\log^3 n$. Dropping the low order ones gives 4^n. Changing the base gives 2^n. Hence, $7 \cdot 4^n \cdot n^2/\log^3 n + 8 \cdot 2^n + 17 \cdot n^2 + 1000 \cdot n$ is in the class $2^{\Theta(1)}$.

1.4.6 The Formal Definition of Θ and $\mathcal{O}$ Notation

We have given a lot of intuition about the above classes. We will not give the formal definitions of these classes.

The Formal Definition of $\Theta(g(n))$: Informally, $\Theta(g(n))$ is the class of functions that are some "constant" times the function $g(n)$. This is complicated by the fact that the "constant" can be anything $c(n)$ in $\Theta(1)$, which includes things like $7 + \sin(n)$ and $7 + \frac{1}{n}$. The formal definition, which will be explained in parts below, for the function $f(n)$ to be included in the class $\Theta(g(n))$ is

$$\exists c_1, c_2 > 0,\ \exists n_0, \forall n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Concrete Examples: To make this as concrete as possible for you, substitute in the constant 1 for $g(n)$ and in doing so define $\Theta(1)$. The name *constant* for this class of functions is a bit of a misnomer. *Bounded between strictly positive constants* would be a better name. Once you understand this definition within this context, try another one. For example, set $g(n) = n^2$ in order to define $\Theta(n^2)$.

Bounded Below and Above by a Constant Times $g(n)$: We prove that the asymptotic growth of the function $f(n)$ is comparable to a constant c times the function $g(n)$, by proving that it is *bounded below* by c_1 times $g(n)$ for a sufficiently small constant c_1 and *bounded above* by c_2 times $g(n)$ for a sufficiently large constant c_2. See Figure 1.3. This is expressed in the definition by

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

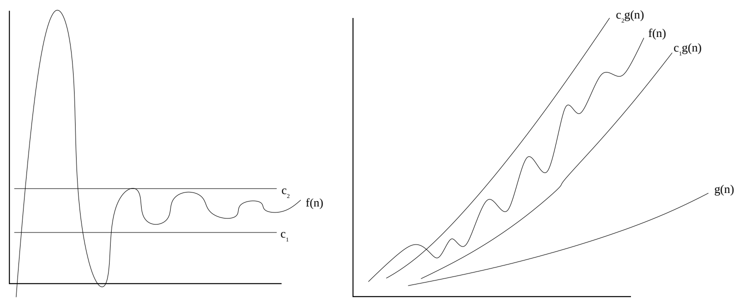


Figure 1.3: The left figure shows a function $f(n) \in \Theta(1)$. For sufficiently large n, it is bounded above and below by some constant. The right figure shows a function $f(n) \in \Theta(g(n))$. For sufficiently large n, it is bounded above and below by some constant times $g(n)$.

Requirements on c_1 and c_2:

Strictly Positive: Because we want to include neither $f(n) = -1 \cdot g(n)$ nor $f(n) = 0 \cdot g(n)$, we require that $c_1 > 0$. Similarly, the function $f(n) = \frac{1}{n} \cdot g(n)$ is not included, because this is bounded below by $c_1 \cdot g(n)$ only when $c_1 = 0$.

Allowing Big and Small Constants: We might not care whether the multiplicative constant is 2, 4, or even 100. But what if it is 1,000, or 1,000,000, or even $10^{10^{10}}$? Similarly, what if it is 0.01, or 0.00000001, or even 10^{-100}? Surely, this will make a difference? Despite the fact that these concerns may lead to miss leading statements at times, all of these are included.

An Unbounded Class of Bounded Functions: One might argue that allowing $\Theta(1)$ to include arbitrarily large values contradicts the fact that the functions in are bounded. However, this is a mistake. Though it is true that the class of function in not bounded, each individual function included is bounded. For example, though $f(n) = 10^{10^{10}}$ big, in itself, it is bounded.

Reasons: There are a few reasons for including all of these big constants. First, if you were to set a limit, what limit would you set. Every application has its own definitions of "reasonable" and even these are open for debate. In contrast, including all strictly positive constants c is a much cleaner mathematical statement. A final reason for including all of these big constants is because they do not, in fact, arrive often and hence are not a big concern. In practice, one writes $\Theta(g(n))$ to mean $g(n)$ times a "reasonable" constant. If it is an unreasonable constant, one will still write $\Theta(g(n))$, but will include a foot note to the fact.

Existential Quantifier $\exists c_1, c_2 > 0$: No requirements are made on the constants c_1 and c_2 other then them being strictly positive. It is sufficient that constants that do the job exist. Formally, we write

$$\exists c_1, c_2 > 0,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$$

In practice, you could assume that $c_1 = 0.00000001$ and $c_2 = 1,000,000$, however, if asked for the bounding constants for a given function, it is better to give constants that bound as tightly as possible.

Which Input Values n:

Universal Quantifier $\forall n$: We have not spoken yet about for which input n, the function $f(n)$ must be bounded. Ideally for all inputs. Formally, this would be

$$\exists c_1, c_2 > 0,\ \forall n,\ c_1 \leq f(n) \leq c_2$$

Order of Quantifiers Matters: The following is wrong.

$$\forall n,\ \exists c_1, c_2 > 0,\ c_1 \leq f(n) \leq c_2$$

The first requires the existence of one constant c_1 and one constant c_2 that bound the function for all input n. The second allows there to be a different constant for all n. This is much easier to do. For example, the following is true. $\forall n,\ \exists c_1, c_2 > 0,\ c_1 \leq n \leq c_2$. Namely, given an n, simply set $c_1 = c_2 = n$.

Sufficiently Large n: Requiring the function to be bounded for all input n, excludes the function $f(n) = \frac{1}{n-3} + 1$ from $\Theta(1)$, because it goes to infinity for the one moment that $n = 3$. It also excludes $f(n) = 2n^2 - 6n$ from $\Theta(n^2)$, because it is negative until n is 3. Now one may fairly argue that such functions should not be included, just as one may argue whether a platypus is a mammal or a tomato is a fruit. However, it turns out to be useful to include them. Recall, when classifying functions, we are not considering its behavior on small values of n or even whether it is monotone increasing, but on how quickly it grows when its input n grows really big. All three of the examples above are bounded for all sufficiently large n. This is why they are included. Formally, we write

$$\exists c_1, c_2 > 0,\ \forall n \geq n_0,\ c_1 \leq f(n) \leq c_2$$

where n_0 is our definition of "sufficiently large" input n.

For Some Definition of Sufficiently Large $\exists n_0$: Like with the constants c_1 and c_2, different applications have different definitions of how large n needs to be to be "sufficiently large". Hence, to make the mathematics clean, we will simply require that there exists some definition n_0 of sufficiently large that works.

The Formal Definition of $\Theta(g(n))$: This completes the discussion of the formal requirement for the function $f(n)$ to be included in $\Theta(g(n))$.

$$\exists c_1, c_2 > 0,\ \exists n_0, \forall n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$$

The Formal Definitions of BigOh and Omega: The definition of $f(n) = \mathcal{O}(g(n))$ includes only the lower bound part of the Theta definition.

$$\exists c > 0,\ \exists n_0,\ \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)$$

Similarly, the definition of $f(n) = \Omega(g(n))$ includes only the upper bound part.

$$\exists c > 0,\ \exists n_0,\ \forall n \geq n_0,\ f(n) \geq c \cdot g(n)$$

Note that $f(n) = \Theta(g(n))$ is true if both $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$ are true.

The Formal Definition of Polynomial $n^{\Theta(1)}$ and of Exponential $2^{\Theta(n)}$: The function $f(n)$ is included in the class of polynomials $n^{\Theta(1)}$ if

$$\exists c_1, c_2 > 0,\ \exists n_0, \forall n \geq n_0,\ n^{c_1} \leq f(n) \leq n^{c_2}$$

and in the class of exponentials $2^{\Theta(n)}$ if

$$\exists c_1, c_2 > 0,\ \exists n_0, \forall n \geq n_0,\ 2^{c_1 n} \leq f(n) \leq 2^{c_2 n}$$

Bounded Below and Above: The function $f(n) = 2^n \cdot n^2$ is in $2^{\Theta}(n)$ because it is bounded below by 2^n and above by $2^{2n} = 2^n \cdot 2^n$, because the 2^n eventually dominates the factor of n^2.

The Formal Definitions of Little Oh and Little Omega: Another useful way to compare $f(n)$ and $g(n)$ is to look at the ratio between them for extremely large values of n.

Class	$\lim_{n\Rightarrow\infty} \frac{f(n)}{g(n)} =$	A practically equivalent definition
$f(n) = \Theta(g(n))$	Some constant	$f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$
$f(n) = o(g(n))$	Zero	$f(n) = \mathcal{O}(g(n))$, but $f(n) \neq \Omega(g(n))$
$f(n) = \omega(g(n))$	∞	$f(n) \neq \mathcal{O}(g(n))$, but $f(n) = \Omega(g(n))$

- $2n^2 + 100n = \Theta(n^2)$ and $\lim_{n\Rightarrow\infty} \frac{2n^2+100n}{n^2} = 2$.
- $2n^3 + 100n = \omega(n^2)$ and $\lim_{n\Rightarrow\infty} \frac{2n^3+100n}{n^2} = \infty$.
- $2n + 100 = o(n^2)$ and $\lim_{n\Rightarrow\infty} \frac{2n+100}{n^2} = 0$.
- Another possibility not listed is that the limit $\lim_{n\Rightarrow\infty} \frac{f(n)}{g(n)}$ does not exit, because the ratio oscillates between two bounded constants c_1 and c_2. See Figure 1.3. In this case, $f(n) = \Theta(g(n))$. This occurs with Sine, but will not occur for functions that are expressed with n, real constants, plus, minus, times, divide, exponentiation, and logarithms. See Figure 1.8.

1.4.7 Formal Proofs

Sometimes you are asked to formally proof that a function $f(n)$ is in $\Theta(g(n)$ or that it is not.

Proving $f(n) \in \Theta(g(n)$: Use the prover-adversary game for proving statements with existential and universal quantifiers to prove the statement $\exists c_1, c_2 > 0,\ \exists n_0, \forall n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$.

- You as the prover provide c_1, c_2, and n_0.
- Some adversary gives you an n that is at least your n_0.
- You then prove that $c_1 g(n) \leq f(n) \leq c_2 g(n)$.

Example 1: For example, $2n^2 + 100n = \Theta(n^2)$. Let $c_1 = 2$, $c = 3$ and $n_0 = 100$. Then, for all $n \geq 100$, $c_1 g(n) = 2n^2 \leq 2n^2 + 100n = f(n)$ and $f(n) = 2n^2 + 100n \leq 2n^2 + n \cdot n = 3n^2 = c_2 g(n)$. The values of c_1, c_2, and n_0 are not unique. For example, $n_0 = 1$, $c_2 = 102$, and $n_0 = 1$ also work because for all $n \geq 1$, $f(n) = 2n^2 + 100n \leq 2n^2 + 100n^2 = 102n^2 = c_2 g(n)$.

Example 2: Another example is $2n^2 - 100n \in \Theta(n^2)$. This is negative for small n, but is positive for sufficiently large n. Here we could set $c_1 = 1$, $c_2 = 2$, and $n_0 = 100$. Then, for all $n \geq 100$, $c_1 g(n) = 1n^2 = 2n^2 - n \cdot n \leq 2n^2 - 100n = f(n)$ and $f(n) \leq 2n^2 = c_2 g(n)$.

Negation of $f(n) \in \Theta(g(n)$ is $f(n) \notin \Theta(g(n)$: Recall that to take the negation of a statement, you must replace the existential quantifiers with universal quantifiers and vice versa. Then you move the negation in to the right. The formal negation of

$$\exists c_1, c_2 > 0,\ \exists n_0, \forall n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$$

is

$$\forall c_1, c_2 > 0,\ \forall n_0,\ \exists n \geq n_0,\ [c_1 g(n) > f(n) \text{ or } f(n) > c_2 g(n)]$$

Note that $f(n)$ can either be excluded because it is too small or because it is too big. To see why is it not $\forall c \leq 0$ review the point "The Domain Does Not Change" within the "Existential and Universal Quantifiers" section.

Proving $f(n) \notin \Theta(g(n)$:

- Some adversary is trying to prove that $f(n) \in \Theta(g(n)$. You are trying to prove him wrong.
- Just as you did you when you were proving $f(n) \in \Theta(g(n))$, the first step for the adversary is to give you constants c_1, c_2, and n_0. To be fair to the adversary, you must let him choose any constants he likes.
- You as the prover must consider his c_1, c_2, and n_0 and then come up with an n. There are two requirements on this n.
  - You must be able to bound $f(n)$. For this you make need to make n large and perhaps even be selective as to which n you take.
  - You must respect the adversary's request that $n \geq n_o$.

  One option is to choose an n that is the maximum between the value that meets the first requirement and the value meeting the second.
- Finally, you must either prove that $f(n)$ is too small, namely that $c_1 g(n) > f(n)$, or that it is too big, namely that $f(n) > c_2 g(n)$.

Example Too Big: We can prove that $f(n) = 14n^8 + 100n^6 \notin \Theta(n^7)$ as follows. Let c_1, c_2, and n_0 be arbitrary values. Here, the issue is that $f(n)$ is too big. Hence, we will ignore the constant c_1 and let $n = \max(c_2, n_0)$. Then we have that $f(n) = 14n^8 + 100n^6 > n \cdot n^7 \geq c_2 \cdot n^7$.

Similarly, we can prove that $2^{2n} \neq \Theta(2^n)$ as follows: Let c_1, c_2, and n_0 be arbitrary values. Let $n = \max(1 + \log_2 c_2, n_0)$. Then we have that $f(n) = 2^{2n} = 2^n \cdot 2^n > c_2 \cdot g(n)$.

Example Too Small: We can prove that $f(n) = 14n^8 + 100n^6 \notin \Theta(n^9)$ as follows. Let c_1, c_2, and n_0 be arbitrary values. Here, the issue is that $f(n)$ is too small. Hence, we will ignore the constant c_2. Note that the adversary's goal is to prove that $c_1 n^9$ is smaller than $f(n)$ and hence will likely choose to make c_1 something like 0.00000001. The smaller he makes his c_1, the larger we will have to make n. Let us make $n = \max(\frac{15}{c_1}, 11, n_0)$. Then we demonstrate that $c_1 g(n) = c_1 n^9 = n \cdot c_1 n^8 \geq \frac{15}{c_1} \cdot c_1 n^8 = 14n^8 + n^8 = 14n^8 + n^2 \cdot n^6 \geq 14n^8 + (11)^2 \cdot n^6 > 14n^8 + 100n^6 = f(n)$.

1.4.8 Solving Equations by Ignoring Details

One technique for solving equations is with the help of Theta notation. Using this technique, you can keep making better and better approximations of the solution until it is as close to the actual answer as you like.

For example, consider the equation $x = 7y^3(\log_2 y)^{18}$. It is in fact impossible to solve this equation for y. However, we will be able to approximate the solution. We have learned that the y^3 is much more significant than the $(\log_2 y)^{18}$. Hence, our first approximation will be $x = 7y^3(\log_2 y)^{18} \in y^{3+o(1)} = \left[y^3, y^{3+\epsilon}\right]$. This approximation can be solved easily, namely, $y = x^{\frac{1}{3+o(1)}}$. This is a fairly good approximation, but maybe we can do better.

Above, we ignored the factor $(\log_2 y)^{18}$ because it made the equation hard to solve. However, we can use what we learned above to approximate it in terms of x. Substituting in $y = x^{\frac{1}{3+o(1)}}$ gives $x = 7y^3(\log_2 y)^{18} = 7y^3(\log_2 x^{\frac{1}{3+o(1)}})^{18} = \frac{7}{(3+o(1))^{18}} y^3 (\log_2 x)^{18}$. This approximation can be solved easily, namely, $y = \frac{(3+o(1))^6}{7^{\frac{1}{3}}} \frac{x^{\frac{1}{3}}}{(\log x)^6} = \left(\frac{3^6}{7^{\frac{1}{3}}} + o(1)\right) \frac{x^{\frac{1}{3}}}{(\log x)^6} = \Theta\left(\frac{x^{\frac{1}{3}}}{(\log x)^6}\right)$.

In the above calculations, we ignored the constant factors c. Sometimes, however, this constant CANNOT be ignored. For example, suppose that $y = \Theta(\log x)$. Which of the following is true: $x = \Theta(2^y)$ or $x = 2^{\Theta(y)}$? Effectively, $y = \Theta(\log x)$ means that $y = c \cdot \log x$ for some unknown constant c. This gives $\log x = \frac{1}{c} y$ and $x = 2^{\frac{1}{c} y} = 2^{\Theta(y)}$. It also gives that $x = \left(2^{\frac{1}{c}}\right)^y \neq \Theta(2^y)$, because we do not know what c is.

With practice, you will be able to determine quickly what matters and what does not matter when solving equations.

You can use similar techniques to solve something like $100n^3 = 3^n$. Setting $n = 10.65199$ gives $100n^3 = 3^n = 120,862.8$. You can find this using binary search or using Newton's method. My quick and dirty method uses approximations. As before, the first step is to note that 3^n is more significant than $100n^3$. Hence, the "n" in the first is more significant than the one in the second. Let's denote the more significant "n" with n_{i+1} and the less significant one with n_i. This gives $100(n_i)^3 = 3^{n_{i+1}}$. Solving for the most significant "n" gives $n_{i+1} = \frac{\log(100) + 3\log(n_i)}{\log 3}$. Thus we have better and better approximations of n. Plugging in $n_1 = 1$ gives $n_2 = 4.1918065$, $n_3 = 8.1052849$, $n_4 = 9.9058778$, $n_5 = 10.453693$, $n_6 = 10.600679$, $n_7 = 10.638807$, $n_8 = 10.648611$, $n_9 = 10.651127$, $n_{10} = 10.651772$, $n_{11} = 10.651937$, $n_{12} = 10.651979$, and $n_{13} = 10.65199$.

Exercise 1.4.2 *Let x be a real value. As you know, $\lfloor x \rfloor$ rounds it down to the next integer. Explain what each of the following do: $2 \cdot \lfloor \frac{x}{2} \rfloor$, $\frac{1}{2} \cdot \lfloor 2 \cdot x \rfloor$, and $2^{\lfloor \log_2 x \rfloor}$.*

Exercise 1.4.3 *Let $f(n)$ be a function. As you know, $\Theta(f(n))$ drops low order terms and the leading coefficient. Explain what each of the following do: $2^{\Theta(\log_2 f(n))}$, and $\log_2(\Theta(2^{f(n)}))$. For each, explain abstractly how the function is approximated.*

1.5 Adding Made Easy Approximations

Sums arise often in the study of computer algorithms. For example, if the i^{th} iteration of a loop takes time $f(i)$ and it loops n times, then the total time is $f(1) + f(2) + f(3) + \ldots f(n)$. This we denoted by $\sum_{i=1}^{n} f(i)$. Note that, even though the individual terms are indexed by i, the total is a function of n.

The goal now is to approximate $\sum_{i=1}^{n} f(i)$ for various functions $f(i)$. We will first summarize the results and of this section. Then we will provide the classic techniques for computing $\sum_{i=1}^{n} 2^i$, $\sum_{i=1}^{n} i$, and $\sum_{i=1}^{n} \frac{1}{i}$. Beyond knowing these techniques, we do not cover how to evaluate sums exactly, but only how to approximate them to within a constant factor. We will give a few easy rules with which you will be able to compute $\Theta(\sum_{i=1}^n f(i))$ for almost every function $f(n)$ that you will encounter.

1.5.1 Summary

This subsection summarizes what will be discussed a length within in this section.

Example: Consider the following nested loops.

```
algorithm Eg(n)
    loop i = 1..n
        loop j = 1..i
            loop k = 1..j
                put "Hi"
            end loop
        end loop
    end loop
end algorithm
```

The inner loop requires time $\sum_{k=1}^j 1 = j$.
The next requires $\sum_{j=1}^i \sum_{k=1}^j 1 = \sum_{j=1}^i j = \Theta(i^2)$.
The total is $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j 1 = \sum_{i=1}^n \Theta(i^2) = \Theta(n^3)$.

Range of Examples: The following examples, include the classic sums as well as some other examples designed to demonstrate the various patterns that the results fall into. The examples are sorted from the sum of very fast growing functions to the sum of very fast decreasing functions.

Classic Geometric Increasing:	$\sum_{i=0}^n 2^i$	$= 2 \cdot 2^n - 1 = \Theta(2^n)$	$= \Theta(f(n))$.
Polynomial:	$\sum_{i=1}^n i^2$	$= \frac{1}{6}(2n^3 + 3n^2 + n) = \Theta(n^3)$	$= \Theta(n \cdot f(n))$.
Classic Arithmetic:	$\sum_{i=1}^n i$	$= \frac{1}{2}n(n+1) = \Theta(n^2)$	$= \Theta(n \cdot f(n))$.
Constant:	$\sum_{i=1}^n 1$	$= n = \Theta(n)$	$= \Theta(n \cdot f(n))$.
Above Harmonic:	$\sum_{i=1}^n \frac{1}{n^{0.999}}$	$\approx 1{,}000\ n^{0.001} = \Theta(n^{0.001})$	$= \Theta(n \cdot f(n))$.
Harmonic:	$\sum_{i=1}^n \frac{1}{n}$	$= \log_e(n) + \Theta(1)$	$= \Theta(\log n)$.
Below Harmonic:	$\sum_{i=1}^n \frac{1}{n^{1.001}}$	$\approx 1{,}000$	$= \Theta(1)$.
1 over a Polynomial:	$\sum_{i=1}^n \frac{1}{n^2}$	$\approx \frac{\Pi}{6} \approx 1.5497..$	$= \Theta(1)$.
Classic Geometric Decreasing:	$\sum_{i=1}^n (\frac{1}{2})^i$	$= 1 - (\frac{1}{2})^n$	$= \Theta(1)$.

Four Different Solutions: All of above examples, in fact all sums that we will consider, have one of four different types of solutions. These results are summarized in the following table.

Determining the Solution: Given a function $f(n)$, one needs to determine which of these four classes the function falls into. For each, a simple rule then provides the approximation of the sum $\Theta(\sum_{i=1}^n f(i))$.

Functions with a Basic Form: Most functions that you will need to sum up will fit into the basic form $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$, where a, b, d, and e are real constants. The table provides which ranges of values lead to each type of solution.

More General Functions using Θ Notation: These rules can be generalized using Theta notation even further to include an even wider range of functions $f(n)$. These generalizations, however, only work for simple analytical functions.

Definition of Simple Analytical Function $f(n)$: A function is said to be *simple analytical* if it can be expressed with n, real constants, plus, minus, times, divide, exponentiation, and logarithms. Oscillating functions like sine and cosine, complex numbers, and non-differentiable functions are not allowed.

See Section 1.5.3 for an explanation of these general classifications.

Geometric Increasing	Arithmetic	Harmonic	Bounded Tail
$\sum_{i=1}^{n} f(i) = \Theta(f(n))$	$\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$	$\sum_{i=1}^{n} f(i) = \Theta(\log n)$	$\sum_{i=1}^{n} f(i) = \Theta(1)$
If the terms grow very quickly, the total is dominated by the last and biggest term $f(n)$.	If half of the terms are roughly the same size, the total is roughly the number of terms times the last term.	A uniquely strange sum.	If the terms shrink quickly, the total is dominated by the first and biggest term $f(1)$, which is assumed here to be $\Theta(1)$.
$f(n)$ is an exponential	$f(n)$ is a polynomial or slowing decreasing	$f(n)$ is on the boundary	$f(n)$ is quickly decreasing
$f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$, $a > 0$, $b > 1$, $d, e \in (-\infty, \infty)$	$f(n) = \Theta(n^d \cdot \log^e n)$ $d > -1$, $e \in (-\infty, \infty)$	$f(n) = \Theta(\frac{1}{n})$	$f(n) = \Theta(\frac{\log^e n}{n^d})$ $d > 1$, $e \in (-\infty, \infty)$ $f(n) = \Theta(\frac{n^d \cdot \log^e n}{b^{an}})$ $a > 0$, $b > 1$, $d, e \in (-\infty, \infty)$
$f(n) \geq 2^{\Omega(n)}$	$f(n) = n^{\Theta(1)-1}$		$f(n) \leq n^{-1-\Omega(1)}$

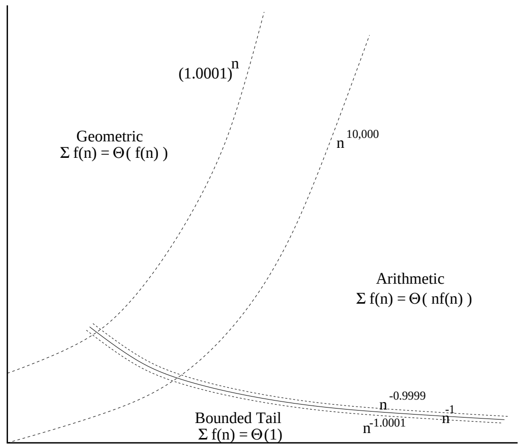


Figure 1.4: Boundaries between geometric, arithmetic, harmonic, and bounded tail.

Examples:

Geometric Increasing:

- $\sum_{i=1}^{n} \frac{2^i}{i^{100}} = \Theta(\frac{2^n}{n^{100}})$.
- $\sum_{i=1}^{n} 3^{i \log i} + 5^i + i^{100} = \Theta(3^{n \log n})$.
- $\sum_{i=1}^{n} 2^{i^2} + i^2 \log i = \Theta(2^{n^2})$.
- $\sum_{i=1}^{n} 2^{2^i - i^2} = \Theta(2^{2^n - n^2})$.

Arithmetic (Increasing):

- $\sum_{i=1}^{n} i^4 + 7i^3 + i^2 = \Theta(n^5)$.
- $\sum_{i=1}^{n} i^{4.3} \log^3 i + i^3 \log^9 i = \Theta(n^{5.3} \log^3 n)$.

Arithmetic (Decreasing):

- $\sum_{i=1}^{n} \frac{1}{\sqrt{i}} = \Theta(n \times f(n)) = \Theta(\sqrt{n})$.
- $\sum_{i=2}^{n} \frac{1}{\log i} = \Theta(\frac{n}{\log n})$.
- $\sum_{i=1}^{n} \frac{\log^3 i}{i^{0.6}} = \Theta(n^{0.4} \log^3 n)$.

Bounded Tail:

- $\sum_{i=1}^{n} \frac{1}{i^2} = \Theta(1)$.

- $\sum_{i=1}^{n} \frac{\log^3 i}{i^{1.6}+3i} = \Theta(1)$.
- $\sum_{i=1}^{n} \frac{i^{100}}{2^i} = \Theta(1)$.
- $\sum_{i=1}^{n} \frac{1}{2^{2^i}} = \Theta(1)$.

1.5.2 The Classic Techniques

We now present a few of the classic techniques for computing sums.

Geometric: A sum is said to be *geometric* if $f(i) = b^i$ for some constant $b \neq 1$. For example, $\sum_{i=0}^{n} 2^i = 1+2+4+8+\ldots+2^n = 2\cdot 2^n - 1$. (See Figure 1.5:a.) The key thing here is that each term is b times more than the previous term.

The Evaluation of a Geometric Sum: It can be summed as follows.

$$\begin{aligned}
S &= \sum_{i=0}^{n} b^i \\
&= 1 + \ b \ + \ b^2 \ + \ \ldots \ b^n \\
b\cdot S &= b \ + \ b^2 \ + \ b^3 \ + \ \ldots \ b^{n+1} \\
&\text{Subtract the above two lines} \\
(1-b)\cdot S &= 1 - b^{n+1} \\
S &= \frac{1-b^{n+1}}{1-b} \text{ or } \frac{b^{n+1}-1}{b-1} \\
&= \frac{f(0)-f(n+1)}{1-b} \text{ or } \frac{f(n+1)-f(0)}{b-1} \\
&= \Theta(\max(f(0), f(n)))
\end{aligned}$$

The sum is within a constant of the maximum term.

Ratio Between Terms: One test to see if a function $f(i)$ grows geometrically is whether for every sufficiently large i, the ratio $\frac{f(i+1)}{f(i)}$ between consecutive terms is at least $1+\epsilon$ for some fixed $\epsilon > 0$. For example, for $f(i) = \frac{2^i}{i}$, the ratio between consecutive terms is $\frac{f(i+1)}{f(i)} = \frac{2^{i+1}}{i+1}\cdot\frac{i}{2^i} = 2\cdot\frac{i}{i+1} = 2\cdot\frac{1}{1+\frac{1}{i}}$ which is at least 1.99 for sufficiently large i. On the other hand, the arithmetic function $f(i) = i$ has a ratio between the terms of $\frac{i+1}{i} = 1+\frac{1}{i}$. Though this is always bigger than one, it is not bounded away from one by some constant $\epsilon > 0$.

A proof that for any such function $\sum_{i=1}^{n} f(n) = \Theta(f(n))$ is as follows. If for every sufficiently large i, the ratio $\frac{f(i+1)}{f(i)}$ is at least $1+\epsilon$, then it follows using induction backward that $f(i) \leq f(n)\times\frac{1}{(1+\epsilon)^{n-i}}$. This gives that $\sum_{i=1}^{n} f(i) \leq \sum_{i=1}^{n} f(n)\times\frac{1}{(1+\epsilon)^{n-i}} = f(n)\times\sum_{j=1}^{n-1}\left(\frac{1}{1+\epsilon}\right)^j$. Recall that $\sum_{i=0}^{n-1} b^i = \frac{f(0)-f(n)}{1-b}$. This gives $\sum_{i=1}^{n} f(n) \leq f(n)\times\frac{1-(1/(1+\epsilon))^n}{1-(1/(1+\epsilon))} \approx \frac{1}{\epsilon}f(n) \leq \mathcal{O}(f(n))$. Conversely, it is obvious that if all the terms are positive, then $\sum_{i=1}^{n} f(n) \geq f(n)$.

Arithmetic: A sum is officially said to be *arithmetic* if $f(i) = ai+b$ for some constants a and b. The key thing is that each term is a fixed constant more than the previous term.

The Evaluation of the Classic Arithmetic Sum: The classic example is $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ which can be summed as follows. (See Figure 1.5:b.)

$$\begin{array}{rcccccccccccccc}
S &=& 1 &+& 2 &+& 3 &+& \ldots &+& n-2 &+& n-1 &+& n \\
S &=& n &+& n-1 &+& n-2 &+& \ldots &+& 3 &+& 2 &+& 1 \\
2S &=& n+1 &+& n+1 &+& n+1 &+& \ldots &+& n+1 &+& n+1 &+& n+1 \\
&=& \multicolumn{13}{l}{n\cdot(n+1)} \\
S &=& \multicolumn{13}{l}{\frac{1}{2}n\cdot(n+1)}
\end{array}$$

The sum is approximately n times the maximum term.

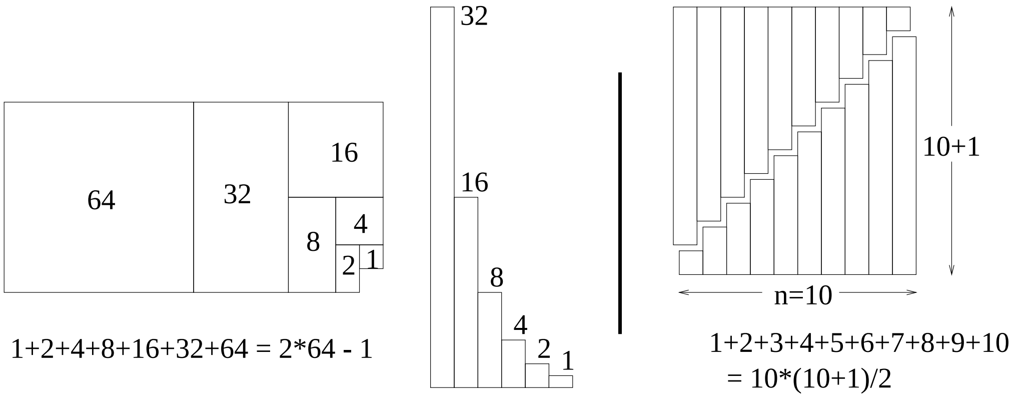


Figure 1.5: The two figures on the left illustrate geometric sums and the right illustrates arithmetic sums

Sum of a Polynomial: This approximation of having sum $\sum_{i=1}^{n} f(i)$ being within a constant of being n times the maximum term turns out to be true for $f(i) = i^d$ for any $d \geq 0$. (Later we will see that it is also true for $d \in (-1, 0]$.) For example, $\sum_{i=1}^{n} i^2$ is approximately $\frac{1}{3}n^3$. A quick intuitive proof of this is as follows. If $f(i)$ is non-decreasing then half of the terms are greater or equal to the middle term $f(\frac{n}{2})$ and all of the terms are smaller or equal to the biggest term $f(n)$. Pictorially this is shown in Figure 1.6. From these it follows that the sum $\sum_{i=1}^{n} f(i)$ is greater or equal to half the number of terms times the middle term and smaller or equal to the number of terms times the largest term, namely $\frac{n}{2} \times f(\frac{n}{2}) \leq \sum_{i=1}^{n} f(i) \leq n \times f(n)$. If these bounds match within a multiplicative constant, i.e. if $f(\frac{n}{2}) = \Theta(f(n))$, then $\sum_{i=1}^{n} f(i) = \Theta(n \times f(n))$. This is the case for $f(i) = i^d$ for $d \geq 0$, because $f(\frac{n}{2}) = \left(\frac{n}{2}\right)^d = \frac{1}{2^d} f(n)$. Later, we will see using integration that the multiplicative constant difference between $\sum_{i=1}^{n} f(i)$ and $n \times f(i)$ is closer to $\frac{1}{d+1}$ than it is to $\frac{1}{2^d}$, but this does prove that it is some constant.

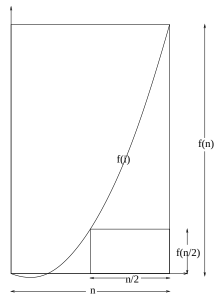


Figure 1.6: Bounding arithmetic sums

The Harmonic Sum: The harmonic sum is a famous sum that arises surprisingly often. $\sum_{i=1}^{n} \frac{1}{i}$ is within one of $\log_e n$.

Blocks: One way of approximating the harmonic sum is to break it into $\log_2 n$ blocks, where the total for each block is between $\frac{1}{2}$ and 1.

$$\sum_{i=1}^{n} \frac{1}{i} = \overbrace{\underbrace{\frac{1}{1}}_{\leq 1\cdot 1 = 1}}^{\geq 1\cdot\frac{1}{2}=\frac{1}{2}} + \overbrace{\underbrace{\frac{1}{2}+\frac{1}{3}}_{\leq 2\cdot\frac{1}{2}=1}}^{\geq 2\cdot\frac{1}{4}=\frac{1}{2}} + \overbrace{\underbrace{\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}}_{\leq 4\cdot\frac{1}{4}=1}}^{\geq 4\cdot\frac{1}{8}=\frac{1}{2}} + \overbrace{\underbrace{\frac{1}{8}+\ldots+\frac{1}{15}}_{\leq 8\cdot\frac{1}{8}=1}}^{\geq 8\cdot\frac{1}{16}=\frac{1}{2}} + \ldots$$

From this, it follows that $\frac{1}{2} \times \log_2 n \leq \sum_{i=1}^{n} \frac{1}{i} \leq 1 \times \log_2 n$.

Different Starting Point: Sometimes it is useful to evaluate the harmonic sum starting at an arbitrary m. $\sum_{i=m}^{n} \frac{1}{i} = \sum_{i=1}^{n} \frac{1}{i} - \sum_{i=1}^{m-1} \frac{1}{i} \log_e n - \log_e(m-1) + \Theta(1) = \log_e \frac{n}{m} + \Theta(1)$.

Integrating: Sums of functions can be approximated by integrating. In fact, both $\sum$ (the Greek letter sigma) and $\int$ are "S" for sum. (See Figure 1.7.) Conversely, every thing said in this chapter about using Θ notation for approximating sums can be said about approximating integrations.

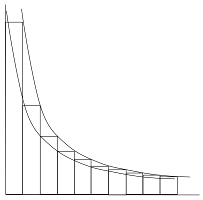


Figure 1.7: The area under the top curve is slightly more than the discrete sum. The area under the bottom curve is slightly less than this sum. Generally, both are good approximations. These areas under the curves are given by integrating.

Bounds:

- Generally, it is safe to say that $\sum_{i=1}^{n} f(i) = \Theta(\int_{x=1}^{n} f(x)\ \delta x)$.
- If $f(i)$ is a monotonically increasing function, then $\int_{x=m-1}^{n} f(x)\ \delta x \leq \sum_{i=m}^{n} f(i) \leq \int_{x=m}^{n} f(x)\ \delta x + f(n) \leq \int_{x=m}^{n+1} f(x)\ \delta x$.
- If $f(i)$ is a monotonically decreasing function, then $\int_{x=m}^{n+1} f(x)\ \delta x \leq \sum_{i=m}^{n} f(i) \leq \int_{x=m-1}^{n} f(x)\ \delta x$.

Examples:

Geometric:

Sum	*Integral*
$\sum_{i=0}^{n} 2^i = 2 \cdot 2^n - 1$	$\int_{x=0}^{n} 2^x\ \delta x = \frac{1}{\ln 2} 2^n$
$\sum_{i=0}^{n} b^i = \frac{1}{b-1}(b^{n+1} - 1)$	$\int_{x=0}^{n} b^x\ \delta x = \frac{1}{\ln b} b^n$

Arithmetic:

$\sum_{i=1}^{n} i = \frac{1}{2}n^2 + \frac{1}{2}n$	$\int_{x=0}^{n} x\ \delta x = \frac{1}{2}n^2$
$\sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$	$\int_{x=0}^{n} x^2\ \delta x = \frac{1}{3}n^3$
$\sum_{i=1}^{n} i^d = \frac{1}{d+1}n^{d+1} + \Theta(n^d)$	$\int_{x=0}^{n} x^d\ \delta x = \frac{1}{d+1}n^{d+1}$

Harmonic:

$\sum_{i=1}^{n} \frac{1}{i} \approx \int_{x=1}^{n+1} \frac{1}{x}\ \delta x = [\log_e x]_{x=1}^{n+1} = [\log_e n + 1] - [\log_e 1] = \log_e n + \Theta(1)$.

Close to Harmonic: Let ϵ be some small positive constant (say 0.0001).

$\sum_{i=1}^{n} \frac{1}{i^{(1+\epsilon)}} \approx \int_{x=1}^{n+1} \frac{1}{x^{(1+\epsilon)}}\ \delta x = \int_{x=1}^{n+1} x^{-1-\epsilon}\ \delta x = \left[\frac{1}{-\epsilon} x^{-\epsilon}\right]_{x=1}^{n+1} = \left[-\frac{1}{\epsilon}(n+1)^{-\epsilon}\right] - \left[-\frac{1}{\epsilon}1^{-\epsilon}\right] = \Theta(\frac{1}{\epsilon}) = \Theta(1)$.

$\sum_{i=1}^{n} \frac{1}{i^{(1-\epsilon)}} \approx \int_{x=1}^{n+1} \frac{1}{x^{(1-\epsilon)}}\ \delta x = \int_{x=1}^{n+1} i^{-1+\epsilon}\ \delta x = \left[\frac{1}{\epsilon} x^{\epsilon}\right]_{x=1}^{n+1} = \left[\frac{1}{\epsilon}(n+1)^{\epsilon}\right] - \left[\frac{1}{\epsilon}1^{\epsilon}\right] = \Theta(\frac{1}{\epsilon}n^{\epsilon}) = \Theta(n^{\epsilon})$.

Difficulty: The problem with this method of evaluating a sum is that integrating can also be difficult. E.g., $\sum_{i=1}^{n} 3^i i^2 \log^3 i \approx \int_{x=0}^{n} 3^x x^2 \log^3 x\ \delta x$. However, integrals can be approximated using the same techniques used here for sums.

1.5.3 The Ranges of The Adding Made Easy Approximations

The Four Types of Approximations:

Geometric Increasing:

If $f(n) = \Theta(b^n \cdot n^d \cdot \log^e n)$ for $b > 1$
or more generally if $f(n) \geq 2^{\Omega(n)}$,
then $\sum_{i=1}^{n} f(i) = \Theta(f(n))$.

The intuition is that if the terms $f(i)$ grow sufficiently quickly, then the sum will be dominated by the largest term. This certainly is true for a silly example like $1+2+3+4+5+1,000,000,000 \approx 1,000,000,000$. We have seen that it is also true for the classic example $\sum_{i=1}^{n} 2^i$ and in fact for $\sum_{i=1}^{n} b^i$ for all constants $b > 1$. Recall that $b^i = 2^{(\log_2 b)i} = 2^{ci}$ for some $c > 0$. Hence, an equivalent way of expressing this is that it is true for $\sum_{i=1}^{n} 2^{ci}$ for all constants $c > 0$. At first, one might think that this set of functions is defined to be the class of *exponential* functions, $2^{\Theta(n)}$. See Section 1.4.1. However, strange functions like $f(n) = 2^{[\frac{1}{2}\cos(\Pi \log_2 n)+1.5]\cdot n}$ are also exponential. Because the cosine introduces a funny wave to this function, it turns out that it is not true that $\sum_{i=1}^{n} f(i) = \Theta(f(n))$. See below. However, this geometric approximation is true for all exponential functions that are simple analytical functions. Recall, that such functions are those expressed with n, real constants, plus, minus, times, divide, exponentiation, and logarithms. Also, if is true for these functions, then it should also be true for any such function that grows even faster than an exponential. We write this requirement as $f(n) \geq 2^{\Omega(n)}$.

Arithmetic:

If $f(n) = \Theta(n^d \cdot \log^e n)$ for $d > -1$
or more generally if $f(n) = n^{\Theta(1)-1}$,
then $\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$.

The intuition is that if most of the terms $f(i)$ have roughly the same value, then the sum is roughly the number of terms, n, times this value. This certainly is true for a silly example like $1,001 + 1,002 + 1,003 + 1,004 + 1,005 + 1,006 \approx 6 \cdot 1,000$. We have seen that it is also true for the classic example $\sum_{i=1}^{n} i$ and in fact for $\sum_{i=1}^{n} i^c$ for all constants $c > 0$. It is not, however, true for all *polynomial* functions, $n^{\Theta(1)}$, because it is not true for the strange function $f(n) = (1 + sin(n)) \cdot n^3 + n$, due to funny wave introduced by the sine. However, this arithmetic approximation is true for all polynomials, $n^{\Theta(1)}$, that are simple analytical functions.

It is true for other functions as well. The function $f(n) = 1$ is not a polynomial, but clearly $\sum_{i=1}^{n} 1 = \Theta(n \times 1)$. It is not true for $\sum_{i=1}^{n} \frac{1}{i^2}$, because $\Theta(n \cdot f(n)) = \Theta(n \cdot \frac{1}{n^2}) = \Theta(\frac{1}{n})$, which is not what $\sum_{i=1}^{n} \frac{1}{i^2}$ sums up to given that the sum is at least the first term $f(1) = 1$. Neither is it true for the harmonic sum $\sum_{i=1}^{n} \frac{1}{i}$, because $\Theta(n \cdot f(n)) = \Theta(n \cdot \frac{1}{n}) = \Theta(1)$, while we have see that $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$. Exploring for the smallest function for which this approximation is true, we see that it is true for a function shrinking just slightly slower than the harmonic sum, namely $\sum_{i=1}^{n} \frac{1}{n^{0.999}} \approx 1,000n^{0.001} = \Theta(n^{0.001}) = \Theta(n^{1-0.999}) = \Theta(n \cdot f(n))$.

In conclusion, the arithmetic approximation $\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$, is true for $f(n) = n^d$ for all $d > -1$. Expressed another way, it is true when ever $f(n) = n^{c-1}$ for any $c > 0$ or even more generally for any simple analytical function $f(n)$ for which $f(n) = n^{\Theta(1)-1}$. Another way to think about this condition is that the sum totals $\Theta(n \cdot f(n))$ as long as this total is a polynomial $n^{\Theta(1)}$.

Harmonic:

If $f(n) = \Theta(\frac{1}{n})$,
then $\sum_{i=1}^{n} f(i) = \Theta(\log n)$

These are the only functions in this class.

Bounded Tail:

If $f(n) = \Theta(n^d \cdot \log^e n)$ for $d < -1$
or if $f(n) = \Theta(b^n \cdot n^d \cdot \log^e n)$ for $b < 1$
or more generally if $f(n) \leq n^{-1-\Omega(1)}$,
then $\sum_{i=1}^{n} f(i) = \Theta(1)$.

The intuition is that if the terms $f(i)$ decay (decreases towards zero) sufficiently quickly, then the sum will be a constant. The classic example is $\sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots = 1$. We have seen that it is not true for the harmonic sum $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$. Exploring for the largest function for which this bounded tail approximation is true, we see that it is true for a function shrinking just slightly faster than the harmonic sum, namely $\sum_{i=1}^{n} \frac{1}{n^{1.001}} \approx 1{,}000 = \Theta(1)$. This makes us want to classify functions $f(n)$ that are n^d for all $d < -1$. Expressed another way, $f(n) = n^{-1-c}$ for any $c > 0$ or even more generally $f(n) = n^{-1-\Theta(1)}$.

If is true for these functions, then it should also be true for any such function that shrinks even faster. We write this requirement as any simple analytical function $f(n)$ for which $f(n) \leq n^{-1-\Omega(1)}$.

Sufficiently Large n: Recall that Θ notation considers the asymptotic behavior of a function for sufficiently large values of n. If $f(n)$ is such that the desired growth behavior does not start until some large $n \geq n_o$, similarly, the approximation for the sum may only be accurate after this point. For example, consider $f(n) = \frac{n^{100}}{2^n}$. This function increases until about $n = 1{,}024$ and then rapidly decreases exponentially. Being a decreasing exponential, $f(n) = \frac{1}{2^{\Theta(n)}}$, the adding made easy rules say that the sum $\sum_{i=1}^{n} f(i)$ should be bounded tail, totaling to ' $\Theta(1)$. This may be hard to believe at first, but it can best be understood by breaking it into two sums, $\sum_{i=1}^{n} f(i) = \sum_{i=1}^{1,024} f(i) + \sum_{i=1,025}^{n} f(i)$. The first sum may total to some big amount, but what ever it is, it is a constant independent of n. The remaining sum will be some reasonable constant even if summed to $n = \infty$.

Functions That Lie Between These Cases:

Between Arithmetic and Geometric: Some functions, $f(i)$, grow too slowly to be "geometric" and too quickly to be "arithmetic". For these, the behavior of the sum $\sum_{i=1}^{n} f(i)$ is between that for the geometric and the arithmetic sums. Specifically, $\Omega(f(n)) \leq \sum_{i=1}^{n} f(i) \leq \mathcal{O}(n \times f(n))$. Two examples are $f(n) = n^{\log n}$ and $f(n) = 2^{\sqrt{n}}$. Their sums are $\sum_{i=1}^{n} i^{\log i} = \Theta(\frac{n}{\log n} n^{\log n}) = \Theta(\frac{n}{\log n} f(n))$ and $\sum_{i=1}^{n} 2^{\sqrt{n}} = \Theta(\sqrt{n} f(n))$.

Between Arithmetic, Harmonic, and Bounded Tail: Let us look closer at the boundary between a function being arithmetic, harmonic, and having a bounded tail.

Taking the Limit: Consider the sum $\sum_{i=1}^{n} \frac{1}{i^{(1-\epsilon)}}$. When $\epsilon > 0$, the sum is arithmetic giving a total of $\Theta(nf(n)) = \Theta(n^{\epsilon})$. When $\epsilon = 0$, the sum is harmonic giving a total of $\Theta(\log_e n)$. When $\epsilon < 0$, the sum has a bounded tail giving a total of $\Theta(1)$. To see how these answers blend together, consider the limit $\lim_{\epsilon \to 0} \sum_{i=1}^{n} \frac{1}{i^{(1-\epsilon)}} \approx \lim_{\epsilon \to 0} \frac{n^{\epsilon}-1}{\epsilon} = \lim_{\epsilon \to 0} \frac{e^{\epsilon \log_e n}-1}{\epsilon}$. Recall that Lhobitell's rule can be applied when both numerator and denominator have a limit of the zero. Taking the derivative with respect to ϵ of both numerator and the denominator gives $\lim_{\epsilon \to 0} \frac{(\log_e n) e^{\epsilon \log_e n}}{1} = \log_e n$. This corresponds to the harmonic sum.

Other Functions: There are other functions that lay between these cases. For example, the function $f(n) = \frac{\log n}{n}$ is between being arithmetic and harmonic. The function $f(n) = \frac{1}{n \log n}$ is between being harmonic and having a bounded tail. Their sums are $\sum_{i=1}^{n} \frac{\log n}{n} = \Theta(\log^2 n)$ and $\sum_{i=1}^{n} \frac{1}{\log(n) n} = \Theta(\log \log n)$.

1.5.4 Harder Examples

Above we considered sums of the form $\sum_{i=1}^{n} f(i)$ for simple analytical functions $f(n)$. We will now consider sums that do not fit into this basic form.

Other Ranges: The sums considered so far have had n terms indexed by the variable i running from either $i = 0$ or $i = 1$ to $i = n$. Other ranges, however, might arise.

Different Variable Names: If $\sum_{i=1}^{n} i^2 = \Theta(n^3)$, it should not be too hard to know that $\sum_{j=1}^{m} j^2 = \Theta(m^3)$.

Different Starting Indexes: In terms of a theta approximation of the sum, it does not matter if it starts from $i = 0$ or $i = 1$. But there may be other starting points as well. As a general rule, $\sum_{i=m}^{n} f(i) = \sum_{i=1}^{n} f(i) - \sum_{i=1}^{m-1} f(i)$.

- $\sum_{i=m}^{n} i^2 = \Theta(n^3) - \Theta(m^3)$, which is $\Theta(n^3)$ unless m is very close to n.

- $\sum_{i=m}^{n} \frac{1}{i} = \Theta(\log n) - \Theta(\log m) = \Theta(\log \frac{n}{m})$.

Different Ending Indexes: Instead having n terms, the number of terms may be a function of n.

- $\sum_{i=1}^{5n^2+n} i^3 \log i$: To solve this let N denote the number of terms. Adding made easy gives that $\sum_{i=1}^{N} i^3 \log i = \Theta(Nf(N)) = \Theta(N^4 \log N)$. Substituting back in $N = 5n^2 + n$ gives $\sum_{i=1}^{5n^2+n} i^3 \log i = \Theta((5n^2+n)^4 \log(5n^2+n)) = \Theta(n^8 \log n)$.
- $\sum_{i=1}^{\log n} 3^i i^2 = \Theta(f(N)) = \Theta(3^N N^2) = \Theta(3^{\log n} (\log n)^2) = \Theta(n^{log 3} \log^2 n)$.

Terms Depend on n: It is quite usual to have a sum where both the number of terms and the terms themselves depend on the same variable n, namely $\sum_{i=1}^{n} i \cdot n$. This can arise, for example, when computing the running time of two nested loops, where there are n iterations of the outer loop, the i^{th} of which requires $i \cdot n$ time.

The Variable n is a Constant: Though n is a variable, depended perhaps on the input size, for the purpose of computing the sum it is a constant. The reason is that its value does not change as we iterate from one term to the next. Only i changes. Hence, when determining whether the sum is arithmetic or geometric, the key is how the terms change with respect to i, not how it changes with respect to n. The only difference between this n within the term and say the constant 3, when approximating the sum, is that the n cannot be absorbed into the Theta.

Factoring Out: Often the dependence on n within the terms can be factored out. If so, this is the easiest way to handle the sums.

- $\sum_{i=1}^{n} i \cdot n \cdot m = nm \cdot \sum_{i=1}^{n} i = nm \cdot \Theta(n) = \Theta(n^2 m)$.
- $\sum_{i=1}^{n} \frac{n}{i} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = n \cdot \Theta(\log n) = \Theta(n \log n)$.
- $\sum_{i=1}^{\log_2 n} 2^{(\log_2 n - i)} \cdot i^2 = 2^{\log_2 n} \cdot \sum_{i=1}^{\log_2 n} 2^{-i} \cdot i^2 = n \cdot \Theta(1) = \Theta(n)$.

Adding Made Easy: The same general adding made easy principles can still be used when the indexing is not $i = 1..n$ or the when terms depend on n.

Geometric Increasing, $\Theta(f(n))$: If the terms are increasing geometrically, then the total is approximately the last term.

- $\sum_{i=1}^{\log_2 n} n^i$. Note this is a polynomial in n but is exponential in i. Hence, it is geometric increasing. The biggest term is $n^{\log n}$ and the total is $\Theta(n^{\log n})$.

Bounded Tail, $\Theta(f(1))$: If the terms are decreasing geometrically or decreasing as fast as $\frac{1}{i^2}$, then the total is still Theta of the biggest term, but this is now the first term. The difference with what we were doing before, however, is that now this first term might not be $\Theta(1)$.

- $\sum_{i=\frac{n}{2}}^{n} \frac{1}{i^2} = \Theta\left(\frac{1}{(\frac{n}{2})^2}\right) = \Theta\left(\frac{1}{n^2}\right)$.
- $\sum_{i=1}^{\log_2 n} 2^{(\log_2 n - i)} \cdot i^2$. If nervous about this, the first thing to check is what the first and last terms are (and how many terms there are). The first term is $f(1) = 2^{(\log n - 1)} \cdot 1^2 = \Theta(n)$. The last term is $f(\log n) = 2^{(\log n - \log n)} \cdot (\log n)^2 = \Theta(\log^2 n)$, which is significantly smaller than the first. This is an extra clue that the terms decrease geometrically in i. The total is then $\Theta(f(1)) = \Theta(n)$.

Arithmetic, Θ(number of terms · biggest term): If the sum is arithmetic then most of the terms contribute to the sum and the total is approximately the number of terms times the biggest term. The new thing to note here is that the number of terms is not necessarily n.

- $\sum_{i=m}^{n} i \cdot n \cdot m$. The number of terms is $n - m$. The biggest is $n^2 m$. The total is $\Theta((n-m)n^2 m)$.
- $\sum_{i=1}^{\log n} n \cdot i^2$. The number of terms is $\log n$. The biggest is $n \log^2 n$. The total is $\Theta(n \log^3 n)$.

Harmonic: It is hard to use the adding made easy method here. It is best to use the techniques above, namely $\sum_{i=m}^{n} \frac{1}{i} = \Theta(\log n) - \Theta(\log m) = \Theta(\log \frac{n}{m})$ and $\sum_{i=1}^{n} \frac{n}{i} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = n \cdot \Theta(\log n) = \Theta(n \log n)$.

Functions that are not "Simple Analytical": The Adding Made Easy Approximations require that the function $f(n)$ is what we called "simple analytical", i.e. that it is expressed with real constants, n, and the operations plus, minus, times, divide, exponentiation, and logarithms. Oscillating functions like sine and cosine are not allowed. Similarly, complex numbers are not allowed, because of the relations $e^{\mathbf{i}x} = \cos(x) + \mathbf{i}\sin(x)$ or $\cos(x) = \frac{1}{2}(e^{\mathbf{i}x} + e^{-\mathbf{i}x})$, where $\mathbf{i} = \sqrt{-1}$.

The functions $f_{stair}(n)$ and $f_{cosine}(n)$ given in the figure below are both counter examples to the statement "If $f(n) \geq 2^{\Omega(n)}$, then $\sum_{i=1}^{n} f(i) = \Theta(f(n))$."

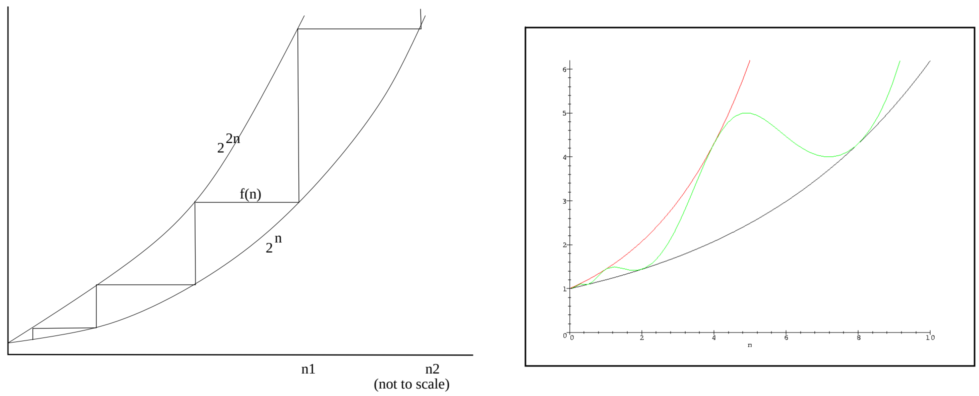


Figure 1.8: The left function $f_{stair}(n)$ is a stair case with its lower corners squeezed between the functions 2^{1n} and 2^{2n}. The right function is a similar function however it is defined with the function $f_{cosine}(n) = 2^{[\frac{1}{2}\cos(\Pi \log_2 n)+1.5]\cdot n}$.

$f_{stair}(n), f_{cosine}(n) = 2^{\Theta(n)}$: The functions $f_{stair}(n)$ and $f_{cosine}(n)$ are both $2^{\Theta(n)}$. One might initially think that they are not because they oscillate up and down. Recall, however, that the formal definition of $f(n) = 2^{\Theta(n)}$ is that $\exists c_1, c_2, n_0,\ \forall n \geq n_0,\ 2^{c_1 n} \leq f(n) \leq 2^{c_2 n}$. Because $f_{stair}(n)$ and $f_{cosine}(n)$ are bounded between 2^{1n} and 2^{2n}, it is true that they are $2^{\Theta(n)}$.

$\sum_{i=1}^{n} f(i) \neq \Theta(f(n))$: Let n_1 and n_2 have the values indicated in the figure. Note $2^{2n_1} = 2^{n_2}$ and hence $n_1 = \frac{1}{2}n_2$. From the picture it is clear that for both $f_{stair}(n)$ and $f_{cosine}(n)$, $\sum_{i=1}^{n_2} f(i) \geq (n_2 - n_1)f(n_2) = \frac{1}{2}n_2 f(n_2) \neq \mathcal{O}(f(n_2))$.

1.6 Recurrence Relations

"Very well, sire," the wise man said at last, "I asked only this: Tomorrow, for the first square of your chessboard, give me one grain of rice; the next day, for the second square, two grains of rice; the next day after that, four grains of rice; then, the following day, eight grains for the next square of your chessboard. Thus for each square give me twice the number of grains of the square before it, and so on for every square of the chessboard."

"Now the King wondered, as anyone would, just how many grains of rice this would be."

... thirty-two days later ...

"This must stop," said the King to the wise man. "There is not enough rice in all of India to reward you."

"No, indeed sire," said the wise man. "There is not enough rice in all the world."

The King's Chessboard by David Birch

The number of grains of rice on square n in above story is given by $T(1) = 1$ and $T(n) = 2T(n-1)$. This is called a recurrence relation. Such relations arise often in the study of computer algorithms. The most common place that they arise is in computing the running time of a recursive program. A recursive program is a routine or algorithm that calls itself on a smaller input instance. See Chapter 11. Similarly, a recurrence relation is an function that is defined in terms of itself on a smaller input instance.

An Equation Involving an Unknown: A recurrence relation is an equation (or a set of equations) involving an unknown variable. Solving it involves finding a "value" for the variable that satisfies the equations. It is similar to algebra.

Algebra with Reals as Values: $x^2 = x + 2$ is an algebraic equation where the variable x is assumed to take on a real value. One way to solve it is to guess a solution and to check to see if it works. Here $x = 2$ works, i.e., $(2)^2 = 4 = (2) + 2$. However, $x = -1$ also works, $(-1)^2 = 1 = (-1) + 2$. Making the further requirement that $x \geq 0$ narrows the solution set to only $x = 2$.

Differential Equations with Functions as Values: $\frac{\delta f(x)}{\delta x} = f(x)$ is a differential equation where the variable f is assumed to take on a function from reals to reals. One way to solve it is to guess a solution and to check to see if it works. Here $f(x) = e^x$ works, i.e., $\frac{\delta e^x}{\delta x} = e^x$. However, $f(x) = c \cdot e^x$ also works for each value of c. Making the further requirement that $f(0) = 5$ narrows the solution set to only $f(x) = 5 \cdot e^x$.

Recurrence Relations with Discrete Functions as Values: $T(n) = 2 \times T(n-1)$ is a recurrence relation where the variable T is assumed to take on a function from integers to reals. One way to solve it is to guess a solution and to check to see if it works. Here $T(n) = 2^n$ works, i.e., $2^n = 2 \times 2^{n-1}$. However, $T(n) = c \cdot 2^n$ also works for each value of c. Making the further requirement that $T(0) = 5$ narrows the solution set to only $T(n) = 5 \cdot 2^n$.

1.6.1 Relating Recurrence Relations to the Timing of Recursive Programs

One of the most common places for recurrence relations to arise is when analyzing the time complexity of recursive algorithms.

Basic Code: Suppose a recursive program, when given an instances of size n, recurses a times on subinstances of size $\frac{n}{b}$.

algorithm $Eg(I_n)$

$\langle$***pre−cond***$\rangle$**:** I_n is an instance of size n.

$\langle$***post−cond***$\rangle$**:** Prints $T(n)$ "Hi"s.

```
begin
    n = |I_n|
    if( n ≤ 1) then
        put "Hi"
    else
        loop i = 1..f(n)
            put "Hi"
        end loop
        loop i = 1..a
            I_{n/b} = an input of size n/b
            Eg(I_{n/b})
        end loop
    end if
end algorithm
```

Stack Frames: A single execution of the routine on a single instance is referred to as a stack frame. See Section 11.1.5. The top stack frame recurses b times creating b more stack frames. Each of these recurse b times for a total of $b \cdot b$ stack at this third *level*. This continues until a base case is reached.

Time for Base Cases: Recursive programs must stop recursing when the input becomes sufficiently small. In this example, only one "Hi" is printed when the input has size zero or one. In general, we will assume that recursive programs spend $\Theta(1)$ time for instances of size $\Theta(1)$. We express this as $T(1) = 1$ or more generally as $T(\Theta(1)) = \Theta(1)$.

Running Time: Let $T(n)$ denote the total computation time for instances of size n. This time can be divided a number of different ways.

Top Level plus Recursion:

Top Level: Before recursing, the top level stack frame prints "Hi" $f(n)$ times. In general, we use $f(n)$ to denote the computation time required by a stack frame on an instance of size n. This time includes the time needed to generate the subinstances and recombining their solutions into the solution for its instance, but excludes the time needed to recurse.

Recursion: The top stack frame, recurses a times on subinstances of size $\frac{n}{b}$. If $T(n)$ is the total computation time for instances of size n, then it follows that $T\left(\frac{n}{b}\right)$ is the total computation time for instances of size $\frac{n}{b}$. Repeating this a times will take time $a \cdot T\left(\frac{n}{b}\right)$.

Total: We have already said that the total computation time is $T(n)$. Now, however, we have also concluded that the total time is $f(n)$ for the top level and $T\left(\frac{n}{b}\right)$ for each of the recursive calls, for a total of $a \cdot T\left(\frac{n}{b}\right) + f(n)$.

The Recurrence Relation: It follows that $T(n)$ satisfies the recursive relation

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

The goal of this section is to determine which function $T(n)$ satisfies this relation.

Sum of Work at each Level of Recursion: Another way of computing the total computation time $T(n)$ is to sum up the time spent by each stack frame. These stack frames form a tree based on who calls who. The stack frames at the same *level* of this tree often have input instances that are of roughly the same size, and hence have roughly equivalent running times. Multiplying this time by the number of stack frames at the level gives the total time spent at this level. Adding up this time for all the levels is another way of computing the total time $T(n)$. This will be our primary way of evaluating recurrence relations.

Instances of Size $n-b$: Suppose instead of recursing a times on instances of size $\frac{n}{b}$, the routine recurses a times on instances of size $n-b$, for some constant b. The same discussion as above will apply, but the related recurrence relation will be

$$T_2(n) = a \cdot T_2(n-b) + f(n)$$

1.6.2 Summary

This subsection summarizes what will be discussed a length within in this section.

The Basic Form: Most recurrence relations that we will consider will have the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, where a and b are some real positive constants and $f(n)$ is a function. The second most common form is $T(n) = a \cdot T(n-b) + f(n)$. This section will be focusing on the first of these forms. Section 1.6.4, we will apply the same idea to the second of these forms and to other recurrence relations.

Base Cases: Though out we will assume that T with a constant input has a constant output. We write this as $T(\Theta(1)) = \Theta(1)$.

Examples: The following examples, include some of the classic recurrence relations as well as some other examples designed to demonstrate the various patterns that the results fall into.

Size of Subinstance of Size $\frac{n}{2}$:

Dominated By	*One Subinstance*	*Two Subinstances*
Top Level	$T(n) = T(n/2) + n = \Theta(n)$	$T(n) = 2 * T(n/2) + n^2 = \Theta(n^2)$
Levels Arithmetic	$T(n) = T(n/2) + 1 = \Theta(\log n)$	$T(n) = 2 * T(n/2) + n = \Theta(n \log n)$
Base Cases	$T(n) = T(n/2) + 0 = \Theta(1)$	$T(n) = 2 * T(n/2) + 1 = \Theta(n)$

Different Sized of Subinstances:

Dominated By	*Three Subinstances*
Top Level	$T(n) = T(\frac{1}{7}n) + T(\frac{2}{7}n) + T(\frac{3}{7}n) + n = \Theta(n)$
Levels Arithmetic	$T(n) = T(\frac{1}{7}n) + T(\frac{2}{7}n) + T(\frac{4}{7}n) + n = \Theta(n \log n)$
Base Cases	$T(n) = T(\frac{1}{7}n) + T(\frac{3}{7}n) + T(\frac{4}{7}n) + n = \Theta(n^?)$

Size of Subinstance of Size $n-1$:

Dominated By	*One Subinstance*	*Two Subinstances*
Top Level	$T(n) = T(n-1) + 2^n = \Theta(2^n)$	$T(n) = 2 * T(n-1) + 3^n = \Theta(3^n)$
Levels Arithmetic	$T(n) = T(n-1) + n = \Theta(n^2)$	$T(n) = 2 * T(n-1) + 2^n = \Theta(n2^n)$
Base Cases	$T(n) = T(n-1) + 0 = \Theta(1)$	$T(n) = 2 * T(n-1) + n = \Theta(2^n)$

A Growing Number of Subinstances of a Shrinking Size: Consider recurrence relations of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ with $T(\Theta(1)) = \Theta(1)$. Each instances having a subinstances means that the number of subinstances grows exponentially by a factor of b. On the other hand, the size of the subinstances shrink exponentially. The amount of work that the instance must do is the function f of this instance size. Whether the growing or the shrinking dominates this process depend on the relationship between a, b, and $f(n)$.

Four Different Solutions: All recurrence relations that we will consider, have one of the following four different types of solutions. These correspond to the four types of solutions for sums, see Section 1.5.

Dominated by Top: The *top level* of the recursion requires $f(n)$ work. If this is sufficiently big then this time will dominate the total and the answer will be $T(n) = \Theta(f(n))$.

Dominated by Base Cases: We will see that the number of *base cases* is $n^{\frac{\log a}{\log b}}$ when $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$. Each of these requires $T(\Theta(1)) = \Theta(1)$ time. If this is sufficiently big then this time will dominate the total and the answer will be $T(n) = \Theta\left(n^{\frac{\log a}{\log b}}\right)$. For example, when $T(n) = 1 \cdot T\left(\frac{n}{2}\right) + f(n)$, $T(n) = \Theta\left(n^{\frac{\log 1}{\log 2}}\right) = \Theta\left(n^0\right) = \Theta(1)$ and when $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + f(n)$, $T(n) = \Theta\left(n^{\frac{\log 2}{\log 2}}\right) = \Theta\left(n^1\right) = \Theta(n)$.

Levels Arithmetic: If the amount of work at the different levels of recursion is sufficiently close to each other, then the total is the number of levels times this amount of work. In this case, there are $\Theta(\log n)$ levels, giving that $T(n) = \Theta(\log n) \times \Theta(f(n))$.

Levels Harmonic: A strange example, included only for completion, is when the work at each of the levels forms a harmonic sum. In this case, $T(n) = \Theta(f(n) \log n \log\log n) = \Theta(n^{\frac{\log a}{\log b}} \log\log n)$.

If $T(n) = a \cdot T(n-b) + f(n)$, then the types of solutions are the same, except that the number of base cases will be $a^{\frac{n}{b}}$ and the number of levels is $\frac{n}{b}$ giving the solutions $T(n) = \Theta(f(n))$, $T(n) = \Theta(a^{\frac{n}{b}})$, and $T(n) = \Theta(n \cdot f(n))$.

Rules for Functions with Basic Form $T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^c \cdot \log^d n)$: To simplify things, let us assume that $f(n)$ has the basic form $\Theta(n^c \cdot \log^d n)$, where c and d are some real positive constants. This recurrence relation can be quickly approximated using the following simple rules.

algorithm *BasicRecurrenceRelation*(a, b, c, d)

$\langle$***pre−cond***$\rangle$**:** We are given a recurrence relation with the basic form $T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^c \cdot \log^d n)$. We assume $a \geq 1$ and $b > 1$.

⟨post−cond⟩: Computes the approximation $\Theta(T(n))$.

```
begin
    if( c > log a / log b ) then
        The recurrence is dominated by the top level and T(n) = Θ(f(n)) = Θ(n^c · log^d n).
    else if( c = log a / log b ) then
        if( d > −1 ) then
            Then the levels are arithmetic and T(n) = Θ(f(n) log n) = Θ(n^c · log^{d+1} n).
        else if( d = −1 ) then
            Then the levels are harmonic and T(n) = Θ(n^c · log log n).
        else if( d < −1 ) then
            The recurrence is dominated by the base cases and T(n) = Θ(n^{log a / log b}).
        end if
    else if( c < log a / log b ) then
        The recurrence is dominated by the base cases and T(n) = Θ(n^{log a / log b}).
    end if
end algorithm
```

Examples:

- $T(n) = 4T(\frac{n}{2}) + \Theta(n^3)$: Here $a = 4$, $b = 2$, $c = 3$, and $d = 0$. Applying the technique, we compare $c = 3$ to $\frac{\log a}{\log b} = \frac{\log 4}{\log 2} = 2$. Because it is bigger, we know that $T(n)$ is dominated by the top level and $T(n) = \Theta(f(n)) = \Theta(n^3)$.
- $T(n) = 4T(\frac{n}{2}) + \Theta(\frac{n^3}{\log^3 n})$: This example is the same except for $d = -3$. Decreasing the time for the top by this little amount does not change the fact that it dominates. $T(n) = \Theta(f(n)) = \Theta(\frac{n^3}{\log^3 n})$.
- $T(n) = 27T(\frac{n}{3}) + \Theta(n^3 \log^4 n)$: Because $\frac{\log a}{\log b} = \frac{\log 27}{\log 3} = 3 = c$ and $d = 4 > -1$, all levels take roughly the same computation time and $T(n) = \Theta(f(n)\log(n)) = \Theta(n^3 \log^5 n)$.
- $T(n) = 4T(\frac{n}{2}) + \Theta(\frac{n^2}{\log n})$: Because $\frac{\log a}{\log b} = \frac{\log 4}{\log 2} = 2 = c$ and $d = -1$, this is the harmonic case and $T(n) = \Theta(n^c \cdot \log\log n) = \Theta(n^2 \log\log n)$.
- $T(n) = 4T(\frac{n}{2}) + \Theta(n)$: Because $\frac{\log a}{\log b} = \frac{\log 4}{\log 2} = 2 > 1 = c$, the computation time is sufficiently dominated by the sum of the base cases and $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^2)$.

More General Functions using Θ Notation: The above rules for approximating $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ can be generalized even further to include low order terms and an even wider range of functions $f(n)$.

Low Order Terms: Some times the subinstances formed are not exactly of size $\frac{n}{b}$, but have a stranger size like $\frac{n}{b} - \sqrt{n} + \log n - 5$. Such low order terms are not significant to the approximation of the total and can simply be ignored. We will denote the fact that such terms are included by considering recurrence relations of the form $T(n) = a \cdot T\left(\frac{n}{b} \pm o(n)\right) + f(n)$. The key is that the difference between the size and $\frac{n}{b}$ is smaller than ϵn for every constant $\epsilon > 0$.

Simple Analytical Function $f(n)$: The only requirement on f is that it is a *simple analytical* function, (See Section 1.5).

The first step in evaluating this recurrence relation is to determine the computation times $\Theta(f(n))$ and $\Theta(n^{\frac{\log a}{\log b}})$ for the top and the bottom levels of recursion. The second step is to take their ratio in order to determine whether one is "sufficiently" bigger than the other.

Dominated by Top: If $\frac{f(n)}{n^{\frac{\log a}{\log b}}} \geq n^{\Omega(1)}$, then $T(n) = \Theta(f(n))$.

Levels Arithmetic: If $\frac{f(n)}{n^{\frac{\log a}{\log b}}} = [\log n]^{\Theta(1)-1} \approx \log^d n$, for $d > -1$, then $T(n) = \Theta(f(n)\log n)$.

Levels Harmonic: If $\frac{f(n)}{n^{\frac{\log a}{\log b}}} = \Theta(\frac{1}{\log n})$, then $T(n) = \Theta(n^{\frac{\log a}{\log b}} \log\log n)$.

Dominated by Base Cases: If $\frac{f(n)}{n^{\frac{\log a}{\log b}}} \leq [\log n]^{-1-\Omega(1)} \approx \log^d n$, for $d < -1$, then $T(n) = \Theta(n^{\frac{\log a}{\log b}})$.

Recall that $\Omega(1)$ includes any increasing function and any constant c strictly bigger than zero.

Examples:

- $T(n) = 4T(\frac{n}{2}) + \Theta(n^3 \log\log n)$: The times for the top and the bottom level of recursion are $f(n) = \Theta(n^3 \log\log n)$ and $\Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{\frac{\log 4}{\log 2}}) = \Theta(n^2)$. The top level sufficiently dominates the bottom level because $f(n)/n^{\frac{\log a}{\log b}} = \frac{n^3 \log\log n}{n^2} = n^1 \log\log n \geq n^{\Omega(1)}$. Hence, we can conclude that $T(n) = \Theta(f(n)) = \Theta(n^3 \log\log n)$.
- $T(n) = 4T(\frac{n}{2}) + \Theta(2^n)$: The time for the top has increase to $f(n) = \Theta(2^n)$. Being even bigger, the top level dominates the computation even more. This can be seen because $f(n)/n^{\frac{\log a}{\log b}} = \frac{2^n}{n^2} \geq n^{\Omega(1)}$. Hence, $T(n) = \Theta(f(n)) = \Theta(2^n)$.
- $T(n) = 4T(\frac{n}{2}) + \Theta(n^2 \log\log n)$: The times for the top and the bottom level of recursion are now $f(n) = \Theta(n^2 \log\log n)$ and $\Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^2)$. Note $f(n)/n^{\frac{\log a}{\log b}} = \Theta(n^2 \log\log n/n^2) = \Theta(\log\log n) \in [\log n]^{\Theta(1)-1}$. (If $\log\log n$ is to be compared to $\log^d n$, for some d, then the closest one is $d = 0$, which is greater than -1.) It follows that $T(n) = \Theta(f(n) \log n) = \Theta(n^2 \log n \log\log n)$.
- $T(n) = 4T(\frac{n}{2}) + \Theta(\log\log n)$: The times for the top and the bottom level of recursion are $f(n) = \Theta(\log\log n)$ and $\Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{\frac{\log 4}{\log 2}}) = \Theta(n^2)$. The computation time is sufficiently dominated by the sum of the base cases because $f(n)/n^{\frac{\log a}{\log b}} = \frac{\Theta(\log\log n)}{n^2} \leq [\log n]^{-\Theta(1)-1}$. It follows that $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^2)$.
- $T(n) = 4T(\frac{n}{2} - \sqrt{n} + \log n - 5) + \Theta(n^3)$: This looks ugly. However, because $\sqrt{n} - \log n + 5$ is $o(n)$, it does not play a significant role in the approximation. Hence, the answer is the same as it would be for $T(n) = 4T(\frac{n}{2}) + \Theta(n^3)$.

This completes the summary of the results.

1.6.3 The Classic Techniques

We now present a few of the classic techniques for computing recurrence relations. We will continue to focus on recurrence relations of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ with $T(\Theta(1)) = \Theta(1)$. Later in Section 1.6.4, we will apply the same idea to other recurrence relations.

Guess and Verify: Consider the example $T(n) = 4T\left(\frac{n}{2}\right) + n$ and $T(1) = 1$. If we could guess that the solution is $T(n) = 2n^2 - n$, we could verify this answer in the following two ways.

Plugging In: The first way to verify that $T(n) = 2n^2 - n$ is the solution is to simply plug it into the two equations $T(n) = 4T\left(\frac{n}{2}\right) + n$ and $T(1) = 1$ and make sure that they are satisfied.

Left Side	Right Side
$T(n) = 2n^2 - n$	$4T(\frac{n}{2}) + n = 4\left[2\left(\frac{n}{2}\right)^2 - \left(\frac{n}{2}\right)\right] - n = 2n^2 - n$
$T(1) = 2n^2 - n = 1$	1

Proof by Induction: Similarly, we can use induction to prove that this is the solution for all n, (at least for all powers of 2, namely $n = 2^i$).

Base Case: Because $T(1) = 2(1)^2 - 1 = 1$, it is correct for $n = 2^0$.

Induction Step: Let $n = 2^i$. Assume that it is correct for $2^{i-1} = \frac{n}{2}$. Because $T(n) = 4T(\frac{n}{2}) + n = 4\left[2\left(\frac{n}{2}\right)^2 - \left(\frac{n}{2}\right)\right] - n = 2n^2 - n$, it is also true for n.

Guess Form and Calculate Coefficients: Suppose that instead of guessing that the solution to $T(n) = 4T\left(\frac{n}{2}\right) + n$ and $T(1) = 1$ is $T(n) = 2n^2 - n$, we are only able to guess that it has the form $T(n) = an^2 + bn + c$ for some constants a, b, and c. We can plug in this solution as done before and solve for the a, b, and c.

Left Side	Right Side
$T(n) = an^2 + bn + c$	$4T(\frac{n}{2}) + n = 4\left[a\left(\frac{n}{2}\right)^2 + b\left(\frac{n}{2}\right) + c\right] - n = an^2 + (2b+1)n + 4c$
$T(1) = a + b + c$	1

These left and right sides must be equal for all n. Both have a as the coefficient of n^2, which is good. To make the coefficient in front n be the same, we need that $b = 2b + 1$, which gives $b = -1$. To make the constant coefficient be the same, we need that $c = 4c$, which gives $c = 0$. To make $T(1) = a(1)^2 + b(1) + c = a(1)^2 - (1) + 0 = 1$, we need that $a = 2$. This gives us the solution $T(n) = 2n^2 - n$ that we had before.

Guessing $\Theta(T(n))$: Another option is to prove by induction that the solution is $T(n) = \mathcal{O}(n^2)$. Sometimes one can proceed as above, but I find that often one gets stuck.

Induction Step: Suppose that by induction we assumed that $T\left(\frac{n}{2}\right) \leq c \cdot \left(\frac{n}{2}\right)^2$ for some constant $c > 0$. Then we would try to prove that $T(n) \leq c \cdot n^2$ for the same constant c. We would proceed by as follows. $T(n) = 4T\left(\frac{n}{2}\right) + n \leq 4\left[c \cdot \left(\frac{n}{2}\right)^2\right] + n = c \cdot n^2 + n$. However, no matter how big we make c, this is not smaller than $c \cdot n^2$. Hence, this proof failed.

Guessing Whether Top or Bottom Dominates: Our goal is to estimate $T(n)$. We have that $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$. One might first ask whether the $a \cdot T\left(\frac{n}{b}\right)$ or the $f(n)$ is more significant. There are three possibilities. The first is that $f(n)$ is much bigger than $a \cdot T\left(\frac{n}{b}\right)$. The second is that $a \cdot T\left(\frac{n}{b}\right)$ is much bigger than $f(n)$. The third is that the terms are close enough in value that they both make a significant contribution. The third case will be harder to analyze. But let us see what we can do in the first two cases.

$f(n)$ is much bigger than $a \cdot T\left(\frac{n}{b}\right)$: It follows that $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) = \Theta(f(n))$ and hence, we are done. The function $f(n)$ is given and the answer is $T(n) = \Theta(f(n))$.

$a \cdot T\left(\frac{n}{b}\right)$ is much bigger than $f(n)$: It follows that $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \approx a \cdot T\left(\frac{n}{b}\right)$. This case is harder. We must still solve this equation. With some insight, let us guess that the general form is $T(n) = n^\alpha$ for some constant α. We will plug this guess into the equation $T(n) = a \cdot T\left(\frac{n}{b}\right)$ and solve for α. Plugging this in gives $n^\alpha = a \cdot \left(\frac{n}{b}\right)^\alpha$. Dividing out the n^α, gives $1 = a \cdot \left(\frac{1}{b}\right)^\alpha$. Bringing over the b^α gives $b^\alpha = a$. Taking the log gives $\alpha \cdot \log b = \log a$ and solving gives $\alpha = \frac{\log a}{\log b}$. The conclusion is that the solution to $T(n) = a \cdot T\left(\frac{n}{b}\right)$ is $T(n) = \Theta(n^{\frac{\log a}{\log b}})$.

This was by no means formal. But it does give some intuition that the answer may well be $T(n) = \Theta(\max(f(n), n^{\frac{\log a}{\log b}}))$. We will now try to be more precise.

Unwinding $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$: Our first formal step will be to express $T(n)$ as the sum of the work completed at each level of the recursion. Later we will evaluate this sum. One way to express $T(n)$ as a sum is to unwind it as follows.

$$\begin{aligned}
T(n) &= f(n) + a \cdot T\left(\frac{n}{b}\right) \\
&= f(n) + a \cdot \left[f\left(\frac{n}{b}\right) + a \cdot T\left(\frac{n}{b^2}\right)\right] \\
&= f(n) + af\left(\frac{n}{b}\right) + a^2 \cdot T\left(\frac{n}{b^2}\right) \\
&= f(n) + af\left(\frac{n}{b}\right) + a^2 \cdot \left[f\left(\frac{n}{b^2}\right) + a \cdot T\left(\frac{n}{b^3}\right)\right]
\end{aligned}$$

$$\begin{aligned} &= f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + a^3 \cdot T\left(\frac{n}{b^3}\right) \\ &= \cdots \\ &= \sum_{i=0}^{h-1} a^i \cdot f\left(\frac{n}{b^i}\right) \quad + a^h \cdot T(1) = \Theta\left(\sum_{i=0}^{h} a^i \cdot f\left(\frac{n}{b^i}\right)\right) \end{aligned}$$

Levels of The Recursion Tree: We can understand the unwinding of the recurrence relation into the above sum better when we examine the tree consisting of one node for every stack frame executed during the computation.

Recall that the recurrence relation $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ relates to the computation time of a recursive program, such that when it is given an instances of size n, it recurses a times on subinstances of size $\frac{n}{b}$. $f(n)$ denotes the computation time required by a single stack frame on an instance of size n and $T(n)$ denotes the entire computation time to recurse.

Level 0: The top level of the recursion has an instance of size n. As stated, excluding the time to recurse, this top stack frame requires $f(n)$ computation time to generate the subinstances and recombining their solutions.

Level 1: The single top level stack frame recurses a times. Hence, the second level (level 1) has a stack frames. Each of them is given a subinstance of size $\frac{n}{b}$ and hence requires $f(\frac{n}{b})$ computation time. It follows that the total computation time for this level is $a \cdot f(\frac{n}{b})$.

Level 2: Each of the a stack frames at the level 1 recurses a times giving a^2 stack frames at the level 2. Each stack frame at the level 1 has a subinstance of size $\frac{n}{b}$, hence they recurse on subsubinstances of size $\frac{n}{b^2}$. Therefore, each stack frame at the level 2 requires $f(\frac{n}{b^2})$ computation time. It follows that the total computation time for this level is $a^2 \cdot f(\frac{n}{b^2})$.

Level 3, etc. $\cdots$

Level i: Each successive level, the number of stack frames goes up by a factor of a and the size of the subinstance goes down by a factor of b.

Number of Stack Frames at this Level: It follows that at the level i, we have a^i subinstances

Size of Subinstances at this Level: Each subinstance at this level has size $\frac{n}{b^i}$.

Time for each Stack Frame at this Level: Given this size, the time taken is $f(\frac{n}{b^i})$.

Total Time for Level: The total time for this level is the number at the level times the time for each, namely $a^i \cdot f(\frac{n}{b^i})$.

Level h: The recursive program stops recursing when it reaches a *base case*.

Size of the Base Case: A base case is an input instance that has some minimum size. Whether this size is zero, one, or two, will not change the approximation $\Theta(T(n))$ of the total time. This is why we specify only that the size of the base case is $\Theta(1)$.

The Number of Levels h: Let h denote the depth of the recursion tree. If an instance the level i has size $\frac{n}{b^i}$, then at the level h, it will have size $\frac{n}{b^h}$. Now we have a choice as to which constant we would like to use for the base case. Being lazy, let us use a constant that makes our life easy.

Size One: If we set the size of the base case to be one, then this gives us the equation $\frac{n}{b^h} = 1$, and the number of levels needed to achieve this is $h = \frac{\log n}{\log b}$.

Size Two: If instead, we set the base case size to be $\frac{n}{b^h} = 2$, then the number of levels needed, $h = \frac{\log(n)-1}{\log b}$. Though this is only a constant less, it is messier.

Size Zero: Surely recursing to instances of size zero will also be fine. The equation becomes $\frac{n}{b^h} = 0$. Solving this for h gives $h = \infty$. Practically, what does this mean? You may have heard of zeno's paradox. If you cut the remaining distance in half and then in half again and so on, then though you get very close very fast, you never actually get there.

Time for a Base Case: Changing the computation time for a base case will change the total running time by only a multiplicative constant. This is why we specify only that it requires some fixed time. We write this as $T(\Theta(1)) = \Theta(1)$.

Number of Base Cases: The number of stack frames at the level i is a^i. Hence, the number of these base case subinstances is $a^h = a^{\frac{\log n}{\log b}}$. This looks really ugly. Is it exponential in n, or polynomial?, or something else weird? The place to look for help is Section 1.2, which gives rules on logarithms and exponentials. One rule state that you can move things between the base and the exponent as long as you add or remove a log. This gives that the number of base cases is $a^h = a^{\frac{\log n}{\log b}} = n^{\frac{\log a}{\log b}}$. Given that $\frac{\log a}{\log b}$ is simply some constant, $n^{\frac{\log a}{\log b}}$ is a simple polynomial in n.

Time for Base Cases: This gives the total time for the base cases to be $a^h \cdot T(1) = \Theta(n^{\frac{\log a}{\log b}})$. This is the same time that we got before.

Sum of Time for Each Level: We obtain the total time $T(n)$ for the recursion by summing up the time at each of these levels. This gives

$$T(n) = \left[\sum_{i=0}^{h-1} a^i \cdot f\left(\frac{n}{b^i}\right)\right] + a^h \cdot T(1) = \Theta\left(\sum_{i=0}^{h} a^i \cdot f\left(\frac{n}{b^i}\right)\right).$$

Key things to remember about this sum is that it has $\Theta(\log n)$ terms, the first term being $\Theta(f(n))$ and the last being $\Theta(n^{\frac{\log a}{\log b}})$.

Some Examples: When working through an example, the following questions are useful.

Dominated by	Base Cases	Levels Arithmetic	Top Level
Examples	$T(n) = 4T(n/2) + n$	$T(n) = 9T(n/3) + n^2$	$T(n) = 2T(n/4) + n^2$
a) # frames at the i^{th} level?	4^i	9^i	2^i
b) Size of instance at the i^{th} level?	$\frac{n}{2^i}$	$\frac{n}{3^i}$	$\frac{n}{4^i}$
c) Time within one stack frame	$f\left(\frac{n}{2^i}\right) = \left(\frac{n}{2^i}\right)$	$f\left(\frac{n}{3^i}\right) = \left(\frac{n}{3^i}\right)^2$	$f\left(\frac{n}{4^i}\right) = \left(\frac{n}{4^i}\right)^2$
d) # levels	$\frac{n}{2^h} = 1$ $h = \frac{\log n}{\log 2} = \Theta(\log n)$	$\frac{n}{3^h} = 1$ $h = \frac{\log n}{\log 3} = \Theta(\log n)$	$\frac{n}{4^h} = 1$ $h = \frac{\log n}{\log 4} = \Theta(\log n)$
e) # base case stack frames	$4^h = 4^{\frac{\log n}{\log 2}}$ $= n^{\frac{\log 4}{\log 2}} = n^2$	$9^h = 9^{\frac{\log n}{\log 3}}$ $= n^{\frac{\log 9}{\log 3}} = n^2$	$2^h = 2^{\frac{\log n}{\log 4}}$ $= n^{\frac{\log 2}{\log 4}} = n^{\frac{1}{2}}$
f) $T(n)$ as a sum	$\sum_{i=0}^{h}$(# at level) $\cdot$ (time each) $= \sum_{i=0}^{\Theta(\log n)} 4^i \cdot \left(\frac{n}{2^i}\right)$ $= n \cdot \sum_{i=0}^{\Theta(\log n)} 2^i$	$\sum_{i=0}^{h}$(# at level) $\cdot$ (time each) $= \sum_{i=0}^{\Theta(\log n)} 9^i \cdot \left(\frac{n}{3^i}\right)^2$ $= n^2 \cdot \sum_{i=0}^{\Theta(\log n)} 1$	$\sum_{i=0}^{h}$(# at level) $\cdot$ (time each) $= \sum_{i=0}^{\Theta(\log n)} 2^i \cdot \left(\frac{n}{4^i}\right)^2$ $n^2 \cdot \sum_{i=0}^{\Theta(\log n)} \left(\frac{1}{8}\right)^i$
g) Dominated by?	geometric increasing: dominated by last term = number of base cases	arithmetic sum: levels roughly the same = time per level $\cdot$ # levels	geometric decreasing: dominated by first term = top level
h) Θ(T(n))	$T(n) = \Theta\left(n^{\frac{\log a}{\log b}}\right)$ $= \Theta\left(n^{\frac{\log 4}{\log 2}}\right) = \Theta\left(n^2\right)$	$T(n) = \Theta\left(f(n)\log n\right)$ $= \Theta\left(n^2 \log n\right)$	$T(n) = \Theta\left(f(n)\right)$ $= \Theta\left(n^2\right)$

Evaluating The Sum: To evaluate this sum, we will use the Adding Made Easy Approximations given in Section 1.5. The four cases below correspond to the sum being geometric increasing, arithmetic, harmonic, or bounded tail.

Dominated by Top: If the computation time $\Theta(f(n))$ for the top level of recursion is sufficiently bigger than the computation time $\Theta(n^{\frac{\log a}{\log b}})$ for the base cases, then the above sum decreases geometrically. The Adding Made Easy Approximations then tells us that such a sum is bounded by the first term, giving that $T(n) = \Theta(f(n))$. In this case, we say that the computation time of the recursive program is dominated by the time for the top level of the recursion.

Sufficiently Big: A condition for $\Theta(f(n))$ to be sufficiently bigger than $\Theta(n^{\frac{\log a}{\log b}})$ is that $f(n) \geq n^c \cdot \log^d n$, where $c > \frac{\log a}{\log b}$. A more general condition can be expressed as $f(n)/n^{\frac{\log a}{\log b}} \geq n^{\Omega(1)}$, where f is a simple analytical function.

Proving Geometric Decreasing: What remains is to prove that the sum is in fact geometric decreasing. To simplify things, consider $f(n) = n^c$ for constant c. We want to evaluate $T(n) = \Theta(\sum_{i=0}^h a^i f(\frac{n}{b^i})) = \Theta(\sum_{i=0}^h a^i (\frac{n}{b^i})^c) = \Theta(n^c \cdot \sum_{i=0}^h (\frac{a}{b^c})^i)$. The Adding Made Easy Approximations state that this sum decreases geometrically if the terms decrease exponentially in terms of i (not in terms of n). This is the case as long as the constant $\frac{a}{b^c}$ is less than one. This occurs when $a < b^c$ or equivalently when $c > \frac{\log a}{\log b}$.

Even Bigger: Having an $f(n)$ that is even bigger means that the time is even more dominated by the top level of recursion. For example, $f(n)$ could be 2^n or 2^{2^n}. Having $f(n)/n^{\frac{\log a}{\log b}} \geq n^{\Omega(1)}$ ensures that $f(n)$ is at least as big as n^c for some $c > \frac{\log a}{\log b}$. Hence, for these it is still the case that $T(n) = \Theta(f(n))$.

A Little Smaller: This function $f(n) \geq n^c \cdot \log^d n$ with $c > \frac{\log a}{\log b}$ is a little smaller when we let d be negative. However, this small change does not change the fact that the time is dominated by the top level of recursion and that $T(n) = \Theta(f(n))$.

Levels Arithmetic: If the computation time $\Theta(f(n))$ for the top levels of recursion is sufficiently close to the computation time $\Theta(n^{\frac{\log a}{\log b}})$ for the base cases, then all levels of the recursion contribute roughly the same to the computation time and hence the sum is arithmetic. The Adding Made Easy Approximations give that the total is the number of terms times the "last" term. The number of terms is the height of the recursion which is $h = \frac{\log(n)}{\log b} = \Theta(\log(n))$. A complication, however, is that the terms are in the reverse order from what they should be to be arithmetic. Hence, the "last" term is top one, $\Theta(f(n))$. This gives $T(n) = \Theta(f(n) \log n)$.

Sufficiently The Same: A condition for $\Theta(f(n))$ to be sufficiently close to $\Theta(n^{\frac{\log a}{\log b}})$ is that $f(n) = \Theta(n^c \cdot \log^d n)$, where $c = \frac{\log a}{\log b}$ and $d > -1$. A more general condition can be expressed as $f(n)/n^{\frac{\log a}{\log b}} = [\log n]^{\Theta(1)-1}$, where f is a simple analytical function.

A Simple Case $f(n) = n^{\frac{\log a}{\log b}}$: To simplify things, let us first consider $f(n) = n^c$ where $c = \frac{\log a}{\log b}$. Above we computed $T(n) = \Theta(\sum_{i=0}^h a^i f(\frac{n}{b^i})) = \Theta(\sum_{i=0}^h a^i (\frac{n}{b^i})^c) = \Theta(n^c \cdot \sum_{i=0}^h (\frac{a}{b^c})^i)$. But now c is chosen so that $\frac{a}{b^c} = 1$. This simplifies the sum to $T(n) = \Theta(n^c \cdot \sum_{i=0}^h (1)^i) = \Theta(n^c h) = \Theta(f(n) \log n)$. Clearly, this sum is arithmetic.

A Harder Case $f(n) = n^{\frac{\log a}{\log b}} \log^d n$: Now let us consider how much $f(n)$ can be shifted away from $f(n) = n^{\frac{\log a}{\log b}}$ and still have the sum be arithmetic. Let us consider $f(n) = n^c \cdot \log^d n$, where $c = \frac{\log a}{\log b}$ and $d > -1$. This adds an extra log to the sum, giving $T(n) = \Theta(\sum_{i=0}^h a^i f(\frac{n}{b^i})) = \Theta(\sum_{i=0}^h a^i (\frac{n}{b^i})^c \log^d(\frac{n}{b^i})) = \Theta(n^c \cdot \sum_{i=0}^h (1)^i \left[\log(\frac{n}{b^i})\right]^d) = \Theta(n^c \cdot \sum_{i=0}^h [\log(n) - i\log(b)]^d)$. This looks ugly, but we can simplify it by reversing the order of the terms.

Reversing the Terms: The expression $\frac{n}{b^i}$ with $i \in [0, h]$ takes on the values $n, \frac{n}{b}, \frac{n}{b^2}, \cdots 1$. Reversing the order of these values gives $1, b, b^2, \cdots, n$. This is done more formally by letting $i = h - j$ so that $\frac{n}{b^i} = \frac{n}{b^{h-j}} = b^j$. Now summing the terms in reverse order with $j \in [0, h]$ gives $T(n) = \Theta(n^c \cdot \sum_{i=0}^h \left[\log(\frac{n}{b^i})\right]^d) = \Theta(n^c \cdot \sum_{j=0}^h \left[\log(b^j)\right]^d) = \Theta(n^c \cdot \sum_{j=0}^h [j \log b]^d)$. This $[\log b]^d$ is a constant that we can hide in the theta. This gives $T(n) = \Theta(n^c \cdot \sum_{j=0}^h j^d)$.

Proving Reverse Arithmetic: The Adding Made Easy Approximations state that this sum $T(n) = \Theta(n^c \cdot \sum_{j=0}^h j^d)$ is arithmetic as long as $d > -1$. In this case, the total is $T(n) = \Theta(n^c \cdot h \cdot h^d) = \Theta(n^c \cdot \log^{d+1} n)$. Another way of viewing this sum is that it is approximately the "last term", which after reversing the order is the top term $f(n)$, times the number of terms, which is $h = \Theta(\log n)$. In conclusion, $T(n) = \Theta(f(n) \log n)$.

Levels Harmonic: Taking the Adding Made Easy Approximations the obvious next step, we let $d = -1$ in the above calculations in order to make a harmonic sum.

Strange Requirement: This new requirement is that $f(n) = \Theta(n^c \cdot \log^d n)$, where $c = \frac{\log a}{\log b}$ and $d = -1$.

The Total: Continuing the above calculations gives $T(n) = \Theta(n^c \cdot \sum_{j=0}^{h} j^d) = \Theta(n^c \cdot \sum_{j=0}^{h} \frac{1}{j})$. This is a harmonic sum adding to $T(n) = \Theta(n^c \cdot \log h) = \Theta(n^c \log\log n) = \Theta(n^{\frac{\log a}{\log b}} \log\log n)$ as required.

Dominated by Base Cases: If the computation time $\Theta(f(n))$ for the top levels of recursion is sufficiently smaller than the computation time $\Theta(n^{\frac{\log a}{\log b}})$ for the base cases, then the terms increase quickly enough for the sum to be bounded by the last term. This last term consists of $\Theta(1)$ computation time for each of the $n^{\frac{\log a}{\log b}}$ base cases. In conclusion, $T(n) = \Theta(n^{\frac{\log a}{log b}})$.

Sufficiently Big: A condition for $\Theta(f(n))$ to be sufficiently smaller than $\Theta(n^{\frac{\log a}{\log b}})$ is that $f(n) \leq n^c \cdot \log^d n$, where either $c < \frac{\log a}{\log b}$ or ($c = \frac{\log a}{\log b}$ and $d < -1$). A more general condition can be expressed as $\frac{f(n)}{n^{\frac{\log a}{\log b}}} \leq [\log n]^{-1-\Omega(1)}$, where f is a simple analytical function.

Geometric Increasing: To begin, suppose that $f(n) = n^c$ for constant $c < \frac{\log a}{\log b}$. Before we evaluated $T(n) = \Theta(\sum_{i=0}^{h} a^i f(\frac{n}{b^i})) = \Theta(\sum_{i=0}^{h} a^i (\frac{n}{b^i})^c) = \Theta(n^c \cdot \sum_{i=0}^{h} (\frac{a}{b^c})^i)$. $c < \frac{\log a}{\log b}$ gives $\frac{a}{b^c} > 1$. Hence, this sum increases exponentially and the total is bounded by the last term, $T(n) = \Theta(n^{\frac{\log a}{log b}})$.

Bounded Tail: With the Adding Made Easy Approximations, however, we learned that the sum does not need to be exponentially decreasing (we are viewing the sum in reverse order) in order to have a bounded tail. Let us now consider $f(n) = n^c \cdot \log^d n$, where $c = \frac{\log a}{\log b}$ and $d < -1$. Above we got that $T(n) = \Theta(n^c \cdot \sum_{j=0}^{h} j^d)$. This is a bounded tail, summing to $T(n) = \Theta(n^c)\Theta(1) = \Theta(n^{\frac{\log a}{log b}})$ as required.

Even Smaller: Having an $f(n)$ that is even smaller means that the time is even less effected by the time for the top level of recursion. This can be expressed as $\frac{f(n)}{n^{\frac{\log a}{\log b}}} \leq [\log n]^{-1-\Omega(1)}$.

Low Order Terms: If the recurrence relation is $T(n) = 4T(\frac{n}{2} - \sqrt{n} + \log n - 5) + \Theta(n^3)$ instead of $T(n) = 4T(\frac{n}{2}) + \Theta(n^3)$, these low order terms, though ugly, do not make a significant difference to the total. We will not prove this formally.

This concludes our approximation of $\Theta(T(n))$.

1.6.4 Other Examples

We will now apply the same idea to other recurrence relations.

Exponential Growth for a Linear Number of Levels: Here we consider recurrence relations of the form $T(n) = aT(n-b) + f(n)$ for $a > 1$, $b > 0$, and later $c > 0$. Each instances having a subinstances means that as before, the number of subinstances grows exponentially. Decreasing the subinstance by only an additive amount means that the number of levels of recursion is linear in the size n of the initial instance. This gives an exponential number of base cases, which will dominate the time unless, the work $f(n)$ in each stack frame is exponential itself.

Dominated by	Base Cases	Levels Arithmetic	Top Level
Examples	$T(n) = aT(n-b) + n^c$	$T(n) = aT(n-b) + a^{\frac{1}{b}n}$	$T(n) = aT(n-b) + a^{\frac{2}{b}n}$
a) # frames at the i^{th} level?	a^i		
b) Size of instance at the i^{th} level?	$n - i \cdot b$		
c) Time within one stack frame	$f(n - i \cdot b) = (n - i \cdot b)^c$	$f(n - i \cdot b) = a^{\frac{n-i \cdot b}{b}} = a^{\frac{1}{b}n} \cdot a^{-i}$	$f(n - i \cdot b) = a^{\frac{2(n-i \cdot b)}{b}} = a^{\frac{2}{b}n} \cdot a^{-2i}$
d) # levels	$n - h \cdot b = 0$, $h = \frac{n}{b}$ Having a base case of size zero makes the math the cleanest.		
e) # base case stack frames	$a^h = a^{\frac{1}{b}n}$		
f) $T(n)$ as a sum	$\sum_{i=0}^{h}$(# at level) $\cdot$ (time each) $= \sum_{i=0}^{\frac{n}{b}} a^i \cdot (n - i \cdot b)^c$	$\sum_{i=0}^{h}$(# at level) $\cdot$ (time each) $= \sum_{i=0}^{\frac{n}{b}} a^i \cdot a^{\frac{1}{b}n} \cdot a^{-i}$ $= a^{\frac{1}{b}n} \cdot \sum_{i=0}^{\frac{n}{b}} 1$	$\sum_{i=0}^{h}$(# at level) $\cdot$ (time each) $= \sum_{i=0}^{\frac{n}{b}} a^i \cdot a^{\frac{2}{b}n} \cdot a^{-2i}$ $= a^{\frac{2}{b}n} \cdot \sum_{i=0}^{\frac{n}{b}} a^{-i}$
g) Dominated by?	geometric increasing: dominated by last term = number of base cases	arithmetic sum: levels roughly the same = time per level $\cdot$ # levels	geometric decreasing: dominated by first term = top level
h) Θ(T(n))	$T(n) = \Theta\left(a^{\frac{n}{b}}\right)$	$T(n) = \Theta\left(\frac{n}{b} \cdot a^{\frac{n}{b}}\right)$	$T(n) = \Theta\left(a^{\frac{2}{b}n}\right)$

Fibonacci Numbers: A famous recurrence relation is $Fib(0) = 0$, $Fib(1) = 1$, and $Fib(n) = Fib(n-1) + Fib(n-2)$. Clearly, $Fib(n)$ grows slower than $T(n) = 2T(n-1) = 2^n$ and faster than $T(n) = 2T(n-2) = 2^{n/2}$. It follows that $Fib(n) = 2^{\Theta(n)}$. If one wants more details than the following calculations give that $Fib(n) = \frac{1}{\sqrt{5}}\left[\left[\frac{1+\sqrt{5}}{2}\right]^n - \left[\frac{1-\sqrt{5}}{2}\right]^n\right] = \Theta((1.61..)^n)$.

Careful Calculations: Let us guess the solution $Fib(n) = \alpha^n$. Plugging this into $Fib(n) = Fib(n-1) + Fib(n-2)$ gives that $\alpha^n = \alpha^{n-1} + \alpha^{n-2}$. Dividing through by α^{n-2} gives $\alpha^2 = \alpha + 1$. Using the formula for quadratic roots gives that either $\alpha = \frac{1+\sqrt{5}}{2}$ or $\alpha = \frac{1-\sqrt{5}}{2}$. Any linear combination of these two solutions will also be a valid solution, namely $Fib(n) = c_1 \cdot (\alpha_1)^n + c_2 \cdot (\alpha_2)^n$. Using the fact that $Fib(0) = 0$ and $Fib(1) = 1$ and solving for c_1 and c_2 gives that $Fib(n) = \frac{1}{\sqrt{5}}\left[\left[\frac{1+\sqrt{5}}{2}\right]^n - \left[\frac{1-\sqrt{5}}{2}\right]^n\right]$.

One Subinstance Each: If each stackframe calls recursively at most once, then there is only a single line of stackframes, one per level. The total time $T(n)$ is the sum of these stack frames.

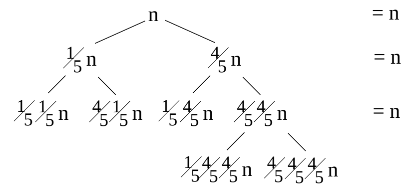
Dominated by	Base Cases	Levels Arithmetic	Top Level
Examples	$T(n) = T(n-b) + 0$	$T(n) = T(n-b) + n^c$	$T(n) = T(n-b) + 2^{cn}$
a) # frames at the i^{th} level?	one		
b) Size of instance at the i^{th} level?	$n - i \cdot b$		
c) Time within one stack frame	$f(n - i \cdot b) =$ zero Except for the base case which has work $\Theta(1)$	$f(n - i \cdot b) = (n - i \cdot b)^c$	$f(n - i \cdot b) = 2^{c(n-i \cdot b)}$
d) # levels	$n - h \cdot b = 0$, $h = \frac{n}{b}$		
e) # base case stack frames	one		
f) $T(n)$ as a sum	$\sum_{i=0}^{h}$(# at level) $\cdot$ (time each) $= \left(\sum_{i=0}^{\frac{n}{b}-1} 1 \cdot 0\right) + 1 \cdot \Theta(1)$ $= \Theta(1)$	$\sum_{i=0}^{h}$(# at level) $\cdot$ (time each) $= \sum_{i=0}^{\frac{n}{b}} 1 \cdot (n - i \cdot b)^c$	$\sum_{i=0}^{h}$(# at level) $\cdot$ (time each) $= \sum_{i=0}^{\frac{n}{b}} 1 \cdot 2^{c(n-i \cdot b)}$
g) Dominated by?	geometric increasing: dominated by last term = number of base cases	arithmetic sum: levels roughly the same = time per level $\cdot$ # levels	geometric decreasing: dominated by first term = top level
h) Θ(T(n))	$T(n) = \Theta(1)$	$T(n) = \Theta\left(\frac{n}{b} \cdot n^c\right) = \Theta\left(n^{c+1}\right)$	$T(n) = \Theta(2^{cn})$

Recursing on Instances of Different Sizes and Linear Work: We will now consider recurrence relations of the form $T(n) = T(u \cdot n) + T(v \cdot n) + \Theta(n)$ for constants $0 < u, v < 1$. Note that each instance does an amount of work that is linear in its input instance and it produces two smaller subinstances of sizes $u \cdot n$ and $v \cdot n$. Such recurrence relations will arise in Section 11.2.1 for quick sort.

$u = v = \frac{1}{b}$: To tie in with what we did before, note that if $u = v = \frac{1}{b}$, then this recurrence relation becomes $T(n) = 2T(\frac{1}{b}n) + \Theta(n)$. We have seen that the time is $T(n) = \Theta(n)$ for $b > 2$ or $u = v < \frac{1}{2}$, $T(n) = \Theta(n \log n)$ for $u = v = \frac{1}{2}$, and $T(n) = \Theta(n^{\frac{\log 2}{\log b}})$ for $u = v > \frac{1}{2}$.

For $u + v = 1$, $T(n) = \Theta(n \log n)$: The pivotal values $u = v = \frac{1}{2}$ is generalized to $u + v = 1$. The key here is that each stack frame forms two subinstances by splitting its input instance into two pieces, whose sizes sum to that of the given instance.

Total of Instance Sizes at Each Level is n: If you trace out the tree of stack frames, you will see that at each level of the recursion, the sum of the sizes of the input instances is $\Theta(n)$, at least until some base cases are reached. To make this concrete, suppose $u = \frac{1}{5}$ and $v = \frac{4}{5}$. The sizes of the instance at each level are given below. Note the sum of any sibling pair sums to their parent and the total at the each level is n.



Total of Work at Each Level is n: Because the work done by each stack frame is linear in its size and the sizes at each level sums to n, it follows that the work at each level adds up to $\Theta(n)$. As an example, if sum of their sizes is $(\frac{1}{5})(\frac{1}{5})n + (\frac{4}{5})(\frac{1}{5})n + (\frac{1}{5})(\frac{4}{5})n + (\frac{4}{5})(\frac{4}{5})n = n$, then the sum of their work is $c(\frac{1}{5})(\frac{1}{5})n + c(\frac{4}{5})(\frac{1}{5})n + c(\frac{1}{5})(\frac{4}{5})n + c(\frac{4}{5})(\frac{4}{5})n = cn$. Note that the same would not be true if the time in a stack frame was say quadratic, $c\left((\frac{1}{5})(\frac{1}{5})n\right)^2$.

Number of Levels is $\Theta(logn)$: The left-left-left branch terminates quickly after $\frac{\log n}{\log u} = 0.43 \log n$ for $u = \frac{1}{5}$. The right-right-right branch terminates slower, but still after only $\frac{\log n}{\log u} = 3.11 \log n$ levels. The middle paths terminate somewhere in between these two extremes.

Total Work is $\Theta(n \log n)$: Given that there is $\Theta(n)$ work at each of $\Theta(\log n)$ levels gives that the total work is $T(n) = \Theta(n \log n)$.

More Parts: The same principle applies no matter how many parts the instance is split into. For example, the solution of $T(n) = T(u \cdot n) + T(v \cdot n) + \ldots + T(w \cdot n) + \Theta(n)$ is $T(n) = \Theta(n \log n)$ as long as $u + v + \ldots + w = 1$.

Less Even Split Means Bigger Constant: Though solution of $T(n) = T(u \cdot n) + T((1-u) \cdot n) + \Theta(n)$ is $T(n) = \Theta(n \log n)$ for all constants $0 < u < 1$, the constant hidden in the Θ for $T(n)$ is smaller when the split is closer to $u = v = \frac{1}{2}$. If you are interested, the constant is roughly $\frac{1}{u} / \log(\frac{1}{u})$ for small u, which grows as u gets small.

For $u + v < 1$, $T(n) = \Theta(n)$: If $u + v < 1$, then the sum of the sizes of the sibling subinstances is smaller than that of the parent subinstance. In this case, the sum work at each level decreases by the constant factor $(u + v)$ each level. The total work is then $T(n) = \sum_{i=0}^{\Theta(\log n)} (u+v)^i \cdot n$. This computation is dominated by the top level and hence $T(n) = \Theta(n)$.

For $u + v > 1$, $T(n) = \Theta(n^?)$: In contrast, if $u + v > 1$, then the sum of the sizes of the sibling subinstances is larger than that of the parent subinstance and the sum work at each level increases by the constant factor $(u + v)$ each level. The total work is dominated by the base cases. The total time will be $T(n) = \Theta(n^\alpha)$ for some constant $\alpha > 1$. One could compute how this α is a function of u and v, but we will not bother.

This competes the discussion on recurrence relations.

Chapter 2

Abstractions

It is hard to think about love in terms of the firing of neurons or to think about a complex data base as a sequence of magnetic charges on a disk. In fact, a ground stone of human intelligence is the ability to build hierarchies of abstractions within which to understand things. This requires cleanly partitioning details into objects, attributing well defined properties and human characteristics to these objects, and seeing the whole as being more than the sum of the parts. This book focuses on different abstractions, analogies, and paradigms of algorithms and of data structures. The tunneling of electrons through doped silicon are abstracted as AND, OR, and NOT gate operations; which are abstracted as machine code operations; the executing of lines of Java code; subroutines; algorithms; and the abstractions of algorithmic concepts. Magnetic charges on a disk are abstracted as sequences of zeros and ones; which are abstracted as integers, strings and pointers; which are abstracted as a graph; which is used to abstract a system of trucking routes. The key to working within a complex system is the ability to both simultaneously and separately operate with each of these abstractions and to be able to translate this work between the levels. In addition, does one need to limit themselves to one hierarchy of abstractions about something. It is useful to use different notations, analogies, and metaphors to view the same ideas. Each representation at your disposal provides new insights and new tools.

2.1 Different Representations of Algorithms

In your courses and in your jobs, you will need to be proficient not only at writing working code, but also at understanding algorithms, designing new algorithms, and describing algorithms to others in such way that their correctness is transparent. Some useful ways of representing an algorithm for this purpose are code, tracing a computation on an example input, mathematical logic, personified analogies and metaphors, and various other higher abstractions. These can be so different from each other that it is hard to believe that they describe the same algorithm. Each has its own advantages and disadvantages.

Computers vs Humans: One spectrum in which different representations are different is that from what is useful for computers to what is useful for humans. Machine code is necessary for computers because they can only blindly follow instructions, but would be painfully tedious for a human. Higher and higher level programming languages are being developed with which it is easier for humans to develop and understand algorithms. Many people (and texts) have the perception that being able to code is the only representation of an algorithm that they will ever need. However, until compilers get much better, I recommend using even higher level abstractions for giving a human an intuitive understanding of an algorithm.

What vs Why: Code focuses on what the algorithm does. We, however, need to understand why it works.

Concrete vs Abstract: It is useful when attempting to understand or explain an algorithm to trace out the code on a few example input instances. This has the advantages of being concrete, dynamic, and often visual. It is best if the simplest examples are found that capture the key ideas. Elaborate examples are prone to mistakes, and it is hard to find the point being made amongst all of the details. However, this process does not prove that the algorithm gives the correct answer on every possible input. To do this, you must be able to prove it works for an abstract arbitrary input. Neither does tracing the algorithm provide the higher-level intuition needed. For this, one needs to see the pattern in what the code is doing and convince him or herself that this pattern of actions solves the problem.

Details vs Big Picture: Computers require that every implementation detail is filled in. Humans, on the other hand, are more comfortable with a higher level of abstraction. Lots of implementation details distract from the underlying meaning. Gauge your audience. Leave some details until later and don't insult them by giving those that he or she can easily fill in. One way of doing this is by describing how subroutines and data structures are used without initially describing how they are implemented. On the other hand, I recommend initially ignoring code and instead use the other algorithmic abstractions that we will cover in this text. Once the big picture is understood, however, code has the advantage of being precise and succinct. Sometimes pseudo code, which is a mixture of code and English, can be helpful. Part of the art of describing an algorithm is knowing when to use what level of abstraction and which details to include.

Formal vs Informal: Students resist anything that is too abstract or too mathematical. However, math and logic are able to provide a precise and succinct clarity that neither code nor English can give. For example, some representations of algorithms like DFA and Turing Machines provide a formalism that is useful for specific applications or for proving theorems. On the other hand, personified analogies and metaphors help to provide both intuition and humor.

Value Simplicity: Abstract away the inessential features of a problem. Our goal is to understand and think about complex algorithms in simple ways. If you have seen an algorithm before, don't tune out when it is being explained. There are deep ideas within the simplicity.

Don't Describe Code: When told to describe an algorithm without giving code, do not describe the code in a paragraph, as in "We loop through i, being from 1 to n-1. For each of these loops, ..."

Incomplete Instructions: If an algorithm works either way, it is best not to specify it completely. For example, you may talk of selecting an item from a set without specifying which item is selected. If you then go on to prove that the algorithm works, you are effectively saying that the algorithm works for every possible way of choosing that value. This flexibility may be useful for the person who later implements the algorithm.

I recommend learning how to develop, think about, and explain algorithms within all of these different representations. Practice. Design your own algorithms, day dream about them, ask question about them, and perhaps the most important, explain them to other people.

2.2 Abstract Data Types (ADTs)

In addition to algorithms, we will develop abstractions of data objects.

Definition of an ADT: An abstract data type consists of a data structure, a list of operations that act upon the data, and a list of assertions that the ADT maintains.

Advantages: Abstract data types make it easier in the following ways to code, understand, design, and describe algorithms.

Abstract Objects: The description of an algorithm may talk of sets, sorted lists, queues, stacks, graphs, binary trees, and other such abstract objects without mentioning the data structure that handles them. In the end, computers represent these data structures as strings of zeros and ones; this does not mean that we have to.

Abstract Actions: It is also useful to group a number of steps into one abstract action, such as "sort a list", "pop a stack", or "find a path in a graph". It is easier to associate these actions with the object they are acting upon then with the algorithm that is using them.

Relationships between Objects: One can also talk abstractly about the relationships between objects with concepts such as paths, parent node, child node, and the like. The following two algorithmic steps are equivalent. Which do you find more intuitively understandable?

- if $A[i] \leq A[\lfloor i/2 \rfloor]$
- if the value of the node in the binary tree is at most that of its parent

Clear Specifications: Each abstract data type requires a clear specification so that the boss is clear about what he wants, the person implementing it knows what to code, users know how to use it, and testers knows what it is supposed to do.

Information Hiding: It is generally believed that global variables are bad form. If many routines in different parts of the system directly modify a variable in undocumented ways, then the program is hard to understand, modify, and debug. For this same reason, programmers who includes an ADT in their code are not allowed to access or modify the data structure except via the operations provided. This is referred to as *information hiding*.

Documentation: Using an abstract data type like a stack in your algorithm, automatically tells someone attempting to understand your algorithm a great deal about the purpose of this data structure.

Clean Boundaries: Using abstract data types breaks your project into smaller parts that are easy to understand and provides clean boundaries between these parts.

User: The clean boundaries allow one to understand and to use a data object without being concerned with how the object is implemented. The information hiding means that the user does not need to worry about accidently messing up the data structure in ways that are not allowed by its integrity constraints.

Implementer: The clean boundaries also allow someone else to implement the data object without knowing how it will be used. Given this, it also allows the implementation to be modified without unexpected effects to the rest of the code. Similarly, for abstract data types like graphs, a great literature of theorems have been proved. The clean boundaries around the abstract data type separates the task of proving that algorithm works from proving these theorems.

Code Reuse: Data structures like stack, sets, and graphs are used in many applications. By defining a general purpose abstract data type, the code, the understanding, and the mathematical theory developed for it can be refused over and over. In fact, an abstract data type can be used many times within the same algorithm by having many instances of it.

Optimizing: Having a limited set of operations guides the implementer to use techniques that are efficient for these operations yet may be slow for the operations excluded.

Examples of Abstract Data Types: The following are examples frequently used.

Simple Types: Integers, floating point numbers, strings, arrays, and records are abstract data types provided by all programming languages. The limited sets of operations that are allowed on each of these structures are well documented.

The List ADT: The concept of a list, eg. shopping list, is well known to everyone.

Data Structure: The data structure is a sequence of objects or elements. Often each element is a single data item. However, you can have a list of anything, even a list of lists. You can also have the empty list.

Invariants: Associated with the data items is an order which is determined by how the list was constructed. Unlike arrays, there are no empty positions in this ordering.

Operations: The i^{th} element in the list can be read and modified. A specific element can be searched for, returning its index. An element can be deleted or inserted at a specified index. This shifts the indices of the elements after it.

Uses: Most algorithms need to keep track of some unpredetermined number of elements. These can be stored in a list.

Running Times: In general, the user of an abstract data type does not need to know anything about its implementation. One useful thing to know, however, is how the different choices in implementation gives a tradeoff between the running times of different operations. For example, the elements in a list could be stored in an array. This allows the access of the i^{th} element in one programming step. Searching for an element takes $\Theta(n)$ time, which is reasonable. However, it is unfortunate that to add or delete an element in the middle of the list also takes $\Theta(n)$ time, because the other elements need to be shifted. This problem can be solved by linking the elements together into a linked list. However, then it takes $\Theta(n)$ time to walk the list to the i^{th} element.

The Set ADT: A set is basically a bag within which you can put any elements that you like.

Data Structure: It is the same as a list, except that the elements are not ordered. Again, the set can contain any types of element. Sets of sets is common. So is the empty set.

Invariants: The only invariant is knowing which elements are in the set and which are not.

Operations: Given an element and a set, one can determine whether or not the element is contained in the set. This is often called *a membership query*. One can ask for an arbitrary element from a set, determine the number of elements (size) of a set, iterate through all the elements, and add and delete elements.

Uses: Often an algorithm does not require its elements in a specific order. Using the set abstract data type instead of a list is useful to document this fact and to give more freedom to the implementer.

Running Times: Not being required to maintain an order of the elements, the set abstract data type can be implemented much more efficiently. One would think that searching for a element would take $\Theta(n)$ time as you compare it with each element in the set. However, if the elements are sorted then one can do this in $\Theta(\log n)$ time by doing a binary search. If the universe of possible elements is relatively small, then a good data structure is to have a boolean array indexed with each of these possible elements. An entry being true will indicate that the corresponding element is in the set. Surprisingly, even if the universe of elements is infinite, using a data structure called *Hash Tables*, all of these set operations can be done in constant time, i.e. independent of the number of items in the set. See Section 19.2.

The Set System ADT: A set system allows you to have a set (or list) of sets.

Operations: The additional operations of a system of sets are being able to form the *union*, *intersection*, or *subtraction* of two given sets and to create new sets. Also one can use an operation called *find* to determine which set a given element is contained in.

Running Times: Taking intersections and subtractions of two sets requires $\Theta(n)$ time. However, another quite surprising result is that on disjoint sets, the union and find operations can be done on average, for all practical purposes, in a constant amount of time. See Section 5.2.2.

Stack ADT: A stack is analogous to a stack of plates, in which a plate can be removed or added only at the top.

Data Structure: It is the same as a list, except that its set of operations is restricted.

Invariants: The order in which the elements were added to the stack is maintained. The end with the last element to be added is referred to as the *top* of the stack. Only this element can be accessed. This is referred to as *Last-In-First-Out* (LIFO).

Operations: A *push* is the operation of adding a new element to the top of the stack. A *pop* is the operation of removing and returning the top element from the stack. One can determine whether a stack is empty and what the top element is without popping it. However, the rest of the stack is hidden from view.

Uses: Stacks are the key data structure for recursion and parsing.

Running Times: Major advantage of restricting oneself to stack operations instead of using a general list is that both push and pop can be done in constant time, i.e. independent of the number of items in the stack. See Section 5.1.

The Queue ADT: A queue is analogous to a line-up for movie tickets; the first person to have arrived is the first person served. This is referred to as *First-In-First-Out* (FIFO).

Data Structure: Like a stack, this is the same as a list, except with a different restricted set of operations.

Invariants: An element enters the *rear* of the queue and when it is its turn, it leaves from the *front*.

Operations: The basic operations of a queue are to *add* an element to the rear and to *remove* the element that is at the front. Again, the rest of the queue is hidden from view.

Uses: An operating system will have a queue of jobs to run; a printer server, queue of jobs to print; a network hub, a queues of packets to transmit; and a simulation of a bank, a queue of customers.

Running Times: Queue operations can be also be done in constant time.

The Priority Queue ADT: In some queues, the more important elements are allowed to move to the front of the line.

Data Structure: Associated with each element is a priority.

Operations: When *inserting* an element, its priority must be specified. This priority can later be changed. When *deleting*, the element with the highest priority in the queue is removed and returned. Ties are broken arbitrarily.

Uses: Priority queues can be used instead of a queue when the priority scheme is needed.

Running Times: If the elements are stored unsorted in an array or linked list then an element can be added in $\Theta(1)$ time, but searching for the element with the next lowest priority will take $\Theta(n)$ time. If the data structure keeps the elements sorted by priority, then the next element can be found in $\Theta(1)$ time, but adding an element and maintaining the order will take $\Theta(n)$ time. If there are only a few different priority levels, than one can implement a priority queue by having separate queue for each possible priority level. Finding the next non-empty priority could then take $\Theta(\#\text{ of priorities})$. Other useful data structures for priority queues are *balanced binary search trees* (also called *AVL trees*) and *Heaps*. These can both insert and delete in $\Theta(\log n)$ time. See Sections 5.2.3 and 6.1.

The Dictionary ADT: A dictionary associates with each word a meaning. Similarly, a dictionary abstract data type associates data with each *key*.

Data Structure: A dictionary is similar to an array, except that each element is indexed by its key instead of by its integer location in the array.

Operations: Given a key one can *retrieve* the data associated with the key. One can also *insert* a new element into the dictionary by providing both the key and the data.

Uses: A name or social insurance number can be the key used to access all the relevant data about the person. Algorithms can also use such abstract data types to organize their data.

Running Time: The dictionary abstract data type can be implemented in the same way as for a set. Hence, all the operations can be done in constant time using hash tables.

The Graph ADT: A *graph* is an abstraction of a network of roads between cities. The cities are called *nodes* and the roads are called *edges*. The information stored is which pairs of nodes are connected by an edge. Though a drawing implicitly places each node at some location on the page, a key abstraction of a graph is that the location of a node is not specified. For example, the three pairs of graphs in Figure 2.1 are considered to be the same (isomorphic). In this way, a graph can be used to represent any set of relationships between pairs of objects from any fixed set of elements.

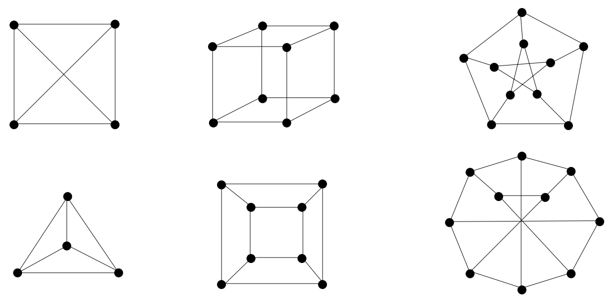


Figure 2.1: The top three graphs are famous: the complete graph on four nodes, the cube, and the peterson graph. The bottom three graphs of the same three graphs with their nodes layed out differently.

Exercise 2.2.1 *For each of the three pairs of graphs in Figure 2.1, number the nodes in such the way that $\langle i, j\rangle$ is an edge in one if and only if it is an edge in the other.*

Data Structure: A formal graph consists only of a set of nodes and a set of edges, where an node is simply a label (eg. $[1..n]$) and an edge is a pair of nodes. More generally, however, more data can be associated with each node or with each edge. For example, often each edge is associated with a number denoting its weight, cost, or length.

Notation:

Definitions	
$G = \langle V, E, \rangle$	A graph G is specified by a set of nodes V and a set of edges E.
vertex	Another name for a node
$u, v \in V$	Two nodes within the set of nodes
$\langle u, v\rangle \in E$	An edge within the set of edges
n and m	The number of nodes and edges
directed vs undirected	The edges of a graph can be either directed, drawn with an arrow and stored as an ordered pair of nodes or be undirected, drawn as a line and stored as a unordered set $\{u, v\}$ of two nodes.
adjacent	Nodes u and v are said to be adjacent if they have an edge between them.
Neighbors, $N(u)$	The neighbors of node u are those nodes that are adjacent to it.
degree, $d(v)$	The degree of a node is the number of neighbors that it has.
path	A path from u to v is a sequence of nodes (or edges) such that between each adjacent pair there is an edge.
simple	A path is simple if no node is visited more than once.
connected	A graph is connected if there is a path between every pair of nodes.
connected component	The nodes of an undirected graph can be partitioned based on which nodes are connected.
cycle	A cycle is a path from u back to itself.
acyclic	An acyclic graph is a graph that contains no cycles.
DAG	A DAG is a directed acyclic graph
tree	A tree is an undirected acyclic graph
complete graph	In a complete graph every pair of nodes has an edge.
dense vs sparse	A dense graph contains most of the possible edges and a sparse graph contains few of them.
singleton	a node with no neighbors

Operations: The basic operations are to determine whether an edge is in a graph, to add or delete an edge, and to iterate through the neighbors of a node. There is a huge literature of more complex operations that one might want to do. For example, one might want to determine which nodes have paths between them or to find the shortest path between two nodes. See Chapter 8.

Uses: The data from a surprisingly large number of computational problems can be organized as a graph. The problem, itself, can often be expressed as a well known graph theory problem.

Space and Time: A common data structure to store a graph is called an *adjacency matrix.* It consists of an $n \times n$ matrix with $M(u,v) = 1$ if $\langle u, v\rangle$ is an edge. An other is an *adjacency list* which lists for each node, the nodes adjacent to it. The first requires more space than the second when the graph is sparse, but on it operations can be executed faster.

The Tree ADT: Data is often organized into a hierarchy. A person has children, who have children of their own. The boss has people under him, who has people under them. The abstract data type for organizing this data is a *tree.*

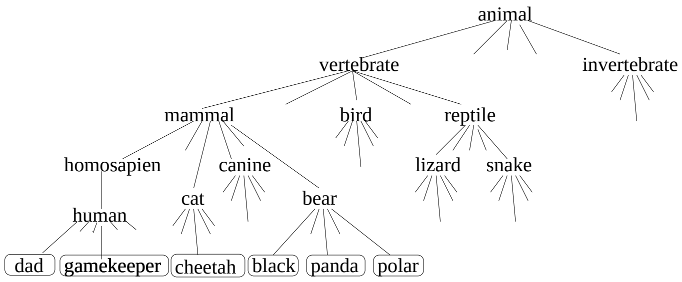


Figure 2.2: Classification Tree of Animals

Notation:

Definitions	
Root	Node at the top
Child	One of the nodes just under a node
Parent	The unique node immediately above a node
Sibling	The nodes with same parent
Ancestors	The nodes on the unique path from a node to the root
Descendants	All the nodes below a node
Leaf	A node with no children
Level of a node	The number of nodes in the unique path from the node to the root Some definitions say that the root is in level 0, others say level 1
Height of tree	The maximum level. Some definitions say that a node with a single node has height 0, others say height 1. It depends on whether you count nodes or edges.
Binary Tree	A binary tree is a tree in which each node has at most two children. Each of these children is designated as either the right child or as the left child.
tree.left	Left subtree of root
tree.right	Right subtree of root

Data Structure: The data structure must store for each node in the hierarchy the information associated with it, its list of children, and possibly (unless it is the root) its parent.

Operations: The basic operations are to retrieve the information about a node, determine which node is the parent of the node, and to loop through its children. One also can search for a specific node or to traverse through all the nodes in the tree in some specific order.

Uses: Often data that falls naturally into a hierarchy. For example, expressions like $3 \times 4 + 7 \times 2$ can be expressed as a tree. See Section 12.4. Also binary search trees and heaps are tree data structures that are alternatives to sorted data. See Sections 5.2.3 and 6.1.

Running Time: Given a pointer to a node, one can find its parent and children in constant time. Traversing the tree, clearly takes $\Theta(n)$ time.

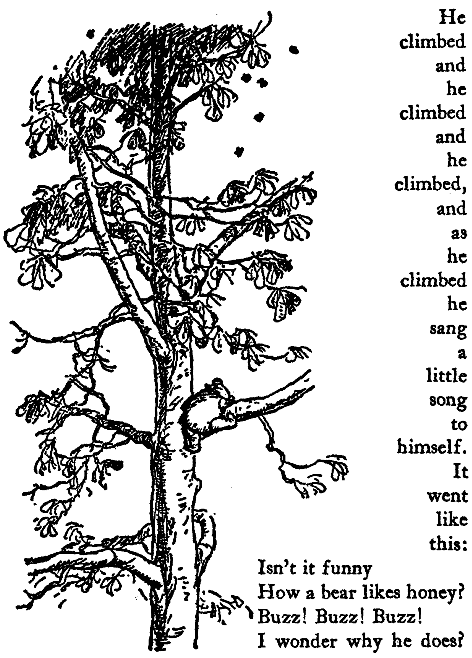
Part II

Loop Invariants for Iterative Algorithms

Chapter 3

Loop Invariants
The Techniques and the Theory

A technique used in many algorithms is to start at the being and take one step at a time towards the final destination. An algorithm that proceeds in this way is referred to as an *iterative* algorithm. Though this sounds simple, designing, understanding, and describing such an algorithm can still be a daunting task. It becomes less daunting if one does not need to worry about the entire journey at once but is able to focus separately on one step at a time.
The classic proverb advises one to focus first on the first step. This, however, is difficult before one knows where he is going. Instead, we advise that one first makes a general statement about the types of places that the computation might be during its algorithmic journey. Then, given any one such place, consider what single step the computation should take from there. This is the method of *assertions* and *loop invariants*.



3.1 Assertions and Invariants As Boundaries Between Parts

Whether you are designing an algorithm, coding an algorithm, trying to understand someone else's algorithm, describing algorithm to someone else, or formally proving that an algorithm works, you do not want to do it all at once. It is much easier to first break the algorithm into clear well defined pieces and then to separately design, code, understand, describe, or prove correct each of the pieces. Assertions provide clean boundaries between these parts. They state what the part can expect to have already been accomplished when part begins and what must be accomplished when it completes. Invariants are the same, except are apply either to a part like a loop that is executed many times or to a part like an object oriented a data structure that has an ongoing life.

Assertions As Boundaries Around A Part: An algorithm is broken into systems, subsystems, routines, and subroutines. You must be very clear about the goals of the overall algorithm and of each of these parts. Pre- and postconditions are assertions that provide a clean boundary around each of these.

Specifications: Any computational problem or sub-problem is defined as follows:

Preconditions: The preconditions state any assumptions that must be true about the input instance for the algorithm to operate correctly.

Postconditions: The postconditions are statements about the output that must be true when the algorithm returns.

Goal: An algorithm for the problem is correct if for *every* legal input instance, i.e. it meets the preconditions, the required output is produced, i.e. it meets the postconditions. On the other hand, if the input instance does not meet the preconditions, then all bets are off (but the program should be polite). Formally, we express this as $\langle pre-cond\rangle$ & $code_{alg} \Rightarrow \langle post-cond\rangle$.

Example: The problem *Sorting* is defined as:

Preconditions: The input is a list of n values with the same value possibly repeated.

Postconditions: The output is a list consisting of the same n values in non-decreasing order.

One Step At A Time: Worry only about your job.

Implementing: When you are writing a subroutine, you do not need to know how it will be used. You only need to make sure that it works correctly, namely if the input instance for the subroutine meets its preconditions then you must ensure that its output meets its postconditions.

Using: Similarly, when you are using the subroutine, you do not need to know how it was implemented. You only need to ensure that the input instance you give it meets the subroutine's preconditions. Then you can trust that the output meets its postconditions.

More Examples: When planning or describing a trip from my house to York University, you initially assume that initially the traveler is already at my house without even considering how he got there. In the end, he must be at York University. Both the author and the reader of a chapter or even of a paragraph need to have a clear description of what the author assumes the reader to know before reading the part and what he hopes the part will accomplish. A lemma needs a precise statement of what it proves, both so that the proof of the theorem can use it and so that it is clear what the proof of the lemma is supposed to proof.

Check Points: An assertion is a statement about the current state of the computation's data structures that is either true or false. It is made at some particular point during the execution of an algorithm. If it is false, then something has gone wrong in the logic of the algorithm.

Example: The task of getting from my home to York University is broken into stages. Assertions along the way are: "We are now at home." "We are now at Ossington Station." "We are now at St. George." "We are now at Downsview." "We are now at York."

Designing, Understanding, and Proving Correct: As described above, assertions are used to break the algorithm into parts. They provide check points along the path of the computation to allow everyone to know where the computation is and where it is going next. Generally, assertions are not tasks for the algorithm to perform, but are only comments that are added for the benefit of the reader.

Debugging: Some languages allow you to insert assertions as lines of code. If during the execution such an assertion is false, then the program automatically stops with a useful error message. This is very helpful when debugging your program. Even after the code is complete it useful to leave these checks in place. This way if something goes wrong, it is easier to determine why. This is what is occuring when an error box pops up during the execution of a program telling you to contact the vendor if the error persists. One difficulty with having the assertions be tested is that often the computation itself is unable to determine whether or not the assertion is true.

Invariants of Ongoing Systems: Algorithms for computational problems with pre and postconditions compute one function, taking an input at the beginning and stopping once the output has been produced. Other algorithms, however, are more dynamic. These are for systems or data structures that continue to receive a stream of information to which they must react. In an object oriented language, these are implemented with objects each of which has its own internal variables and tasks. A calculator example is presented in Section 7. The data structures described in Chapter 5 are other examples. Each such system has a set integrity constraints that must be maintained. These are basically assertions that must be true every time the system is entered or left. We refer to them as *invariants.* They come in two forms.

Public Specifications: Each specification of a system has a number of invariants that any outside user of the system needs to know about so that he can use the system correctly. For example, a user should be assured that his bank account will always correctly reflects the amount of money that he has. He should also know what happens when this amount goes negative. Similarly, a user of a stack should know that when he pops it, the last object that he pushed will be returned.

Hidden Invariants: Each implementation of a system has a number of invariants that only the system's designers need to know about and maintain. For example, a stack may be implemented using a linked list which always maintains a pointer to the first object.

Loop Invariants: A special type of assertions consist of those that are placed at the top of a loop. They are referred to as *loop invariants*, because they must hold true every time the computation returns to the top of the loop.

Algorithmic Technique: Loop invariant are the focus of a major part of this text because they form the basis of the algorithmic technique referred to as *iterative algorithms.*

3.2 An Introduction to Iterative Algorithms

An algorithm that consists of small steps implemented by a main loop is referred to an *iterative algorithm.* Understanding what happens within a loop can be surprisingly difficult. Formally, one proves that an iterative algorithm works correctly using loop invariants. (See Section 3.4.2.) I believe that this concept can also be used as *a level of abstraction* within which to design, understand, and explain iterative algorithms.

3.2.1 Various Abstractions and the Main Steps

Iterative Algorithms: A good way to structure many computer programs is to store the key information you currently know in some data representation and then each iteration of the main loop takes a step towards your destination by making a simple change to this data.

Loop Invariants: A loop invariant is an assertion that must be true about the state of this data structure at the start of every iteration and when the loop terminates. It expresses important relationships among the variables and in doing so expresses the progress that has been made towards the postcondition.

Analogies: The following are three anolgies of these ideas.

One Step At A Time: The Buddhist way is not to have cravings or aversions about the past or the future, but to be in the here and now. Though you do not want to have a fixed predetermined fantasy about your goal, you do need to have some guiding principles to point you more or less in the correct direction. Meditate until you understand the key aspects of your current situation. Trust that your current simple needs are met. Rome was not built in a day. Your current task is only to make some small simple step. With this step, you must both ensure that your simple needs are met tomorrow and that you make some kind of progress towards your final destination. Don't worry. Be happy.

Steps From A Safe Location To A Safe Location: The algorithm's attempt to get from the preconditions to the postconditions is like being dropped off in a strange city and weaving a path to some required destination. The current *state* or location of the computation is determined by values of all the variables. Instead of worrying about the entire computation, take one step at a time. Given the fact that your single algorithm must work for an infinite number of input instances and given all the obstacles that you pass along the way, it can be difficult to predict where computation might be in the middle of the computation. Hence, for every possible location that the computation might be, the algorithm attempts to define what step to take next. By ensuring that each such step will make some progress towards the destination, the algorithm can ensure that the computation does not wondering aimlessly, but weaves and spirals in towards the destination. The problem with this approach is that there are far too many locations that the computation may be in and some of these may be very confusing. Hence, the algorithm uses *loop invariants* to define a set of safe locations. A loop invariant can be thought of as the assertion that the computation is currently in such a safe location. For example, one assumption might be that during our trip we do not end up in a ditch or up in a tree. The algorithm then defines how to take a step from each such safe location. This step now has two requirements. It must make progress and it must not go from a safe location to an unsafe location. Assuming that initially the computation is in a safe location, this ensures that it remains in a safe location. This ensures that the next step is always defined. Assuming that initially the computation is not infinitely far from its destination, making progress each step ensures that eventually the computation will have made enough progress that the computation stops. Finally, the design of the algorithm must ensure that being in a safe location with this much progress made is enough to ensure that the final destination is reached. This completes the design of the algorithm.

A Relay Race: The loop can be viewed as a relay race. Your task is not to run the entire race. You take the baton from a friend. Though in the back of your mind you know that the baton has traveled many times around the track already, this fact does not concern you. You have been assured that the baton meets the conditions of the *loop invariants*. Your task is to carry the baton once around the track. You must make progress, and you must ensure that the baton still meets the conditions after you have taken it around the track, i.e., that the loop invariants have been maintained. Then, you hand the baton on to another friend. This is the end of your job. You do not worry about how it continues to circle the track. The difficulty is that you must be able to handle *any* baton that meets the loop invariants.

Structure of An Iterative Algorithm:

begin routine
    $\langle pre-cond \rangle$
    $code_{pre\text{-}loop}$ % Establish loop invariant
    loop
        $\langle loop-invariant \rangle$
        exit when $\langle exit-cond \rangle$
        $code_{loop}$ % Make progress while maintaining the loop invariant
    end loop
    $code_{post\text{-}loop}$ % Clean up loose ends
    $\langle post-cond \rangle$
end routine

The Most Important Steps: The most important steps when developing an iterative algorithm within the loop invariant level of abstraction are the following:

The Steps:

- What invariant is maintained?
- How is this invariant initially obtained?
- How is progress made while maintaining the invariant?
- How does the exit condition together with the invariant ensure that the problem is solved?

Induction Justification: Section 3.4.2 uses induction to prove that if the loop invariant is initially established and is maintained then it will be true at the beginning of each iteration. Then in the end, this loop invariant is used to prove that problem is solved.

Faith in the Method: Instead of rethinking difficult things every day, it is better to have some general principles with which to work. For example, every time you walk into a store, you do not want to be rethinking the issue of whether or not you should steal. Similarly, everytime you consider a hard algorithm, you do not want to be rethinking the issue of whether or not you believe in the loop invarient method. Understanding the algorithm itself will be hard enough. Hence, while reading this chapter you should once and for all come to understand and believe to the depth of your soul how the above mentioned steps are sufficient to describing an algorithm. Doing this can be difficult. It requires a whole new way of looking at algorithms. However, at least for the duration of this course, adopt this as something that you believe in.

3.2.2 Examples: Quadratic Sorts

The algorithms Selection Sort and Insertion Sort are classic examples of iterative algorithms. Though you have likely seen them before, use them to understand these required steps.

Selection Sort: We maintain that the k smallest of the elements are sorted in a list. The larger elements are in a set on the side. Progress is made by finding the smallest element in the remaining set of large elements and adding this selected element at the end of the sorted list of elements. This increases k by one. Initially, with $k = 0$, all the elements are set aside. Stop when $k = n$. At this point, all the elements have been selected and the list is sorted.

If the input is presented as an array of values, then sorting can happen in place. The first k entries of the array store the sorted sublist, while the remaining entries store the set of values that are on the side. Finding the smallest value from $A[k+1] \ldots A[n]$ simply involves scanning the list for it. Once it is found, moving it to the end of the sorted list involves only swapping it with the value at $A[k+1]$. The fact that the value $A[k+1]$ is moved to an arbitrary place in the right-hand side of the array is not a problem, because these values are considered to be an unsorted set anyway.

Runnin Time: We must select n times. Selecting from a sublist of size i takes $\Theta(i)$ time. Hence, the total time is $\Theta(n + (n-1) + ... + 2 + 1) = \Theta(n^2)$.

Insertion Sort: We maintain a subset of elements sorted within a list. The remaining elements are off to the side somewhere. Progress is made by taking one of the elements that is off to the side and *inserting* it into the sorted list where it belongs. This gives a sorted list that is one element longer than it was before. Initially, think of the first element in the array as a sorted list of length one. When the last element has been inserted, the array is completely sorted.

There are two steps involved in inserting an element into a sorted list. The most obvious step is to locate where it belongs. The second step to shift all the elements that are bigger than the new element one to the right to make room for it. You can find the location for the new element using a binary search. However, it is easier to search and shift the larger elements simultaneously.

Runnin Time: We must insert n times. Inserting into a sublist of size i takes $\Theta(i)$ time. Hence, the total time is $\Theta(1 + 2 + 3 + ... + n) = \Theta(n^2)$.

For $i = 1..n$: One might think that one does not need a loop invariant when accessing each element of an array. Using the statement "for $i = 1..n$ do", one probably does not. However, code like "i=1 while($i \leq n$) $A[i] = 0$; $i = i + 1$; end while" is surprisingly prone without a loop invariant to the error of being off by one. The loop invariant is that when at the top of the loop, i indexes the next element to zero.

3.3 The Steps In Developing An Iterative Algorithm

This section presents the more detailed steps that I recommend using when developing an iterative algorithm.

3.3.1 The Steps

Perliminaries: Before we can design an iterative algorithm, we need to know precisely what it is supposed to do and have some idea about the kinds of steps it will take.

1) Specifications: Carefully write the specifications for the problem.

Preconditions: What are the legal instances (inputs) for which the algorithm should work?

Postconditions: What is the required output for each legal instance?

2) Basic Steps: As a preliminary to designing the algorithm it can be helpful to consider what basic steps or operations might be performed in order to make progress towards solving this problem. Take a few of these steps on a simple input instance in order to get some intuition as to where the computation might go. How might the information gained narrow down the computation problem?

A Middle Iteration on a General Instance: Though the algorithm that you are designing needs to work correctly on each possible input instance, do not start by considering special case input instances. Instead, consider a large and general instance. If there are a number of different types of instances, consider the one that seem to be the most general.

Being an iterative algorithm, this algorithm will execute its main loop many times. Each of these is called an *iteration*. Each iteration must work correctly and must connect well with the previous and the next iterations. Do not start designing the algorithm by considering the steps before the loop or by considering the first iteration. In order to see the big picture of the algorithm more clearly, jump into the middle of the computation. From there, design the main loop of your algorithm to work correctly on a single iteration. The steps to do this are as follows.

3) Loop Invariant: Describe what you would like the data structure to look like when the computation is at the beginning of this middle iteration.

Draw a Picture: Your description should leave your reader with a visual image. Draw a picture if you like.

Don't Be Frightened: A loop invariant should not consist of formal mathematical mumble jumble if an informal description would get the idea across better. On the other hand, English is sometimes misleading and hence a more mathematical language sometimes helps. Say things twice if necessary. In general, I recommend pretending that you are describing the algorithm to a first year student.

Safe Place: A loop invariant must ensure that the computation is still within a safe place along the road and has not fallen into a ditch or landed in a tree.

The Work Completed: The loop invariant must characterize what work has been completed towards solving the problem and what work still needs to be done.

4) Measure of Progress: At each iteration, the computation must make progress. You, however, have complete freedom do decide how this progress is measured. A significant step in the design of an algorithm is defining this *measure* according to which you gauge the amount of progress that the computation has made.

5) Main Steps: You are now ready to make your first guess as to what the algorithmic steps within the main loop will be. When doing this, recall that your only goal is to make progress while maintaining the loop invariant.

6) Maintaining the Loop Invariant: To check whether you have succeeded in the last step, you must prove that the loop invariant is maintained by these steps that you have put within the loop.

Again, assume that you are in the middle of the computation at the top of the loop. Your loop invariant describes the state of the data structure. Refer back to the picture that you drew. Execute one iteration of the loop. You must prove that when you get back to the top of the loop again, the requirements set by the loop invariant are met once more.

7) Making Progress: You must also prove that progress of at least one (according to your measure) is made every time the algorithm goes around the loop.

Sometimes, according to the most obvious measure of progress, the algorithm can iterate around the loop without making any measurable progress. This is not acceptable. The danger is that the algorithm will loop forever. In this case, you must define another measure that better shows how you are making progress during such iterations.

Beginning & Ending: Once you have passed through steps 2-7 enough times that you get them to work smoothy together, you can consider beginning the first iteration and ending the last iteration.

8) Initial Conditions: Now that you have an idea of where you are going, you have a better idea about how to begin. In this step, you must develop the initiating pseudo code for the algorithm. Your only task is to initially establish the loop invariant. Do it in the easiest way possible. For example, if you need to construct a set such that all the dragons within it are purple, the easiest way to do it is to construct the empty set. Note that all the dragons in this set are purple, because it contains no dragons that are not purple.

Careful. Sometimes it is difficult to know how to initially set the variable to make the loop invariant initially true. In such cases, try setting them to ensure that it is true after the first iteration. For example, what is the maximum value within an empty list of values? One might think 0 or ∞. However, a better answer is $-\infty$. When adding a new value, one uses the code $newMax = \max(oldMax, newValue)$. Starting with $oldMax = -\infty$, gives the correct answer when the first value is added.

9) Exit Condition: The next thing that you must design in your algorithm is the condition that causes the computation to break out of the loop. The following are two ways of initially guessing what a good exit condition might be.

Sufficient Progress: Ideally, when designing the exit condition, you have some insight into how much progress the algorithm must make before it is able to solve the problem. This then will be the exit condition.

Stuck: Sometimes, however, though your intuition is that your algorithm designed so far is making progress each iteration, you have no clue whether heading in this direction the algorithm will ever solve the problem or how you would know it if it happened. One way to obtain an initial guess of what the exit condition might be is to have your algorithm exit when ever it gets stuck, i.e. conditions in which the algorithm is unable to execute its main loop and make progress. For such situations, you must either think of other ways for your algorithm to make progress or have it exit. A good first step is to exit. In step 11 below, you will have to prove that when your algorithm exits in this way that you actually are able to solve the problem. If you are unable to do this, then you will have to go back and redesign your algorithm.

Loop While vs Exit When: As an aside, note that the following are equivalent:

while(A and B)
    ....
end while
	loop
	    $\langle loop{-}invariant \rangle$
	    exit when (not A or not B)
	    ...
	end loop

The second is more useful here because it focuses on the conditions needed to exit the loop, while the first focuses on the conditions needed to continue. A secondary advantage of the second is that it also allows you to slip in the loop invariant between the top of the loop and the exit condition.

10) Termination and Running Time: In this next step, you must ensure that the algorithm does not loop forever. To do this, you do not need to precisely determine the exact point at which the exit condition will be met. Instead, state some upper bound (perhaps the size of the input n) and prove that if this much progress has been made, then the exit condition has definitely been met. If it exits earlier then this, all the better.

There is no point in wasting a lot of time developing an algorithm that does not run fast enough to meet your needs. Hence, it is worth estimating at this point the running time of the algorithm. Generally, this done by dividing the total progress required by the amount of progress made each iteration.

11) Ending: Now that you have an idea of where you will be in the middle of your computation, you know from which direction you will be heading when you finally arrive at the destination. In this step, you must ensure that once the loop has exited that you will be able to solve the problem. To help you with this task, you know two things about the state your data structures will be in at this point in time. First, you know that the loop invariant will be true. You know this because the loop invariant must always be true. Second, you know that the exit condition is true. This you know, by the fact that the loop has exited. From these two facts and these facts alone, you must be able to deduce that with only a few last touches the problem can be solved. Develop the pseudo code needed after the loop to complete these last steps.

12) Testing: Try your algorithm by hand on a couple of examples.

Fitting the Pieces Together: The above steps complete all the parts of the algorithm. Though these steps were listed as independent steps and in fact can be completed in any order, there is an interdependence between them. In fact, you really cannot complete any one step until you have an understanding of all of them. Sometimes after completing the steps, things do not quite fit together. It is then necessary to cycle through the steps again using your new understanding. I recommend cycling through the steps a number of times. The following are more ideas to use as you cycle through these steps.

Flow Smoothly: The loop invariant should flow smoothly from the beginning to the end of the algorithm.

- At the beginning, it should follow easily from the preconditions.
- It should progress in small natural steps.
- Once the exit condition has been met, the postconditions should easily follow.

Ask for 100%: A good philosophy in life is to ask for 100% of what you want, but not to assume that you will get it.

Dream: Do not be shy. What would you like to be true in the middle of your computation? This may be a reasonable loop invariant, or it may not be.

Pretend: Pretend that a genie has granted your wish. You are now in the middle of your computation and your dream loop invariant is true.

Maintain the Loop Invariant: From here, are you able to take some computational steps that will make progress while maintaining the loop invariant? If so, great. If not, there are two common reasons.

Too Weak: If your loop invariant is too weak, then the genie has not provided you with everything you need to move on.

Too Strong: If your loop invariant is too strong, then you will not be able to establish it initially or maintain it.

No Unstated Assumptions: Often students give loop invariants that lack detail or are too weak to proceed to the next step. Don't make assumptions that you don't state. As a check pretend that you are a Martian who has jumped into the top of the loop knowing *nothing* that is not stated in the loop invariant.

13) Special Cases: In the above steps, you were considering one general type of large input instances. If there is a type of input that you have not considered repeat the steps considering them. There may also be special cases interations that need to be considered. These are likely occure near the beginning or the end of the algorithm. Continue repeating the steps until you have considered all of the special cases input instances.

Though these special cases may require separate code, start by tracing out what the algorithm that you have already designed would do given such an input. Often this algorithm will just happen to handle a lot of these cases automatically without requiring separate code. When ever adding code to handle a special case, be sure to check that the previously handled cases still are handled.

14) Coding and Implementation Details: Now you are ready to put all the pieces together and produce pseudo code for the algorithm. It may be necessary at this point to provide extra implementation details.

15) Formal Proof: After the above steps seem to fit well together, you should cycle one last time through them being particularly careful that you have met all the formal requirements needed to prove that the algorithm works. These requirements are as follows.

6') Maintaining Loop Invariant (Revisited): Many subtleties can arise given the huge number of different input instances and the huge number of different places the computation might get to. In this step, you must double check that you have caught all of these subtleties. To do this, you must ensure that the loop invariant is maintained when the iteration starts in *any* of the possible places that the computation might be in. This is particularly important for those complex algorithms for which you have little grasp of where the computation will go.

Proof Technique: To prove this, pretend that you are at the top of the loop. It does not matter how you got there. As said, you may have dropped in from Mars. You can assume that the loop invariant is satisfied, because your task here is only to maintain the loop invariant. You can also assume that the exit condition is not satisfied, because otherwise the loop would exit at this point and there would be no need to maintain the loop invariant during an iteration. However, besides these two things you know nothing about the state of the data structure. Make no other assumptions. Execute one iteration of the loop. You must then be able to prove that when you get back to the top of the loop again, the requirements set by the loop invariant are met once more.

Differentiating between Iterations: The statement "$x = x + 2$" is meaningful to a computer scientist as a line of code. However, to a mathematician it is a statement that is false unless you are working over the integers modulo 2. We do not want to see such statements in mathematical proofs. Hence, it is useful to have a way to differentiate between the values of the variables at the beginning of the iteration and the new values after going around the loop one more time. One notation is to denote the former with x' and the latter with x''. (One could also use x_i and x_{i+1}.) Similarly, $\langle loop{-}invariant' \rangle$ can be used to state that the loop invariant is true for the x' values and $\langle loop{-}invariant'' \rangle$ that it is true for the x'' values.

The Formal Statement: Whether or not you want to prove it formally, the formal statement that must be true is $\langle loop{-}invariant' \rangle$ & not $\langle exit{-}cond \rangle$ & $code_{loop} \Rightarrow \langle loop{-}invariant'' \rangle$.

The Formal Proof Technique: Assume that

- $\langle loop{-}invariant' \rangle$
- not $\langle exit{-}cond \rangle$
- the effect of code B (e.g., $x'' = x' + 2$)

and then prove from these assumptions the conclusion that $\langle loop{-}invariant'' \rangle$.

7') Making Progress (Revisited): These special cases may also affect whether progress is made during each iteration. Again, assume nothing about the state of the data structure except that the loop invariant is satisfied and the exit condition is not. Execute one iteration of the loop. Prove that significant progress has been made according to your measure.

8') Initial Conditions (Revisited): You must ensure that the initial code establishes the loop invariant. The difficulty, however, is that you must ensure that this happens not matter what the input instance is. When the algorithm begins, the one thing that you can assume is the preconditions are true. You can assume this because, if it happens that they are not true then you are not expected to solve the problem.

The formal statement that must be true is $\langle pre{-}cond \rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop{-}invariant \rangle$.

11') Ending: In this step, you must ensure that once the loop has exited that the loop invariant (which you know will always be true) and the exit condition (which caused the loop to exit) together will give you enough so that your last few touches solves the problem. The formal statement that must be true is $\langle loop-invariant\rangle$ & $\langle exit-cond\rangle$ & $code_{post\text{-}loop} \Rightarrow \langle post-cond\rangle$.

10') Running Time: The task of designing an algorithm is not complete until you have determined its running time. Your measure of progress will be helpful when bounding the number of iterations that get executed. You must also consider the amount of work per iteration. Sometimes each iteration requires a different amount of work. In this case, you will need to approximate the sum.

12') Testing: Try your algorithm by hand on more examples. Consider both general input instances and special cases. You may also want to code it up and run it.

This completes the steps for developing an iterative algorithm. Likely you will find that coming up with the loop invariant is the hardest part of designing an algorithm. It requires practice, perseverance, and insight. However, from it, the rest of the algorithm follows easily with little extra work. Here are a few more pointers that should help design loop invariants.

A Starry Night: How did Van Gogh come up with his famous painting, A Starry Night? There's no easy answer. In the same way, coming up with loop invariants and algorithms is an art form.

Use This Process: Don't come up with the loop invariant after the fact to make me happy. Use it to design your algorithm.

Know What an LI Is: Be clear about what a loop invariant is. On midterms, many students write, "the LI is ..." and then give code, a precondition, a postcondition, or some other inappropriate piece of information. For example, stating something that is ALWAYS true, such as $1 + 1 = 2$ or "The root is the max of any heap", may be useful information for the answer to the problem, but should not be a part of the loop invariant.

3.3.2 Example: Binary Search

In a study, a group of experienced programmers were asked to code binary search. Easy, yes? 80% got it wrong!! My guess is that if they had used loop invariants, they all would have got it correct.

Exercise 3.3.1 *Before reading this section, try writing code for binary search without the use of loop invariants. Then think about loop invariants and try it again. Finally, read this section. Which of these algorithms work correctly?*

I will use the detailed steps given above to develop a binary search algorithm. Formal proofs can seem tedious when the result seems to be intuitively obvious. However, you need to know how to do these tedious proofs – not so you can do it for every loop you write, but to develop your intuition about where bugs may lie.

1) Specifications:

Preconditions: An input instance consists of a sorted list $L(1..n)$ of elements and a key to be searched for. Elements may be repeated. The key may or may not appear in the list.

Postconditions: If the key is in the list, then the output consists of an index i such that $L(i) = key$. If the key is not in the list, then the output reports this.

2) Basic Steps: The basic step compares the key with the element at the center of the sublist. This tells you which half of the sublist the key is in. Keep only the appropriate half.

3) Loop Invariant: The algorithm maintains a sublist that contains the key. English sometimes being misleading, a more mathematical statement might be "If the key is contained in the original list, then the key is contained in the sublist $L(i..j)$." Another way of saying this is that we have determined that the key is not within $L(1..i-1)$ or $L(j+1..n)$. It is also worth being clear, as we have done with the notation $L(i..j)$, that the sublist includes the end points i and j. Confusions in details like this are the cause of many bugs.

4) Measure of Progress: The obvious measure of progress is the number of elements in our sublist, namely $j - i + 1$.

5) Main Steps: As I said, each iteration compares the key with the element at the center of the sublist. This determines which half of the sublist the key is in and hence which half to keep. It sounds easy, however, there are a few minor decisions to make which may or may not have an impact.

1. If there are an even number of elements in the sublist, do we take the "middle" to be the element slightly to the left of the true middle or the one slightly to the right?
2. Do we compare the key with the middle element using $key < L(mid)$ or $key \leq L(mid)$?
3. Should we also check whether or not $key = L(mid)$?
4. When splitting the sublist $L(i..j)$ into two halves do we include the middle in the left half or in the right half?

Decisions like these have the following different types of impact on the correctness of the algorithm.

Does Not Matter: If it really does not make a difference which choice you make and you think that the implementer may benefit from having this flexibility, then document this fact.

Consistent: Often it does not matter which choice is made, but bugs can be introduced if you are not consistent and clear as to which choice has been made.

Matters: Sometimes these subtle points matter.

Interdependence: In this case, we will see that each choice can be made either way, but some combinations of choices work and some do not.

When in the first stages of designing an algorithm, I usually do something between flipping a coin and sticking to this decision until a bug arises and attempting to keep all of the possibilities open until I see reasons that it matters.

6) Maintaining the Loop Invariant: You must prove that $\langle loop{-}invariant' \rangle$ & *not* $\langle exit{-}cond \rangle$ & $code_{loop} \Rightarrow \langle loop{-}invariant'' \rangle$.

If the key is not in the original list, then the statement of the loop invariant is trivially true. Hence, let us assume for now that we have a reasonably large sublist $L(i..j)$ containing the key. Consider the following three cases:

- Suppose that key is strictly less than $L(mid)$. Because the list is sorted, $L(mid) \leq L(mid+1) \leq L(mid+2) \leq \ldots$. Combining these fact tells us that the key is not contained in $L(mid, mid+1, mid+2, \ldots)$ and hence it must be contained in $L(i'..mid{-}1)$.
- The case in which key is strictly more than $L(mid)$ is similar.
- Care needs to be taken when key is equal to $L(mid)$. One option, as has been mentioned, is to test for this case separately. Though finding the key in this way would allow you to stop early, extensive testing shows that this extra comparison slows down the computation. The danger of not testing for it, however, is that we may skip over the key by including the middle in the wrong half. If the test $key < L(mid)$ is used, the test will fail when key and $L(mid)$ are equal. Thinking that the key is bigger, the algorithm will keep the right half of the sublist. Hence, the middle element should be included in this half, namely, the sublist $L(i..j)$ should be split into $L(i..mid{-}1)$ and $L(mid..j)$. Conversely, if $key \leq L(mid)$ is used, the test will pass and the left half will be kept. Hence, the sublist should be split into $L(i..mid)$ and $L(mid{+}1..j)$.

7) Making Progress: You be sure to make progress with each iteration, i.e., the sublist must get strictly smaller. Clearly, if the sublist is large and you throw away roughly half of it at every iteration, then it gets smaller. We take note to be careful when the list becomes small.

8) Initial Conditions: Initially, you obtain the loop invariant by considering the entire list as the sublist. It trivially follows that if the key is in the entire list, then it is also in this sublist.

9) Exit Condition: One might argue that a sublist containing only one element, cannot get any smaller and hence the program must stop. The exit condition would then be defined to be $i = j$. In general having such specific exit condition is prone to bugs, because if by mistake j were to become less then i, then the computation might loop for ever. Hence, we will use the more general exit condition like $j \leq i$. The bug in the above argument is that sublists of size one can be made smaller. You need to consider the possibility of the empty list.

10) Termination and Running Time: Initially, our measure of progress is the size of the input list. The assumption is that this is finite. Step 10 proves that this measure decreases each iteration by at least one. In fact, progress is made fast. Hence, this measure will quickly be less or equal to one. At this point, the exit condition will be met and the loop will exit.

11) Ending: We must now prove that, with the loop invariant and the exit condition together, the problem can be solved, namely that $\langle loop\!-\!invariant\rangle$ & $\langle exit\!-\!cond\rangle$ & $code_{post\text{-}loop} \Rightarrow \langle post\!-\!cond\rangle$.

The exit condition gives that $j \leq i$. If $i = j$, then this final sublist contains one element. If $j = i - 1$, then it is empty. If $j < i - 1$, then something is wrong. Assuming that the sublist is $L(i = j)$, reasonable final code would be to test whether the key is equal to this element $L(i)$. (This is our first test for equality.) If they are equal, the index i is returned. If they are not equal, then we claim that the key is not in the list. On the other hand, if the final sublist is empty then we will simply claim that the key is not in the list. We must now prove that this code ensures that the postcondition has been met.

By the loop invariant, we know that "If the key is contained in the original list, then the key is contained in the sublist $L(i..j)$." Hence, if the key is contained in the original list, then $L(i..j)$ cannot be empty and the program will find it in the one location $L(i)$ and give the correct answer. On the other hand, if it is not in the list, then it is definitely not in $L(i)$ and the program will correctly state that it is not in the list.

12) Testing: Try a few examples.

13: Special Cases:

6) Maintaining Loop Invariant (Revisited): When proving that the loop invariant was maintained, we assumed that we still had a large sublist for which $i < mid < j$ were three distinct indexes. When the sublist has three elements, this is still the case. However, when the sublist has two elements, the "middle" element must be either the first or the last. This may add some subtleties. I will leave it as an exercise for you to check that the three cases considered in the original "Maintaining Loop Invariant" part still hold.

7) Making Progress (Revisited): You must also make sure that you continue to make progress when the sublist becomes small. A sublist of three elements divides into one list of one element and one list of two elements. Hence, progress has been made. Be careful, however, with sublists of two elements.

The following is a common bug: Suppose that we decide to consider the element just to the left of center as being the middle and we decide that the middle element should be included in the right half. Then given the sublist $L(3, 4)$, the middle will be element indexed with 3 and the right sublist will be still be $L(mid..j) = L(3, 4)$. If this sublist is kept, no progress will be made and the algorithm will loop forever. We ensure that this sublist is cut in half either by having the middle be the one to the left and including it on the left or the one to the right and including it on the right. As seen, in the first case, we must use the test $key \leq L(mid)$ and in the second case the test $key < L(mid)$. Typically, the algorithm is coded using the first option.

14) Implementation Details:

Math Details: Small math operations like computing the index of the middle element are prone to bugs. (Try it on your own.) The exact middle is the average between i and j, namely $\frac{i+j}{2}$. If you want the integer slightly to the left of this, then you round down. Hence, $mid = \lfloor \frac{i+j}{2} \rfloor$. It is a good idea to test these operations on small examples.

Won't Happen: We might note that the sublist never becomes empty, assuming that is that the initial list is not empty. (Check this yourself.) Hence, the code does not need to consider this case.

Code: Putting all these pieces together gives the following code.

```
algorithm BinarySearch (⟨L(1..n), key⟩)
⟨pre−cond⟩: ⟨L(1..n), key⟩ is a sorted list and key is an element.
⟨post−cond⟩: If the key is in the list, then the output consists of an index i such that L(i) = key.

begin
    i = 1, j = n
    loop
        ⟨loop−invariant⟩: If the key is contained in L(1..n), then the key is contained
            in the sublist L(i..j).
        exit when j ≤ i
        mid = ⌊(i+j)/2⌋
        if(key ≤ L(mid)) then
            j = mid          % Sublist changed from L(i, j) to L(i..mid)
        else
            i = mid + 1      % Sublist changed from L(i, j) to L(mid+1, j)
        end if
    end loop
    if(key = L(i)) then
        return( i )
    else
        return( "key is not in list" )
    end if
end algorithm
```

15) Formal Proof: Now that we have fixed the algorithm, all the proofs should be checked again. The following proof is a more formal presentation of that given before.

6') Maintaining the Loop Invariant: We must prove that $\langle loop\!-\!invariant' \rangle$ & *not* $\langle exit\!-\!cond \rangle$ & $code_{loop} \Rightarrow \langle loop\!-\!invariant'' \rangle$. Assume that the computaion is at the top of the loop, the loop invariant is true, and the exit condition is not true. Then, let the computation go around the loop one more time. When it returns to the top, you must prove that the loop invariant is again true. To distinguish between these states let i' and j' denote the values of the variables i and j before executing the loop and let i'' and j'' denote these values afterwards. By $\langle loop\!-\!invariant' \rangle$, you know that "If the key is contained in the original list, then the key is contained in the sublist $L(i'..j')$." By *not* $\langle exit\!-\!cond \rangle$, we know that $i' < j'$. From these assumptions, you must prove $\langle loop\!-\!invariant'' \rangle$, namely that "If the key is contained in the original list, then the key is contained in the sublist $L(i''..j'')$." If the key is not in the original list, then the statement $\langle loop\!-\!invariant'' \rangle$ is trivially true, so assume that the key is in the original list. By $\langle loop\!-\!invariant' \rangle$, it follows that the key is in $L(i'..j')$. There are three cases to consider:

- Suppose that key is strictly less than $L(mid)$. The comparison $key \leq L(mid)$ will pass, j'' will be set to mid, i'' by default will be i', and the new sublist will be $L(i'..mid)$. Because the list is sorted, $L(mid) \leq L(mid+1) \leq L(mid+2) \leq \ldots$. Combining this fact with the assumption that $key < L(mid)$ tells you that the key is not contained in $L(mid, mid+1, mid+2, ...)$. You know that the key is contained in $L(i'..., mid, ...j')$. Hence, it must be contained in $L(i'..mid-1)$, which means that it is also contained in $L(i''..j'') = L(i'..mid)$ as required.

- Suppose that key is strictly more than $L(mid)$. The test will fail. The argument that the key is contained in the new sublist $L(mid+1..j')$ is the same as that above, replacing $+$ with $-$.
- Finally, suppose that key is equal to $L(mid)$. The test will pass and hence the new sublist will be $L(i'..mid)$. According to our assumption that $key = L(mid)$, the key is trivially contained in this new sublist.

7', 8', 11' Making these steps more formal will be left as an exercise.

10') Running Time: The sizes of the sublists are approximately $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, ..., 8, 4, 2, 1$. Hence, only $\Theta(\log n)$ splits are needed. Each split takes $O(1)$ time. Hence, the total time is $\Theta(\log n)$.

12') Testing: Most of the testing I will leave as an exercise for you. One test case to definitely consider is the input instance in which the initial list to be searched in is empty. In this case, the sublist would be $L(1..n)$ with $n = -1$. Tracing the code, everything looks fine. The loop breaks before iterating. The key wont be found and the correct answer is returned. On closer examination, however, note that the if statement accesses the element $L(1)$. If the array L is allocated zero elements, then this may cause a run time error. In a language like C, the program will simply access this memory cell, even though it may not belong to the array and perhaps not even to the program. Such assesses might not cause a bug for years. However, what if the memory cell $L(1)$ happens to contain the key. Then the program returns that the key is contained at index 1. This would be the wrong answer.

To recap, the goal of a binary search is to find a key within a sorted list. We maintain a sublist that contains the key (if the key is in the original list). Originally, the sublist consists of the entire list. Each iteration compares the key with the element at the center of the sublist. This tells us which half of the sublist the key is in. Keep only the appropriate half. Stop when the sublist contains only one element. Using the invariant, we know that if the key was in the original list, then it is this remaining element.

3.4 A Formal Proof of Correctness

Our philosophy is about learning how to think about, develop, and describe algorithms in such way that their correctness is transparent. In order to accomplish this, one needs to at least understand the required steps in a formal proof of correctness.

3.4.1 The Formal Proof Technique

Definition of the Correctness of a Program: An algorithm works correctly on every input instance if, for an arbitrary instance, $\langle pre\!-\!cond\rangle$ & $code_{alg} \Rightarrow \langle post\!-\!cond\rangle$.

Consider some instance. If this instance meets the preconditions, then the output must meet the postconditions. If this instance does not meet the preconditions, then all bets are off (but the program should be polite).

Note that the correctness of an algorithm is only with respect to the stated specifications. It does not guarantee that it will work in situation that are not taken into account by this specification.

Breaking into Parts: The method of proving that an algorithm is correct is by breaking the algorithm into smaller and smaller well defined parts. The task of each part, subpart, and sub-subpart is defined in the same way using pre- and postconditions.

A Single Line of Code: When a part of the algorithm is broken down to a single line of code then this line of code has implied pre- and postconditions and we must trust the compiler to translate the line of code correctly. For example:

$\langle pre\!-\!cond\rangle$: The variables x and y have meaningful values.
z = x + y
$\langle post\!-\!cond\rangle$: The variable z takes on the sum of the value of x and the value of y.
The previous value of z is lost.

Combining the Parts: Once the individual parts of the algorithm are proved to be correct, the correctness of the combined parts can be proved as follows.

Structure of Algorithmic Fragment:
$\langle assertion_0 \rangle$
$code_1$
$\langle assertion_1 \rangle$
$code_2$
$\langle assertion_2 \rangle$

Steps in Proving Its Correctness:

- The proof of correctness of the first part proves that $\langle assertion_0 \rangle$ $\& code_1 \Rightarrow \langle assertion_1 \rangle$
- and of the second part that $\langle assertion_1 \rangle$ $\& code_2 \Rightarrow \langle assertion_2 \rangle$.
- Formal logic allows us to combine these to give $\langle assertion_0 \rangle$ $\&$ $\langle code_1 \& code_2 \rangle \Rightarrow \langle assertion_2 \rangle$.

Proving the Correctness of an *if* Statement: Once we prove that each block of straight line code works correctly, we can prove that more complex algorithmic structures work correctly.

Structure of Algorithmic Fragment:
$\langle pre{-}if \rangle$
if($\langle condition \rangle$) then
    $code_{true}$
else
    $code_{false}$
end if
$\langle post{-}if \rangle$

Steps in Proving Its Correctness: Its correctness $\langle pre{-}if \rangle$ $\&$ $code \Rightarrow \langle post{-}if \rangle$ is proved by proving that each of the two paths through the proof are correct, namely

- $\langle pre{-}if \rangle$ $\&$ $\langle condition \rangle$ $\&$ $code_{true} \Rightarrow \langle post{-}if \rangle$ and
- $\langle pre{-}if \rangle$ $\&$ $\neg \langle condition \rangle$ $\&$ $code_{false>} \Rightarrow \langle pre{-}if \rangle$.

Exponential Number of Paths: An additional advantage of this proof approach is that it substantially decreases the number of different things that you need to prove. Suppose that you have a sequence of n of these if statements. There would be 2^n different paths that a computation might take through the code. If you had to prove separately for each of these paths that the computation works correctly, it would take you a long time. It is much easier to prove the above two statements for each of the n if statements.

Proving the Correctness of a *loop* Statement: Loops are another construct that must be proven correct.

Structure of Algorithmic Fragment:
$\langle pre{-}loop \rangle$
loop
    $\langle loop{-}invariant \rangle$
    exit when $\langle exit{-}cond \rangle$
    $code_{loop}$
end loop
$\langle post{-}loop \rangle$

Steps in Proving Correctness: Its correctness $\langle pre{-}loop \rangle$ $\&$ $code \Rightarrow \langle post{-}loop \rangle$ is proved by breaking a path through the code into subpaths and proving that each these parts works correctly, namely

- $\langle pre{-}loop \rangle \Rightarrow \langle loop{-}invariant \rangle$
- $\langle loop{-}invariant' \rangle$ $\&$ not $\langle exit{-}cond \rangle$ $\&$ $code_{loop} \Rightarrow \langle loop{-}invariant'' \rangle$

- $\langle loop-invariant \rangle$ & $\langle exit-cond \rangle \Rightarrow \langle post-loop \rangle$
- Termination (or even better, giving a bound on the running time).

Infinite Number of Paths: Depending on the number of times around the loop, this code has an infinite number of possible paths through it. We, however, are able to prove all at once that each iteration of the loop works correctly.

This technique is discussed more in the next section.

Proving the Correctness of a Function Call: Function and subroutine calls are also a key algorithmic technique that must be proven correct.

Structure of Algorithmic Fragment:
$\langle pre-call \rangle$
output = RoutineCall(*input*)
$\langle post-call \rangle$

Steps in Proving Correctness: Its correctness $\langle pre-call \rangle$ & *code* $\Rightarrow \langle post-call \rangle$ is proved by ensuring that the input instance past to the routine meets the routine's precondition, trusting that the routine ensures that its output meets its postconditions, and ensuring that this postcondition meets the needs of the algorithm fragment.

- $\langle pre-call \rangle \Rightarrow \langle pre-cond \rangle_{input}$
- $\langle post-cond \rangle_{input} \Rightarrow \langle post-call \rangle$

This technique is discussed more in Chapter 11 on recursion.

3.4.2 Proving Correctness of Iterative Algorithms with Induction

Section 3.3.1 describes the *loop invariant* level of abstraction and what is needed to develop an algorithm within it. The next step is to use induction to prove that this process produces working programs.

Structure of An Iterative Algorithm:

```
⟨pre−cond⟩
code_pre-loop      % Establish loop invariant
loop
    ⟨loop−invariant⟩
    exit when ⟨exit−cond⟩
    code_loop      % Make progress while maintaining the loop invariant
end loop
code_post-loop     % Clean up loose ends
⟨post−cond⟩
```

Steps in Proving Correctness:

- $\langle pre-cond \rangle$ &$code_{pre\text{-}loop} \Rightarrow \langle loop-invariant \rangle$
- $\langle loop-invariant' \rangle$ & not $\langle exit-cond \rangle$ &$code_{loop} \Rightarrow \langle loop-invariant'' \rangle$
- $\langle loop-invariant \rangle$ & $\langle exit-cond \rangle$ &$code_{post\text{-}loop} \Rightarrow \langle post-cond \rangle$
- Termination (or even better, giving a bound on the running time).

Mathematical Induction: Induction is an extremely important mathematical technique for proving statements with a universal quantifier. (See Section 1.1.)

A Statement for Each n: For each value of $n \geq 0$, let $S(n)$ represent a boolean statement. This statement may be true for some values of n and false for others.

Goal: The goal is to prove that for every value of n the statement is true, namely that $\forall n \geq 0,\ S(n)$.

Proof Outline: The proof is by induction on n.

Induction Hypothesis: The first step is to state the induction hypothesis clearly: "For each $n \geq 0$, let $S(n)$ be the statement that ...".

Base Case: Prove that the statement $S(0)$ is true.

Induction Step: For each $n \geq 1$, prove $S(n-1) \Rightarrow S(n)$. The method is to assume that $S(n-1)$ is true and then to prove that it follows that $S(n)$ must also be true.

Conclusion: By way of induction, you can conclude that $\forall n \geq 0,\ S(n)$.

Types: Mind the "type" of everything (as you do when writing a program). n is an integer, not something to be proved. Do not say "assume $n-1$ and prove n". Instead, say "assume $S(n-1)$ and prove $S(n)$" or "assume that it is true for $n-1$ and prove that it is true for n."

The "Process" of Induction:

$S(0)$ is true (by base case)
$S(0) \Rightarrow S(1)$ (by induction step, $n = 1$)
hence, $S(1)$ is true
$S(1) \Rightarrow S(2)$ (by induction step, $n = 2$)
hence, $S(2)$ is true
$S(2) \Rightarrow S(3)$ (by induction step, $n = 3$)
hence, $S(3)$ is true
...

The Connection between Loop Invariants and Induction:

Induction Hypothesis: For each $n \geq 0$, let $S(n)$ be the statement, "If the loop has not yet exited, then the loop invariant is true when you are at the top of the loop after going around n times."

Goal: The goal is to prove that $\forall n \geq 0,\ S(n)$, namely that "As long as the loop has not yet exited, the loop invariant is always true when you are at the top of the loop".

Proof Outline: Proof by induction on n.

Base Case: Proving $S(0)$ involves proving that the loop invariant is true when the algorithm first gets to the top of the loop. This is achieved by proving the statement $\langle pre\!-\!cond\rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop\!-\!invariant\rangle$.

Induction Step: Proving $S(n{-}1) \Rightarrow S(n)$ involves proving that the loop invariant is maintained. This is achieved by proving the statement $\langle loop\!-\!invariant'\rangle$ & not $\langle exit\!-\!cond\rangle$ & $code_{loop}$ $\Rightarrow \langle loop\!-\!invariant''\rangle$.

Conclusion: By way of induction, we can conclude that $\forall n \geq 0,\ S(n)$, i.e., that the loop invariant is always true when at the top of the loop.

Proving that the Iterative Algorithm Works: To prove that the iterative algorithm works correctly on every input instance, you must prove that, for an arbitrary instance, $\langle pre\!-\!cond\rangle$ & $code_{alg} \Rightarrow$ $\langle post\!-\!cond\rangle$.

Consider some instance. Assume that this instance meets the preconditions. Proof by induction proves that as long as the loop has not yet exited, the loop invariant is always true.

The algorithm does not work correctly if it executes forever. Hence, you must prove that the loop eventually exits. According to the loop invariant level of abstraction, the algorithm must make progress of at least one at each iteration. It also gives an upper bound on the amount of progress that can be made. Hence, you know that after this number of iterations at most, the exit condition will be met.

Thus, you also know that, at some point during the computation, both the loop invariant and the exit condition will be simultaneously met. You then use the fact that $\langle loop\!-\!invariant\rangle$ & $\langle exit\!-\!cond\rangle$ & $code_{post\text{-}loop} \Rightarrow \langle post\!-\!cond\rangle$ to prove that the postconditions will be met at the end of the algorithm.

Chapter 4

Examples of Iterative Algorithms

I will now give examples of iterative algorithms. For each example, look for the key steps of the loop invariant paradigm. What is the loop invariant? How is it obtained and maintained? What is the measure of progress? How is the correct final answer ensured?

4.1 VLSI Chip Testing

The following is a strange problem with strange rules. However, it is no stranger than problems that you will need to solve in the world. We will use this as an example of how to develop a strange loop invariant with which the algorithm and its correctness becomes transparent.

Specification: Our boss has n supposedly identical VLSI chips that are potentially capable of testing each other. His test jig accommodates two chips at a time. The result is either that they are the same, i.e. "both are good or both are bad" or that they are different, i.e. "at least one is bad." The professor hires us to design an algorithm to distinguish good chips from bad ones. [CLR]

Impossible? Some computational problems have exponential time algorithms, but no polynomial time algorithms. Because we are limited in what we are able to do, this problem may not have an algorithm at all. It is often hard to know. A good thing to do with a new problem is to alternate between good cop and bad cop. The good cop does his best to design an algorithm for the problem. The bad cop does his best to prove that the good cop's algorithm does not work or even better prove that no algorithm works.

Chip Testing: Suppose that the professor happened to have one good chip and one bad chip. His test would tell him that these chips are different. However, he has no way of knowing which chip is which. Our job is done. We have proved that there is no algorithm using only this single test that is always able to distinguish good chips from bad ones. The professor may not be happy with our findings, but he will not be able to blame us.

Though we have proved that there is no algorithm that distinguishes these two chips, perhaps we can find an algorithm that can be of some use for the professor.

Simple Examples: As just seen, it is useful to try simple examples.

A Data Structure: It is useful to have a good data structure with which to store the information that has been collected.

Chip Testing: Graphs are often useful. (See Section 2.2.) Here we can have a node for each chip. After testing a pair of chips, we put a solid edge between the corresponding nodes if they are reportedly the same and a dotted edge if they are different.

The Brute Force Algorithm: One way of understanding a problem better is to initially pretend that you have unbounded time and energy. With this, what tasks can you accomplish?

Chip Testing: With $\Theta(n^2)$ tests we can test every pair of chips. We assume that the test is commutative, meaning that if chip a test to be the same as b, which test to be the same as c, then a will test to be the same as c. Given this, we can conclude that the tests will partition the chips into sets of chips that are the same. (In graph theory we call these sets *cliques*, like in a clique of friends in which everyone in the group is friends with everyone else in the group.) There is, however, no test available to determine which of these sets contain the good chips.

Change The Problem: When we get stuck, a useful thing to do is to go back to your boss or to the application at hand and see if you can change the problem to make it easier. There are three ways of doing this.

More Tools: One option is to allow the algorithm more powerful tools.

Chip Testing: Certainly, in this example, a test that told you whether a chip was good would solve the problem. On the other hand, if the professor had such a test, you would be out of a job.

Change The Preconditions: You can change the preconditions to require additional information about the input instance or to disallow particularly difficult instances.

Chip Testing: We need some way of distinguishing between the good chips and the various forms of bad chips. Perhaps we can get the professor to assure us that at least half of the chips are good. With this we can solve the problem. We test all pairs of chips and partition the chips into the sets of equivalent chips. The largest of these sets will be the good chips.

Change The Postconditions: Another options is to change the postconditions by not requiring so much in the output.

Chip Testing: Instead of needing to distinguish completely between good and bad chips, an easier task would be to find a single good chip.

A Faster Algorithm: Once we have brute force algorithm, we will want to find a faster algorithm.

Chip Testing: The above algorithm takes $\Theta(n^2)$ time. Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Can we do this faster?

Designing The Loop Invariant: In designing an iterative algorithm for this problem, the most creative step is designing the loop invariant.

Start with Small Steps: What basic steps might you follow to make some kind of progress.

Chip Testing: Certainly the first step is to test two chips. There are two cases.

Different: Suppose that we determine that the two chips are different. In general, one way to make progress is to narrow down the input instance while maintaining what we know about it. What we know is that more than half of the chips are good. Because we know that at least one of the two tested chips is bad, we can throw both of them away. We know that we do not go wrong by doing this because we maintain the loop invariant that more than half of the chips are good. From this we know that there is still at least one good chip remaining which we can return as the answer.

Same: If the two chips test to be the same, we cannot through them away because they might both be good. However, this too seems like we are making progress because, as in the brute force algorithm, we are building up a set of chips that are the same.

Picture from the Middle: What would you like your data structure to look like when you are half done?

Chip Testing: From our single step, we saw two forms of progress. First, we saw that some chips will have been set aside. Let S denote the subset containing all the chips that we have not set aside. Second, we saw that we were building up sets of chips that we know to be the same. It may turn out that we will need to maintain a number of these sets. However, to begin lets start with the simplest picture. Let us build only one such set.

Loop Invariant: Define your loop invariant in a way so that it is clear precisely what it requires and what it does not.

Chip Testing: We maintain two sets. The set S contains the chips that we have not set aside. We maintain that more than half of the chips in S are good. The set C is a subset of S. We maintain that all of the chips in C are the same, though we do not know whether they are all good or all bad.

Maintaining the Loop Invariant: i.e., $\langle loop\!-\!invariant' \rangle$ & not $\langle exit\!-\!cond \rangle$ &$code_{loop}$ $\Rightarrow$ $\langle loop\!-\!invariant'' \rangle$

Chip Testing: Assume that all we know is that the loop invariant is true. Being the only thing that we know how to do, we must test two chips. But which two. Testing two from C is not useful, because we already know that they are the same. Testing two that are not in C is dangerous, because if we learn that they are same, then we will have to start a second set of alike chips and we have decided to maintain only one. The remaining possibility is to choose any chip from C and any from $S-C$ and test them. Let us denote these chips by c and s.

Same: If the conclusion is that the chips are the same, then add chip s to C. We have not changed S so its LI still holds. From our test, we know that s has the same as c. From the LI, we know that c is that same as all the other chips in C. Hence, we know that s is the same as all the other chips in C and the LI follows.

Different: If the conclusion is that "at least one is bad", then delete both c and s from C and S. S has lost two chips, at least one of which is bad. Hence, we have maintained the fact that more than half of the chips in S are good. C has become smaller. Hence, we have maintained the fact that its chips are all the same.

Either way we maintain the loop invariant while making some (yet undefined) progress.

Handle All Cases: Be sure to cover all possible events.

Chip Testing: We can only test one chip from C and one from $S-C$ if both are non-empty. We need to consider the cases in which they are not.

S **Is Empty:** If S is empty, then we are in trouble because we have no more chips to return as the answer. We must stop before this.

$S-C$ **Is Empty:** If $S-C$ is empty, then we know that all the chips in $S = C$ are the same. Because more than half of them must be good, we know that all of them are good. Hence we are done.

C **Is Empty:** If C is empty, take any chip from S and add it to C.
We have not changed S, so its LI still holds. The single chip in C is the same as itself.

Measure of Progress: We need a measure of progress that always decreases.

Chip Testing: Let the measure be $|S-C|$. In two cases, we remove a chip from $S-C$ and add it to C. In another case, we remove a chip from $S-C$ and one from C. Hence, in all cases the measure decreases by 1.

Initial Code (i.e., $\langle pre\!-\!cond \rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop\!-\!invariant \rangle$): The initial code of the algorithm must establish the loop invariant to be true. To do this it relies on the fact the preconditions on the input instance are true.

Chip Testing: Initially, we have neither tested nor thrown away any chips. Let S be all the chips and C be empty. More than half of the chips in S are good according to the problem's precondition. Because there are no chips in C, all the chips that are in it are the same.

Exiting Loop (i.e., $\langle$*loop−invariant*$\rangle$ & $\langle$*exit−cond*$\rangle$ &$code_{post\text{-}loop} \Rightarrow$ $\langle$*post−cond*$\rangle$): We need to design the exit condition of the loop so that we exit as soon as we are able to solve the problem.

Chip Testing: $|S-C| = 0$ is a good halting condition, but not the first. Halt when $|C| > |S|/2$ and return any chip from C. According to the LI, the chips in C are either all good or all bad. The chips in C constitute more than half the chips, so if they were all bad, more than half of the chips in S would also be bad. This contradicts the LI. Hence, the chips in C are all good.

Running Time: We bound the running time, by showing that initially our measure of progress is not too big, that it decreases each iteration, and that if it gets small enough the exit condition will occur.

Chip Testing: Initially, the measure of progress $|S-C|$ is n. We showed that it decreases by at least 1 each iteration. Hence, there are at most n steps before $S-C$ is empty. We are guaranteed to exit the loop by this point because $|S-C| = 0$ insures us that the exit condition $|C| = |S| > |S|/2$ is met. Note that S contains at least one chip because by the loop invariant more than half of them are good.

Additional Observations:

Chip Testing: C can flip back and forth from being all bad to being all good many times. Suppose it is all bad. If s from $S-C$ happens to be bad, then C gets bigger. If s from $S-C$ happens to be good, then C gets smaller. If C ever becomes empty during this process, then a new chip is added to C. This chip may be good or bad. The process repeats.

Extending The Algorithm: The above algorithm finds one good chip. What is the time complexity of finding all the good chips?

Answer: Once you have one good chip, in more $\mathcal{O}(n)$ time, this good chip will tell you which of the other chips are good.

4.2 Colouring The Plane

An input instance consists of a set of n (infinitely long) lines. These lines form a subdivision of the plane, that is, they partition the plane into a finite number of regions (some of them unbounded). The output consists of a colouring of each region with either black or white so that any two regions with a common boundary have different colours. (Note that an algorithm for this problem proves the theorem that such a colouring exists for any such subdivision of the plane.)

Code:

algorithm *ColouringPlane(lines)*

$\langle$***pre−cond***$\rangle$: *lines* specifies n (infinitely long) lines.

$\langle$***post−cond***$\rangle$: C is a proper colouring of the plane subdivided by the lines.

begin
  C = the colouring that colours the entire plane white.
  $i = 0$
  loop
    $\langle$***loop−invariant***$\rangle$: C is a proper colouring of the plane subdivided by the first i lines.

exit when $(i = n)$
% Make progress while maintaining the loop invariant
Line $i + 1$ cuts the plane in half.
On one half, the new colouring C' is the same as the old one C.
On the other one half, the new colouring C' is the same as the old one C,
except white is switched to black and black to white.
$i = i + 1$ & $C = C'$
end loop
return(C)
end algorithm

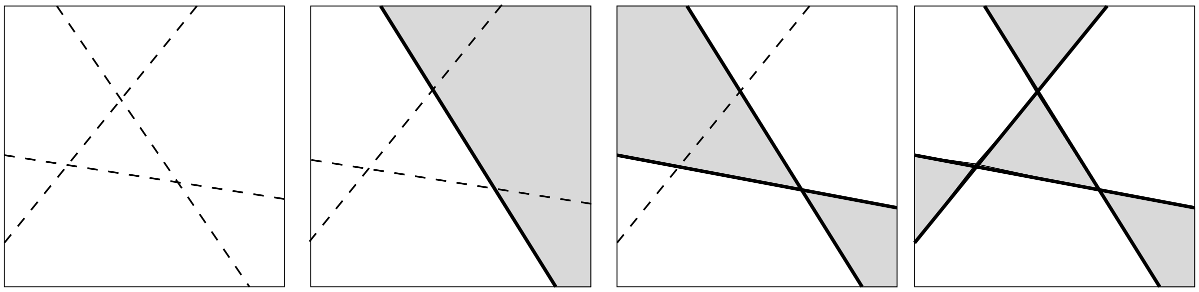


Figure 4.1: An example of colouring the plane.

Proof of Correctness:

$\langle$***pre−cond***$\rangle$ **&** ***code***$_{\boldsymbol{pre\text{-}loop}}$ $\Rightarrow$ $\langle$***loop−invariant***$\rangle$**:**

With $i = 0$ lines, the plane is all one region. The colouring that makes the entire plane white works.

$\langle$***loop−invariant′***$\rangle$ **& not** $\langle$***exit−cond***$\rangle$ **&** ***code***$_{\boldsymbol{loop}}$ $\Rightarrow$ $\langle$***loop−invariant′′***$\rangle$**:**

We know that C is a proper colouring given the first i lines. We must establish that C' is a proper colouring given the first $i + 1$ lines. Consider any two regions with a common boundary in the plane subdivided by the first $i + 1$ lines.

If their boundary is not line $i + 1$, then they were two regions with a common boundary in the plane subdivided by the first i lines. Hence, by the loop invariant, the colouring C gives them different colours. The loop either changes neither of their colours or both of their colours. Hence, they still have different colours in the new colouring C'.

If their boundary is line $i + 1$, then they were one region in the plane subdivided by the first i lines. Hence, by the loop invariant, the colouring C gives them the same colour. The loop changes one of their colours. Hence, they have different colours in the new colouring C'.

$\langle$***loop−invariant***$\rangle$ **&** $\langle$***exit−cond***$\rangle$ **&** ***code***$_{\boldsymbol{post\text{-}loop}}$ $\Rightarrow$ $\langle$***post−cond***$\rangle$**:**

If C is a proper colouring given the first i lines and $i = n$, then clearly C is a proper colouring given all of the lines.

4.3 Euclid's GCD Algorithm

The following is an amazing algorithm for finding the greatest common divisor (GCD) of two integers. E.g., $GCD(18, 12) = 6$. It was first done by Euclid, an ancient Greek. Without the use of loop invariants, you would never be able to understand what the algorithm does; with their help, it is easy.

Specifications:

Preconditions: An input instance consists of two positive integers, a and b.

Postconditions: The output is $GCD(a, b)$.

Loop Invariant: The creative step in designing this algorithm is coming up with the loop invariant. The algorithm maintains two variables x and y whose values change with each iteration of the loop under the invariant that their GCD, $GCD(x, y)$, does not change, but remains equal to the required output $GCD(a, b)$.

Initial Conditions: The easiest way of establishing the loop invariant that $GCD(x, y) = GCD(a, b)$ is by setting x to a and y to b.

Measure of Progress: Progress is made by making x or y smaller.

Ending: We will exit when x or y is small enough that we can compute their GCD easily. By the loop invariant, this will be the required answer.

A Middle Iteration on a General Instance: Let us first consider a general situation in which x is bigger than y and both are positive.

Main Steps: Our goal is to make x or y smaller without changing their GCD. A useful fact is that $GCD(x, y) = GCD(x - y, y)$, eg. $GCD(52, 10) = GCD(42, 10) = 2$. The reason is that any value that divides into x and y also divides into $x - y$ and similarly any value that divides into $x - y$ and y also divides into x. Hence, replacing x with $x - y$ would make progress while maintaining the loop invariant.

Exponential?: Lets jump ahead in designing the algorithm and estimate its running time. A loop executing only $x = x - y$ will iterate $\frac{a}{b}$ times. If b is much smaller than a, then this may take a while. However, even if $b = 1$, this is only a iterations. This looks like it is linear time. However, you should express the running time of an algorithm as a function of input size. See Section 1.3. The number of bits needed to represent the instance $\langle a, b\rangle$ is $n = \log a + \log b$. Expressed in these terms, the running time is $Time(n) = \Theta(a) = \Theta(2^n)$. This is exponential time. For example, if $a = 1,000,000,000,000,000$ and $b = 1$, I would not want to wait for it.

Faster Main Steps: Instead of subtracting one y from x each iteration, why not speed up the process by subtracting many all at once. We could set $x_{new} = x - d \cdot y$ for some integer value of d. Our goal is to make x_{new} as small as possible without making it negative. Clearly, d should be $\lfloor \frac{x}{y} \rfloor$. This gives $x_{new} = x - \lfloor \frac{x}{y} \rfloor \cdot y = x \text{ } mod \text{ } y$, which is within the range $[0..y - 1]$ and is the remainder when dividing y into x, e.g., $52 \text{ } mod \text{ } 10 = 2$.

Maintaining the Loop Invariant: The step $x_{new} = x \text{ } mod \text{ } y$, maintains the loop invariant because $GCD(x, y) = GCD(x \text{ } mod \text{ } y, y)$, e.g., $GCD(52, 10) = GCD(2, 10) = 2$.

Making Progress: The step $x_{new} = x \text{ } mod \text{ } y$ makes progress by making x smaller only if $x \text{ } mod \text{ } y$ is smaller than x. This is only true if x is greater or equal to y. Suppose that initially this is true because a is greater than b. After one iteration of $x_{new} = x \text{ } mod \text{ } y$ becomes smaller than y. Then the next iteration will do nothing. A solution is to then swap x and y.

New Main Steps: Combining $x_{new} = x \text{ } mod \text{ } y$ with a swap gives the main steps of $x_{new} = y$ and $y_{new} = x \text{ } mod \text{ } y$.

Maintaining the Loop Invariant: This maintains our original loop invariant because $GCD(x, y) = GCD(y, x \text{ } mod \text{ } y)$, e.g., $GCD(52, 10) = GCD(10, 2) = 2$. It also maintains the new loop invariant that x less than y.

Making Progress: These new steps do not make x smaller. However, because $y_{new} = x \text{ } mod \text{ } y \in [0..y - 1]$ is smaller than y, we make progress by making y smaller.

Special Cases: Setting $x = a$ and $y = b$ does not establish the loop invariant that x is at least y if a is smaller than b. An obvious solutions is to initially test for this and to swap them if necessary. However, as advised in Section 3.3.1, it is sometimes fruitful to try tracing out what the algorithm that you have already designed would do given such an input. Suppose $a = 10$ and $b = 52$. The first iteration would set $x_{new} = 52$ and $y_{new} = 10 \; mod \; 52$. This last value is a value within the range [0..51] that is the remainder when dividing 10 by 52. Clearly this is 10. Hence, the code automatically swaps the values by setting $x_{new} = 52$ and $y_{new} = 10$. Hence, no new code is needed. Similarly, if a and b happen to be negative, the initial iteration will make y positive and the next will make both x and y positive.

Exit Condition: We are making progress by making y smaller. We should stop when y is small enough that we can solve compute the GCD easily. Lets try small values of y. Using $GCD(x, 1) = 1$, the GCD is easy to compute when $y = 1$, however, we will never get this unless $GCD(a, b) = 1$. How about $GCD(x, 0)$? This turns out to be x because x divides evenly into both x and 0. Lets try an exit condition of $y = 0$.

Termination: We know that the program will eventually stop as follows. $y_{new} = x \; mod \; y \in [0..y-1]$ ensures that each step y gets strictly smaller and does not go negative. Hence, eventually y must be zero.

Ending: Formally we prove that $\langle loop\!-\!invariant \rangle$ & $\langle exit\!-\!cond \rangle$ & $code_{post\text{-}loop} \Rightarrow \langle post\!-\!cond \rangle$. $\langle loop\!-\!invariant \rangle$ gives $GCD(x, y) = GCD(a, b)$ and $\langle exit\!-\!cond \rangle$ gives $y = 0$. Hence $GCD(a, b) = GCD(x, 0) = x$. The final code will return the value of x. This establishes the $\langle post\!-\!cond \rangle$ that $GCD(a, b)$ is returned.

Code:

algorithm $GCD(a, b)$

$\langle$***pre−cond***$\rangle$**:** a and b are integers.

$\langle$***post−cond***$\rangle$**:** Returns $GCD(a, b)$.

```
begin
    int x,y
    x = a
    y = b
    loop
        ⟨loop−invariant⟩: GCD(x,y) = GCD(a,b).
        if(y = 0) exit
        x_new = y    y_new = x mod y
        x = x_new
        y = y_new
    end loop
    return( x )
end algorithm
```

Example: The following traces the algorithm give $a = 22$ and $b = 32$.

iteration	value of x	value of y
1st	22	32
2nd	32	22
3rd	22	10
4th	10	2
5th	2	0

$GCD(22, 32) = 2$.

Running Time: We have seen already that the running time is exponential if y decreases by a small integer each iteration. For the running time to be linear in the size of the input, the number of bits $\log y$ to represent y must decrease by at least one each iteration. This means that y must decrease by at least a factor of two. Consider the example of $x = 19$ and $y = 10$. y_{new} becomes $19 \ mod \ 10 = 9$, which is only a decrease of one. However, the next value of y will be $10 \ mod \ 9 = 1$ which is a huge drop.

What we will be able to prove is that every two iterations, y drops by a factor of two, namely that $y_{k+2} < y_k/2$. There are two cases. In the first case, $y_{k+1} \leq y_k/2$. Then we are done because as stated above $y_{k+2} < y_{k+1}$. In the second case, $y_{k+1} \in [y_k/2 + 1, y_k - 1]$. Unwinding the algorithm gives that $y_{k+2} = x_{k+1} \ mod \ y_{k+1} = y_k \ mod \ y_{k+1}$. One algorithm for computing $y_k \ mod \ y_{k+1}$ is to continually subtract y_{k+1} from y_k until the amount is less than y_{k+1}. Because y_k is more than y_{k+1}, this y_{k+1} is subtracted at least once. It follows that $y_k \ mod \ y_{k+1} \leq y_k - y_{k+1}$. By the case, $y_{k+1} > y_k/2$. In conclusion $y_{k+2} = y_k \ mod \ y_{k+1} \leq y_k - y_{k+1} < y_k/2$.

We prove that the number of times that the loop iterates is $\mathcal{O}(\log(\min(a,b))) = \mathcal{O}(n)$ as follows. After the first or second iteration, y is $\min(a,b)$. Every iteration y goes down by at least a factor of 2. Hence, after k iterations, y_k is at most $\min(a,b)/2^k$ and after $\mathcal{O}(\log(\min(a,b)))$ iterations is at most one.

The algorithm iterates a linear number $\mathcal{O}(n)$ of times. Each iteration must do a *mod* operation. Poor Euclid had to compute these by hand, which must have gotten very tedious. A computer may be able to do *mod*s in one operation, however, the number of bit operations need for two n bit inputs is $\mathcal{O}(n \log n)$. Hence, the time complexity of this GCD algorithm is $\mathcal{O}(n^2 \log n)$.

Lower Bound: We will prove a lower bound, not of the minimum time for any algorithm to find the GCD, but of this particular algorithm by finding a family of input values $\langle a, b\rangle$ for which the program loops $\Theta(\log(\min(a,b)))$ times. Unwinding the code gives $y_{k+2} = x_{k+1} \ mod \ y_{k+1} = y_k \ mod \ y_{k+1}$. As stated, $y_k \ mod \ y_{k+1}$ is computed by subtracting y_{k+1} from y_k a number of times. We want the y's to shrink as slowly as possible. Hence, let us say that it is subtracted only once. This gives $y_{k+2} = y_k - y_{k+1}$ or $y_k = y_{k+1} + y_{k+2}$. This is the definition of Fibonacci numbers only backwards, i.e. $Fib(0) = 0$, $Fib(1) = 1$ and $Fib(n) = Fib(n-1) + Fib(n-2)$. See Section 1.6.4. On input $a = Fib(n+1)$ and $b = Fib(n)$, the program iterates n times. This is $\Theta(\log(\min(a,b)))$, because $Fib(n) = 2^{\Theta(n)}$.

4.4 Magic Sevens

My mom gave my son Joshua a book of magic tricks. Here is one that I knew as a kid. The book writes, "This is a really magic trick. It comes right every time you do it, but there is no explanation why." As it turns out, there is a bug in the way that they explain the trick. Hence, I will explain it differently and more generally. Our task is to fix their bug and to counter their "there is no explanation why." The only Magic is that of Loop Invariants. The algorithm is a variant on binary search.

The Trick:

- Let c, an odd integer, be the number of "columns". They use $c = 3$.
- Let r, an odd integer, be the number of "rows". They use $r = 7$.
- Let $n = c \cdot r$ be the number of cards. They use $n = 21$.
- Let t be the number of iterations. They use $t = 2$.
- Let f be the final index of the selected card. They use $f = 11$.
- Ask someone to select one of the n cards and then shuffle the deck.
- Repeat t times:
  - Spread the cards out as follows. Put c cards in a row left to right face up. Put a second row on top, but shifted down slightly so that you can see what both the first and second row of cards are. Repeat for r rows. This forms c columns of r cards each.
  - Ask which column the selected card is in.

 - Stack the cards in each column. Put the selected column in the middle. (This is why c is odd.)
- Output the f^{th} card.

Our task is to determine for which values c, r, n, t, and f this trick finds the selected card.

Easier Version: Analyzing this trick, turns out to be harder than I initially thought. Hence, consider the following easier trick first. Instead of putting the selected column in the middle of the other columns, put it in the front.

Basic Steps: Each iteration we gain some information about which card had been selected. The trick seems to be similar to binary search. A difference is that binary search splits the current sublist into two parts while this trick splits the entire pile into c parts. A similarity is that each iteration we learn which of these piles the sought after element is in.

Loop Invariant: A good first guess for a loop invariant would be that used by binary search. The loop invariant will state that some subset S_i of the cards contains the selected card. In this easier version, the column containing the card is continuously moved to the front of the stack. Hence, let us guess that $S_i = \{1, 2, \ldots, s_i\}$ indexes the first s_i cards in the deck. The loop invariant states that after i rounds the selected card is one of the first s_i cards in the deck. We must define s_i in terms of c, r, n, and i.

Initial Conditions: Again, as done in binary search, we initially obtain the loop invariant by considering the entire stack of cards. This is done by setting $s_0 = n$ and $S_0 = [1..n]$. The loop invariant then only claims that the selected card is in the deck. This is true before the first iteration.

Maintaining the Loop Invariant: Suppose that before the i^{th} iteration, the selected card is one of the first s_{i-1} in the deck. When the cards are laid out, the first s_{i-1} cards will be spread on the tops of the c columns. Some columns will get $\lceil \frac{s_{i-1}}{c} \rceil$ of these cards and some will get $\lfloor \frac{s_{i-1}}{c} \rfloor$ of them. When we are told which column the selected card is in, we will know that the selected card is one of the first $\lceil \frac{s_{i-1}}{c} \rceil$ cards in this column. We use the ceiling instead of the floor here, because this is the worst case. In conclusion $s_i = \lceil \frac{s_{i-1}}{c} \rceil$. We solve this recurrence relation by guessing that for each i, $s_i = \lceil \frac{n}{c^i} \rceil$. We verify this guess by checking that $s_0 = \lceil \frac{n}{c^0} \rceil = n$ and that $s_i = \lceil \frac{s_{i-1}}{c} \rceil = \left\lceil \frac{\lceil \frac{n}{c^{i-1}} \rceil}{c} \right\rceil = \lceil \frac{n}{c^i} \rceil$.

Exit Condition: Let $t = \lceil \log_c n \rceil$ and let $f = 1$. After t rounds, the loop invariant will establish that the selected card is one of the first $s_t = \lceil \frac{n}{c^t} \rceil = 1$ in the deck. Hence, the selected card must be the first card.

Lower Bound: For a matching lower bound on the number of iterations needed see Section 17.3.

Trick in Book: The book has $n = 21$, $c = 3$, and $t = 2$. Because $21 = n \not\leq c^t = 3^2 = 9$, the trick in the book does not work. Two rounds is not enough. There needs to be three.

Original Trick: Consider again the original trick where the selected column is put into the middle.

Loop Invariant: Because the selected column is put into the middle, let us guess that S_i consists of the middle s_i cards. More formally, let $d_i = \frac{n-s_i}{2}$. Neither the first nor the last d_i cards will be the selected card. Instead it will be one of $S_i = \{d_i + 1, \ldots, d_i + s_i\}$. Note, however, that this requires that s_i is odd.

Initial Conditions: For $i = 0$, $s_0 = n$, $d_0 = 0$, and the selected card can be any card in the deck.

Maintaining the Loop Invariant: Suppose that before the i^{th} iteration, the selected card is not one of the first d_{i-1} cards, but is one of the middle s_{i-1} in the deck. Then when the cards are laid out, the first d_{i-1} cards will be spread on the tops of the c columns. Some columns will get $\lceil \frac{d_{i-1}}{c} \rceil$ of these cards and some will get $\lfloor \frac{d_{i-1}}{c} \rfloor$ of them. In general, however, we can say that the first $\lfloor \frac{d_{i-1}}{c} \rfloor$ cards of each column are not the selected card. We use the floor instead of the ceiling here, because this is the worst case. By symmetry, we also know that the selected card is not one of the last $\lfloor \frac{d_{i-1}}{c} \rfloor$ cards in each column.

When the person points at a column, we learn that the selected card is somewhere in that column. However, from before we knew that the selected card is not one of the first or last $\lfloor \frac{d_{i-1}}{c} \rfloor$ cards in this column. There are only r cards in the column. Hence, the selected card must be one of the middle $r - 2\lfloor \frac{d_{i-1}}{c} \rfloor$ cards in the column. Define s_i to be this value. The new deck is formed by stacking the columns together with these cards in the middle.

Running Time: When sufficient rounds have occurred so that $s_t = 1$, then the selected card will be in the middle indexed by $f = \lceil \frac{n}{2} \rceil$.

Trick in Book: The book has $n = 21$, $c = 3$, and $r = 7$.

$s_i = r - 2\lfloor \frac{d_{i-1}}{c} \rfloor$	$d_i = \frac{n-s_i}{2}$	$S_i = \{d_i + 1, \ldots, d_i + s_i\}$
$s_0 = n = 21$	$d_0 = \frac{21-21}{2} = 0$	$S_0 = \{1, 2, \ldots, 21\}$
$s_1 = 7 - 2\lfloor \frac{0}{3} \rfloor = 7$	$d_1 = \frac{21-7}{2} = 7$	$S_1 = \{8, 9, \ldots, 14\}$
$s_2 = 7 - 2\lfloor \frac{7}{3} \rfloor = 3$	$d_2 = \frac{21-3}{2} = 9$	$S_2 = \{10, 11, 12\}$
$s_3 = 7 - 2\lfloor \frac{9}{3} \rfloor = 1$	$d_3 = \frac{21-1}{2} = 10$	$S_3 = \{11\}$

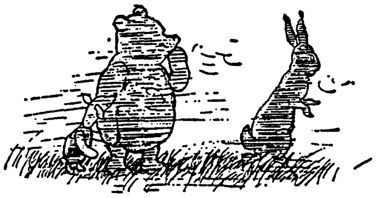
Again three and not two rounds are needed.

Running Time: Temporally ignoring the floor in the equation for s_i makes the analysis easier. $s_i = r - 2\lfloor \frac{d_{i-1}}{c} \rfloor \approx r - 2\frac{d_{i-1}}{c} = \frac{n}{c} - 2\frac{(n-s_{i-1})/2}{c} = \frac{s_{i-1}}{c}$. Again, this recurrence relation gives that $s_i = \frac{n}{c^i}$. If we include the floor, challenging manipulations give that $s_i = 2\lceil \frac{s_{i-1}}{2c} - \frac{1}{2} \rfloor + 1$. More calculations give that s_i is always $\frac{n}{c^i}$ rounded up to the next odd integer.

Chapter 8

Graph Search Algorithms

A surprisingly large number of problems in computer science can be expressed as a graph theory problem. In this chapter, we will first learn a generic search algorithm in which the algorithm finds more and more of the graph by following arbitrary edges from nodes that have already been found. We also consider the more specific orders of depth first and breadth first search to traverse the graph. Using these ideas, we are able to discover shortest paths between pairs of nodes and learn information about the structure of the graph.



8.1 A Generic Search Algorithm

Specifications of the Problem: *Reachability-from-single-source s*

Preconditions: The input is a graph G (either directed or undirected) and a source node s.

Postconditions: The output consists of all the nodes u that are reachable by a path in G from s.

Basic Steps: Consider what basic steps or operations will make progress towards solving this problem.

Suppose you know that node u is reachable from s (denoted as $s \longrightarrow u$) and that there is an edge from u to v. Then you can conclude that v is reachable from s (i.e., $s \longrightarrow u \rightarrow v$). You can use such steps to build up a set of reachable nodes.

- s has an edge to v_4 and v_9. Hence, v_4 and v_9 are reachable.
- v_4 has an edge to v_7 and v_3. Hence, v_7 and v_3 are reachable.
- v_7 has an edge to v_2 and v_8. ...

Difficulties:

- How do you keep track of all this?
- How do you know that you have found all the nodes?
- How do you avoid cycling, i.e., reaching the same node many times, as in $s \rightarrow v_4 \rightarrow v_7 \rightarrow v_2 \rightarrow v_4 \rightarrow v_7 \rightarrow v_2 \rightarrow v_4 \rightarrow v_7 \rightarrow v_2 \rightarrow v_4 \ldots$ forever?

Designing the Loop Invariant: Look at a few of the basic steps. How have you made "progress"? Describe what the data structure looks like. A loop invariant should leave the reader with an image. Draw a picture if you like.

Found: If you trace a path from s to a node, then we will say that the node has been *found*.

Handled: At some point in time after node u has been found, you will want to follow all the edges from u and find all the nodes v that have edges from u. When you have done that for node u, we say that it has been *handled*.

Data Structure: You must maintain (1) the set of nodes $foundHandled$ that have been found and handled and (2) the set of nodes $foundNotHandled$ that have been found but *not* handled.

Loop Invariant:

LI1: For each found node v, we know that v is reachable from s because we have traced out a path $s \longrightarrow v$ from s to it.

LI2: If a node has been handled, then all of its neighbors have been found.

Recall that a loop invariant should follow easily from the preconditions. We must be able to maintain it while making some (as yet undefined) kind of progress. When the "exit" conditions are met, we must be able to conclude that the postconditions have been met. The above loop invariant is simple enough that the first two requirements should be easy to meet. But does it suffice to prove the postcondition? We will see.

Body of the Loop: A reasonable step would be:

- Choose some node u from $foundNotHandled$ and handle it. This involves following all the edges from u.
- Newly-found nodes are now added to the set $foundNotHandled$ (if they have not been found already).
- u is moved from $foundNotHandled$ to $foundHandled$.

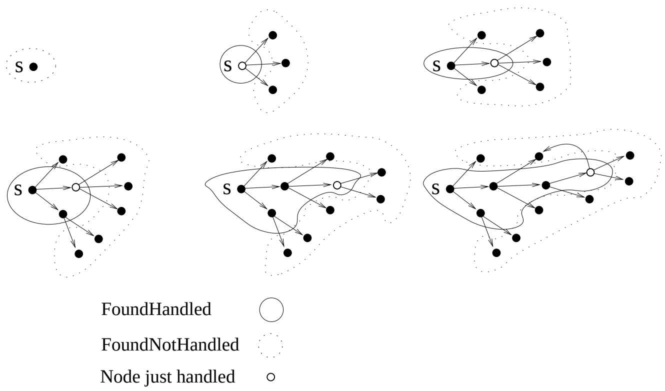


Figure 8.1: The generic search algorithm *handles* one found node at a time by *finding* their neighbors.

Code:

algorithm $Search(G, s)$

$\langle$***pre−cond***$\rangle$**:** G is a (directed or undirected) graph and s is one of its nodes.

$\langle$***post−cond***$\rangle$**:** The output consists of all the nodes u that are reachable by a path in G from s.

begin
    $foundHandled = \emptyset$
    $foundNotHandled = \{s\}$
    loop

```
        ⟨loop−invariant⟩: See above.
        exit when foundNotHandled = ∅
        let u be some node from foundNotHandled
        for each v connected to u
            if v has not previously been found then
                add v to foundNotHandled
            end if
        end for
        move u from foundNotHandled to foundHandled
    end loop
    return foundHandled
end algorithm
```

Maintaining the Loop Invariant (i.e., $\langle LI' \rangle$ & not $\langle exit \rangle$ & $code_{loop} \to \langle LI'' \rangle$): Suppose that LI' which denotes the statement of the loop invariant before the iteration is true, the exit condition $\langle exit \rangle$ is not, and we have executed another iteration of the algorithm.

Maintaining LI1: After the iteration, the node v is considered found. Hence, in order to maintain the loop invariant, we must be sure that v is reachable from s. Because u was in $foundNotHandled$, the loop invariant assures us that we have traced out a path $s \longrightarrow u$ to it. Now that we have traced the edge $u \to v$, we have traced a path $s \longrightarrow u \to v$ to v.

Maintaining LI2: Node u is designated handled only after ensuring that all its neighbors have been found.

Measure of Progress: The measure of progress requires the following three properties:

Progress: We must guarantee that our measure of progress increases by at least one every time around the loop. Otherwise, we may loop forever, making no progress.

Bounded: There must be an upper bound on the progress required before the loop exits. Otherwise, we may loop forever, increasing the measure of progress to infinity.

Conclusion: When sufficient progress has been made to exit, we must be able to conclude that the problem is solved.

An obvious measure would be the number of found nodes. The problem is that when handling a node, you may only find nodes that have already been found. In such a case, no progress is actually made.

A better measure of progress is the number of nodes that have been handled. We can make progress simply by handling a node that has not yet been handled. We also know that if the graph G has only n nodes, then this measure cannot increase past n.

Exit Condition: Given our measure of progress, when are we finished? We can only handle nodes that have been found and not handled. Hence, when all the nodes that have been found have also been handled, we can make no more progress. At this point, we must stop.

Initial Code (i.e., $\langle pre-cond \rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop-invariant \rangle$): Initially, we know only that s is reachable from s. Hence, let's start by saying that s is found but not handled and that all other nodes have not yet been found.

Exiting Loop (i.e., $\langle LI \rangle$ & $\langle exit \rangle \to \langle post \rangle$): Our output will be the set of found nodes. The postcondition requires the following two claims to be true.

Claim: Found nodes are reachable from s.
This is clearly stated in the loop invariant.

Claim: Every reachable node has been found. A logically equivalent statement is that every node that has not been found is not reachable.

One Proof:

- Draw a circle around the nodes of the graph G that have been found.
- If there are no edges going from the inside of the circle to the outside of the circle, then there are no paths from s to the nodes outside of the circle. Hence, we can claim we have found all the nodes reachable from s.
- How do we know that this circle has no edges leaving it?
  - Consider a node u in the circle. Because u has been found and $foundNotHandled = \emptyset$, we know that u has also been handled.
  - By the loop invariant LI2, if $\langle u, v\rangle$ is an edge, then v has been found and thus is in the circle as well.
  - Hence, if u is in the circle and $\langle u, v\rangle$ is an edge, then v is in the circle as well (i.e., no edges leave the circle).
  - This is known as a *closure property.* See Section 16.2.8 for more information on this property.

Another Proof: Proof by contradiction.

- Suppose that w is reachable from s and that w has not been found.
- Consider a path from s to w.
- Because s has been found and w has not, the path starts in the set of found nodes and at some point leaves it.
- Let $\langle u, v\rangle$ be the first edge in the path for which u but not v has been found.
- Because u has been found and $foundNotHandled = \emptyset$, it follows that u has been handled.
- Because u has been handled, v must be found.
- This contradicts the definition of v.

Running Time:

A Simple but False Argument: For every iteration of the loop, one node is handled and no node is handled more then once. Hence, the measure of progress (the number of nodes handled) increases by one with every loop. G only has $|V| = n$ nodes. Hence, the algorithm loops at most n times. Thus, the running time is $\mathcal{O}(n)$.

This argument is false, because while handling u we must consider v for every edge coming out of u.

Overestimation: Each node has at most n edges coming out of it. Hence, the running time is $\mathcal{O}(n^2)$.

Correct Complexity: Each edge of G is looked at exactly twice, once from each direction. The algorithm's time is dominated by this fact. Hence, the running time is $\mathcal{O}(|E|)$, where E is the set of edges in G.

The Order of Handling Nodes: This algorithm specifically did not indicate which node u to select from $foundNotHandled$. It did not need to, because the algorithm works no matter how this choice is made. We will now consider specific orders in which handle the nodes and specific applications of these orders.

Queue/Breadth-First Search: One option is to handle nodes in the order they are found. This treats $foundNotHandled$ as a queue: "first in, first out". Try this out on a few graphs. The effect is that the search is *breadth first*, meaning that all nodes at distance 1 from s are handled first, then all those at distance two, and so on. A byproduct of this is that we find for each node v a shortest path from s to v. See Section 8.2.

Priority Queue/Shortest (Weighted) Paths: Another option calculates for each node v in $foundNotHandled$ the minimum weighted distance from s to v along any path seen so far. It then handles the node that is closest to s according to this approximation. Because these approximations change through out time, $foundNotHandled$ is implemented using a priority queue: "highest current priority out first". Try this out on a few graphs. Like breadth-first search, the search handles nodes that are closest to s first, but now the length of a path is the sum of its edge weights. A by-product of this method is that we find for each node v the shortest weighted path from s to ‘v. See Section 8.3.

Stack/Depth-First Search: Another option is to handle the node that was found most recently. This method treats $foundNotHandled$ as a stack: “last in, first out”. Try this out as well. The effect is that the search is *depth first*, meaning that a particular path is followed as deeply as possible into the graph until a dead-end is reached, forcing the algorithm to backtrack. See Section 8.4.

See Section 2.2 for an explanation of stacks, queues, and priority queues.

8.2 Breadth-First Search/Shortest Paths

We will now develop an algorithm for the *shortest-paths problem*. The algorithm uses a *breadth-first search*. This algorithm is the same as the generic algorithm we looked at previously, but it is less generic, because the order in which the nodes are handled is now specified more precisely. The previous loop invariants are strengthened in order to solve the shortest-paths problem.

Two Versions of the Problem:

The Shortest-Path st Problem: Given a graph G (either directed or undirected) and specific nodes s and t, the problem is to find one of the shortest paths from s to t in G.

Ingredients of an Optimization Problem: This problem has the ingredients of an optimization problem. Such problems involve searching for a best solution from some large set of solutions. See Section 15.1.1 for a formal definition.

Instances: An instance $\langle G, s, t\rangle$ consists of a graph G and specific nodes s and t.

Solutions for Instance: A solution for instance $\langle G, s, t\rangle$ is a path π from s to t.

Cost of Solution: The length (or cost) of a path π is the number of edges in the path.

Goal: Given an instance $\langle G, s, t\rangle$, the goal is to find an optimal solution, i.e., a shortest path from s to t in G.

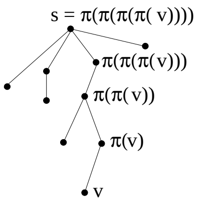
As in other optimization problems, the set of solutions for an instance (i.e., the set of paths from s to t) may well be exponential. We do not want to check them all.

The Shortest Path - Single Source s Multiple Sink t:

Preconditions: The input is a graph G (either directed or undirected) and a source node s.

Postconditions: The output consists of a d and a π for each node of G. It has the following properties:

1. For each node v, $d(v)$ gives the length $\delta(s, v)$ of the shortest path from s to v.
2. The *shortest paths* or *breadth-first search tree* is defined using π as follows: s is the root of the tree. $\pi(v)$ is the parent of v in the tree. For each node v, one of the shortest paths from s to v is given backward, with $v, \pi(v), \pi(\pi(v)), \pi(\pi(\pi(v))), \ldots, s$. A recursive definition is that this shortest path from s to v is the given shortest path from s to $\pi(v)$, followed by the edge $\langle \pi(v), v\rangle$.



Prove Path Is Shortest: In order to claim that the shortest path from s to v is of some length $d(v)$, you must do two things:

Not Further: You must produce a suitable path of this length. We call this path a *witness* of the fact that the distance from s to v is at most $d(v)$.

Not Closer: You must prove that there are no shorter paths. This is harder. Other than checking an exponential number of paths, how can you prove that there are no shorter paths?

These are accomplished as follows.

Not Further: In *finding* a node, we trace out a path from s to it. If we have already traced out a shortest path from s to u with $d(u)$ edges in it and we trace an edge from u to v, then we have traced a path from s to v with $d(v) = d(u) + 1$ edges in it. In this path from s to v, the node preceding v is $\pi(v) = u$.

Not Closer: As I already said, proving that there are no shorter paths is somewhat difficult. We will do it using the following trick: Suppose we can ensure that the order in which we find the nodes is according to the length of the shortest path from s to them. Then, when we find v, we know that there isn't a shorter path to it or else we would have found it already.

Definition V_j: Let V_j denote the set of nodes at distance j from s.

Loop Invariant: This proof method requires the following loop invariants to be maintained.

LI1: For each found node v, $d(v)$ and $\pi(v)$ are as required, i.e., they give the shortest length and a shortest path from s to the node.

LI2: If a node has been handled, then all of its neighbors have been found.

LI3: So far, the order in which the nodes have been found is according to the length of the shortest path from s to it, i.e. the nodes in V_j before those in V_{j+1}.

Order to Handle Nodes: The only way in which we are changing the general search algorithm is by being more careful in our choice of which node from $foundNotHandled$ to handle next. According to LI3, the nodes that were found earlier are closer to s than those that are found later. The closer a node is to s, the closer are its neighbors. Hence, in an attempt to find close nodes, the algorithm will next handle the most recently found node. This is accomplished by treating the set $foundNotHandled$ as a queue, "first in, first out".

Maintaining the Loop Invariant LI3 (Informal): The loop invariant LI3 ensures that the nodes in V_j are found before any node in V_{j+1} is found. Handling the nodes in the order in which they were found will ensure that all the nodes in V_j are handled before any node in V_{j+1} is handled. Handling all the nodes in V_j will find all the nodes in V_{j+1} before any node in V_{j+2} is found.

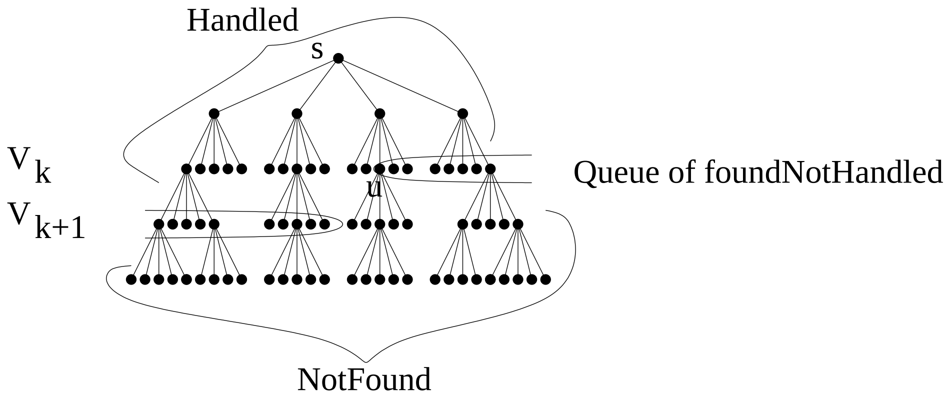


Figure 8.2: Breadth-First Search Tree: We cannot assume that the graph is a tree. Here I have presented only the tree edges given by π. The figure helps to explain the loop invariant, showing which nodes have been found, which found but not handled, and which handled. The dotted line separates the nodes V_0 V_1, V_2, ... with different distances d from s.

Body of the Loop: Remove the first node u from the $foundNotHandled$ queue and handle it as follows. For every neighbor v of u that has not been found,

- add the node to the queue,
- let $d(v) = d(u) + 1$,
- let $\pi(v) = u$, and
- consider u to be handled and v to be $foundNotHandled$.

Code:

algorithm $ShortestPath\,(G, s)$

$\langle \boldsymbol{pre-cond} \rangle$**:** G is a (directed or undirected) graph and s is one of its nodes.

$\langle \boldsymbol{post-cond} \rangle$**:** π specifies a shortest path from s to each node of G and d specifies their lengths.

begin
    $foundHandled = \emptyset$
    $foundNotHandled = \{s\}$
    $d(s) = 0$, $\pi(s) = \epsilon$
    loop
        $\langle \boldsymbol{loop-invariant} \rangle$**:** See above.
        exit when $foundNotHandled = \emptyset$
        let u be the node in the front of the queue $foundNotHandled$
        for each v connected to u
            if v has not previously been found then
                add v to $foundNotHandled$
                $d(v) = d(u) + 1$
                $\pi(v) = u$
            end if
        end for
        move u from $foundNotHandled$ to $foundHandled$
    end loop
    (for unfound v, $d(v) = \infty$)
    return $\langle d, \pi \rangle$
end algorithm

Example:

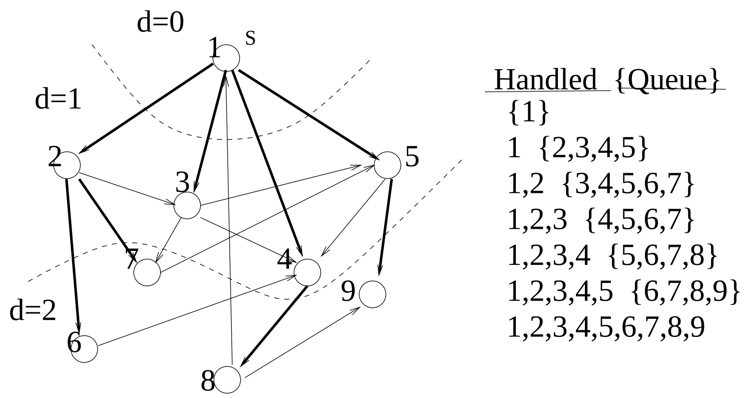


Figure 8.3: Breadth-First Search of a Graph. The numbers show the order in which the nodes were found. The contents of the queue are given at each step. The tree edges are darkened.

Maintaining the Loop Invariant (i.e., $\langle \boldsymbol{LI'} \rangle$ & not $\langle \boldsymbol{exit} \rangle$ & $\boldsymbol{code_{loop}} \rightarrow \langle \boldsymbol{LI''} \rangle$): Suppose that LI' which denotes the statement of the loop invariant before the iteration is true, the exit condition $\langle exit \rangle$ is not, and we have executed another iteration of the algorithm.

Closer Nodes Have Already Been Found: We will need the following claim twice.

Claim: If the first node in the queue $foundNotHandled$, i.e. u, is in V_k, then

1. all the nodes in V_0, V_1, V_2, ..., V_{k-1} have already been found and handled
2. and all the nodes in V_k have already been found.

Proof of Claim 1: Let u' denote any node in V_0, V_1, V_2, ..., V_{k-1}. Because LI3' ensures that nodes have been found in the order of their distance and because because u' is closer to s than u, u' must have been found earlier than u. Hence, u' cannot be in the queue $foundNotHandled$ or else it would be earlier in the queue than u, yet u is first. This proves that u' has been handled.

Proof of Claim 2: Consider any node v in V_k and any path of length k to it. Let u' be the previous node in this path. Because the subpath to u' is of length $k-1$, u' is in V_{k-1}, and hence by claim 1 has already been handled. Therefore, by LI2', the neighbors of u', of which v is one, must have been found.

Maintaining LI1: During this iteration, all the neighbors v of node u that had not been found are now considered found. Hence, their $d(v)$ and $\pi(v)$ must now give the shortest length and a shortest path from s. The code sets $d(v)$ to $d(u)+1$ and $\pi(v)$ to u. Hence, we must prove that the neighbors v are in V_{k+1}. As said above in the general steps, there are two steps to do this.

Not Further: There is a path from s to v of length $k+1$: follow the path of length k to u and then take edge to v. Hence, the shortest path to v can be no longer then this.

Not Closer: We know that there isn't a shorter path to v or it would have been found already. More formally, the claim states that all the nodes in V_0, V_1, V_2, ..., V_k have already been found. Because v has not already been found, it cannot be one of these.

Maintaining LI2: Node u is designated handled only after ensuring that all its neighbors have been found.

Maintaining LI3: By the claim, all the nodes in V_0, V_1, V_2, ..., V_k have already been found and hence have already been added to the queue. Above we prove that the node v being found is in V_{k+1}. It follows that the order in which the nodes are found continues to be according to their distance from s.

Initial Code (i.e., $\langle pre \rangle \rightarrow \langle LI \rangle$): The initial code puts the source s into $foundNotHandled$ and sets $d(s) = 0$ and $\pi(s) = \epsilon$. This is correct, given that initially s has been found but not handled. The other nodes have not been found and hence their $d(v)$ and $\pi(v)$ are irrelevant. The loop invariants follow easily.

Exiting Loop (i.e., $\langle LI \rangle$ & $\langle exit \rangle \rightarrow \langle post \rangle$): The general-search postconditions prove that all reachable nodes have been found. LI1 states that for these nodes the values of $d(v)$ and $\pi(v)$ are as required.

For the nodes that are unreachable from s, you can set $d(v) = \infty$ or you can leave them undefined. In some applications (such as the world wide web), you have no access to unreachable nodes. An advantage of this algorithm is that it never needs to know about a node unless it has been found.

Exercise 8.2.1 *(See solution in Section 20) Suppose u is being handled, $u \in V_k$, and v is a neighbor of u. For each of the following cases, explain which $V_{k'}$ v might be in:*

- *$\langle u, v\rangle$ is an undirected edge, and v has been found before.*
- *$\langle u, v\rangle$ is an undirected edge, and v has not been found before.*
- *$\langle u, v\rangle$ is a directed edge, and v has been found before.*
- *$\langle u, v\rangle$ is a directed edge, and v has not been found before.*

8.3 Shortest-Weighted Paths

We will now make the shortest-paths problem more general by allowing each edge to have a different weight (length). The length of a path from s to v will be the sum of the weights on the edges of the path. Only small changes need to be made to the algorithm. This algorithm is called *Dijkstra's algorithm*.

Specifications of the Problem: Name: The *shortest-weighted-path* problem.

Preconditions: The input is an graph G (either directed or undirected) and a source node s. Each edge $\langle u, v\rangle$ is allocated a weight $w_{\langle u,v\rangle} \in (0, \infty]$.

Postconditions: The output consists of d and π, where for each node v of G, $d(v)$ gives the length $\delta(s, v)$ of the shortest-weighted path from s to v and π defines a *shortest-weighted-paths tree*. (See Section 8.2.)

Basic Steps: As before, proving that the shortest path from s to v is of some length $d(v)$ involves producing a suitable path of this length and proving that there are no shorter paths.

Not Further: The path is traced out as before. The only change is that when we find a path $s \longrightarrow u \rightarrow v$, we compute its length to be $d(v) = d(u) + w_{\langle u,v\rangle}$.

Not Closer: Again this is done by handling (not finding) the nodes v in the order according to the length of the shortest path from it to s. When we find v, we know that there is no shorter path to it because otherwise we would have handled it already.

Circular Argument?: This seems like a circular argument. Initially, we do not know the length of the shortest path to node, so how can we possibly handle the nodes in this order. In breadth-first search, the argument seemed more reasonable, because the nodes that have a small distance from s are only a few edges from s and hence these can be found quickly. However, now the shortest path to a node may wind deep into the graph along many short edges instead of along a few long edges.

Growing The Tree One Edge At A Time: One key to a happy life is to give thought only to today and leave the problems of tomorrow until then. As we have seen, this is the philosophy of all iterative loop invariant algorithms. Suppose that at the beginning of the i^{th} day we have already handled the i nodes with the shortest paths from s. Our only task is to determine which of the unhandled nodes has the shortest path from s and handle this node.

A Property of the Next Node: The following claim will help us narrow down the search for this next node.

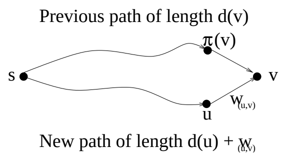
Claim: Let $s = v_0, v_1, v_2, \ldots v_{i-1}, v_i$, be the $i+1$ nodes with the shortest paths from s. Then any shortest path from s to v_i will contain only nodes from $s = v_0, v_1, \ldots v_{i-1}$, except of course for the last node which will be v_i itself.

Proof of Claim: Suppose that this shortest path $s \longrightarrow w \longrightarrow v_i$ contains the node w. The subpath $s \longrightarrow w$ to w must be shorter than the shortest path to v_i because the edges in the rest of the path $w \longrightarrow v_i$ all have positive lengths (weights). Hence, the shortest path to w is less that to v_i and hence w must be one of $s = v_0, v_1, \ldots v_{i-1}$.

Because the next node to handle is always only one edge from a previously handled node, we can slowly expand the tree of handled nodes out, one edge (node) at a time.

The Greedy Criteria: We still must, however, determine which node to handle next. To do this we will simply choose the one that looks best according to the *greedy criteria* of which appears to be closest to s according to our current approximation. (See Chapter 10 for more on greedy algorithms.) For every node v, $d(v)$ and $\pi(v)$ will give the shortest length and path from s to v from among those paths that we have considered so far. This information is continuously updated as we find shorter paths to v. For example, if we find v when handling u, then we update these values as follows:

$foundPathLength = d(u) + w_{\langle u,v\rangle}$
if $d(v) > foundPathLength$ then
    $d(v) = foundPathLength$
    $\pi(v) = u$
end if



Later, if we again find v when handling another node u', then these values are updated again in the same way.

The next obvious question is how accurately this $d(v)$ approximates the actual length of the shortest path. We will see that it gives the length of the shortest path from amongst those paths from s to v that we have *handled*. This set of paths is defined as follows.

Definition of A Handled Path: When handling node u, we follow the edges from u to its neighbors v. When this has been done, we say that the edge from u to v has be *handled*. (The edge or half-edge from v to u might not have been handled.) We say that a path has been handled if it contains only handled edges. Such paths start at s, visit as any number of handled nodes, and then follow one last edge to a node that may or may not be handled. (See the paths to u and to v in Figure 8.5.)

Choosing Next Node to Handle: From among the nodes not yet handled, we will choose the node u with the smallest $d(u)$ value, i.e., the one that (as far as we know) is the closest to s.

Priority Queue: Searching the remaining list of nodes each iteration for the next best node would be too time consuming. Re-sorting the nodes each iteration according to the new greedy criteria would also be too time consuming. A more efficient implementation uses a priority queue to hold the remaining nodes prioritized according to the current greedy criteria. This can be implemented using a Heap. (See Section 6.1.) We will denote this priority queue by $notHandled$.

Consider All Nodes "Found": No path has yet been handled to any node that has not yet been found, and hence $d(v) = \infty$. If we add these nodes to the queue, they will be selected last. Therefore, there is no harm in adding them. Hence, we will distinguish only between those nodes that have been handled and those that have not.

Loop Invariant:

LI1: For each handled node v, the values of $d(v)$ and $\pi(v)$ are as required, i.e., they give the shortest length and a shortest path from s.

LI2: For each of the unhandled nodes v, the values of $d(v)$ and $\pi(v)$ give the shortest length and path from among those paths that have been *handled*.

LI3: So far, the order in which the nodes have been handled is according to the length of the shortest path from s to it.

Body of Loop: Take the next node from the priority queue $notHandled$ and handle it. This involves handling all edges $\langle u, v\rangle$ out of this node u. Handling edge $\langle u, v\rangle$ involves updating the $d(v)$ and $\pi(v)$ values. The priorities of these nodes are changed in the priority queue as necessary.

Code:

algorithm *ShortestWeightedPath* (G, s)

$\langle$***pre−cond***$\rangle$: G is a weighted (directed or undirected) graph and s is one of its nodes.

$\langle$***post−cond***$\rangle$: π specifies a shortest weighted path from s to each node of G and d specifies their lengths.

begin
    $d(s) = 0$, $\pi(s) = \epsilon$
    for other v, $d(v) = \infty$ and $\pi(v) = nil$

$handled = \emptyset$
$notHandled$ = priority queue containing all nodes. Priorities given by $d(v)$.
loop
    ⟨***loop−invariant***⟩**:** See above.
    exit when $notHandled = \emptyset$
    let u be a node from $notHandled$ with smallest $d(u)$
    for each v connected to u
        $foundPathLength = d(u) + w_{\langle u,v \rangle}$
        if $d(v) > foundPathLength$ then
            $d(v) = foundPathLength$
            (update the $notHandled$ priority queue)
            $\pi(v) = u$
        end if
    end for
    move u from $notHandled$ to $handled$
end loop
return $\langle d, \pi \rangle$
end algorithm

Example:

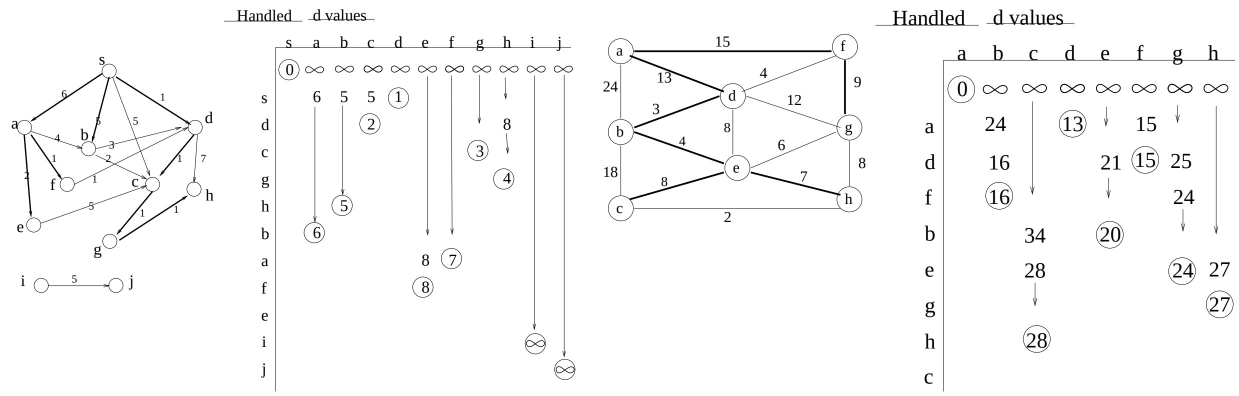


Figure 8.4: Dijkstra's Algorithm. The d value at each step is given for each node. The tree edges are darkened.

Maintaining the Loop Invariant (i.e., $\langle LI' \rangle$ & not $\langle exit \rangle$ & $code_{loop} \rightarrow \langle LI'' \rangle$): Suppose that LI' which denotes the statement of the loop invariant before the iteration is true, the exit condition $\langle exit \rangle$ is not, and we have executed another iteration of the algorithm.

Maintaining LI1 and LI3: The loop handles node u. Hence to maintain LI1, we much ensure that its $d(u)$ and $\pi(u)$ values are as required. In order to maintain LI3, we much ensure that u is the unhandled node with the next shortest path from s. To do these two things, let we will let v_i denote the unhandled node with the next shortest path from s and will prove the following two claims.

$d(v_i)$ and $\pi(v_i)$ are as required: The earlier claim proves that any shortest path from to v_i will contain only nodes from $s = v_0, v_1, \ldots v_{i-1}$, except of course for the last node which will be v_i itself, where these are the $i+1$ nodes with the shortest paths from s. LI3' ensures that these nodes have been handled already. Hence, by the definition of a handled path, this shortest path to v_i is a *handled* path. LI2' ensures that $d(v_i)$ and $\pi(v_i)$ give the shortest length and path from among those paths that have been *handled.* Combining these two facts gives that $d(v_i)$ and $\pi(v_i)$ are in fact that for the over all shortest path to v_i and hence are as required.

$v_i = u$: The node v_i must have the smallest $d(u)$ value from amongst the unhandled nodes or else this node (by LI2') would have a shorter path from s contradicting the definition of v_i being the unhandled node with the next shortest path. It follows that v_i is in fact the node u that is removed from the *notHandled* priority queue.

These facts about the next node handled prove that both LI1 and LI3 are maintained.

Maintaining LI2: Consider some node v. Let $d'(v)$ and $d''(v)$ be the values before and after this iteration of the loop. Let $\mathcal{P}'$ and $\mathcal{P}''$ be the set of paths that were handled before and after this iteration. LI2' gives that $d'(v)$ is the length of the shortest path to v from among the paths in $\mathcal{P}'$. The code in the loop sets $d''(v)$ to be $\min\{d'(v), d'(u) + w_{\langle u,v\rangle}\}$ and considers more paths $\mathcal{P}''$. We must prove LI2", i.e. that $d''(v)$ is the length of the shortest path from among those in $\mathcal{P}''$.

The set of handled paths $\mathcal{P}''$ changes because the loop handles u. We consider the paths in $\mathcal{P}''$ in the following three cases (see Figure 8.5):

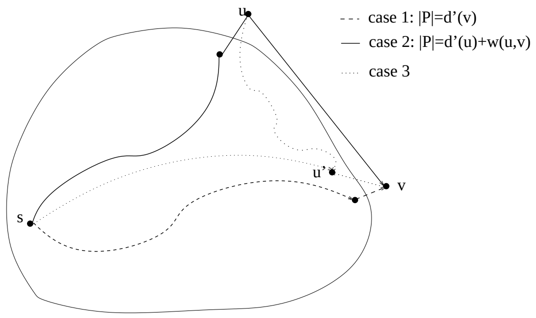


Figure 8.5: Maintaining LI2 in Dijkstra's Algorithm.

- Consider a path P that (1) goes from s to v and (2) is in $\mathcal{P}'$, i.e., was handled before entering the loop. By LI2', $d'(v) \leq |P|$. Hence, $d''(v) = \min\{d'(v), d'(u) + w_{\langle u,v\rangle}\} \leq d'(v) \leq |P|$.
- Consider a path P that starts at s, visits any number of previously handled nodes, visits the newly handled node u, and then follows the last edge $\langle u, v\rangle$ to v. Here $d''(v) = \min\{d'(v), d'(u) + w_{\langle u,v\rangle}\} \leq d'(u) + w_{\langle u,v\rangle} \leq |P|$.
- There is yet one other type of newly handled paths in P''. Consider a path P that starts at s, visits any number of previously handled nodes, visits the newly handled node u, visits some more previously handled nodes, and then follows one last edge to v. Let u' be the node that occurs in P just before v. Because u' is previously handled, we know that there is a shortest path from s to u' that does not pass through node u. A path that is at least as good as P takes this more direct root to u' and then takes the edge $\langle u', v\rangle$. Such paths have been considered already.

Initial Code (i.e., $\langle pre \rangle \rightarrow \langle LI \rangle$): The initial code is the same as that for the previous shortest path (number of edges) algorithm. I.e., s is found but not handled with $d(s) = 0$, $\pi(s) = \epsilon$. Initially no paths to v have been *handled* and hence the length of the shortest handled path to v is $d(v) = \infty$. This satisfies all three loop invariants.

Exiting Loop (i.e., $\langle LI \rangle$ & $\langle exit \rangle \rightarrow \langle post \rangle$): See the shortest paths algorithm above.

Running Time: As I said, the general search algorithm takes time $\mathcal{O}(|E|)$, where E are the edges in G. This algorithm is the same, except for handling the priority queue. A node u is removed from the priority queue $|V|$ times and a $d(v)$ value is updated at most $|E|$ times, once for each edge. Section 6.1 covered how priority queues can be implemented using a heap so that deletions and updates each take $\Theta(\log(\text{size of the priority queue}))$ time. Hence, the running time of this algorithm is $\Theta(|E| \log(|V|))$.

8.4 Depth-First Search

The two classic search orders are breadth first and depth first. We have considered breadth-first search that first visits nodes at distance 1 from s, then those at distance 2, and so on. We will now consider a *depth-first search*, which continues to follow some path as deeply as possible into the graph before it is forced to backtrack.

Changes to the Basic Search Algorithm: The next node u we handle is the one most recently found. $foundNotHandled$ will be implemented as a stack. At each iteration, we pop the most recently pushed node and handle it. Try this out on a graph (or on a tree). The pattern in which nodes are found consists of a single path with single edges hanging off it. See Figure 8.6a.

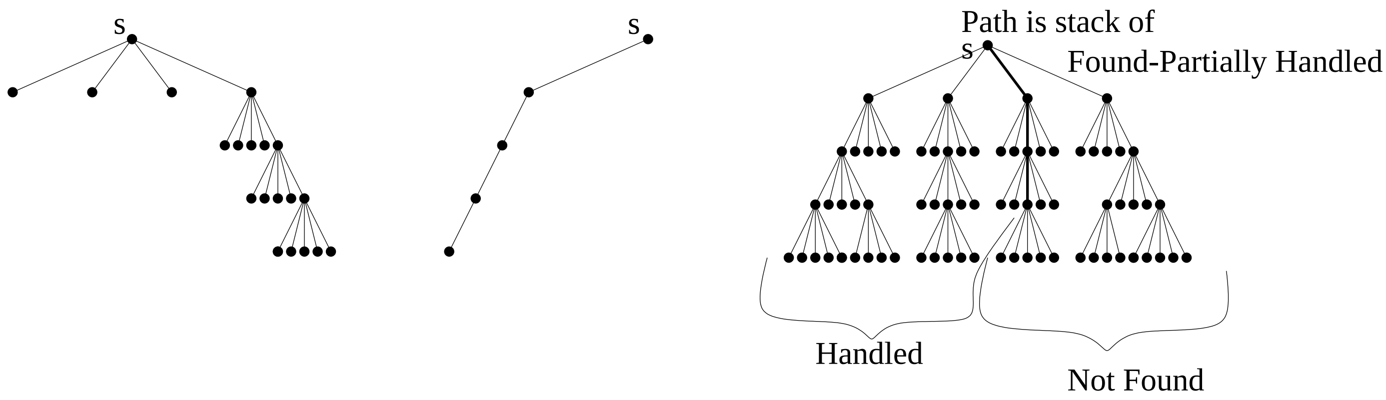


Figure 8.6: If the next node in the stack was completely handled, then the initial order in which nodes are found is given in (a). If the next node is only partially handled, then this initial order is given in (b). Clearly, (b) is a more useful order. This is why the algorithm is designed this way. (c) presents more of the order in which the nodes are found. Though the input graph may not be a tree, presented are only the tree edges given by π.

In order to prevent the single edges hanging off the path from being searched, a second change is made to the original searching algorithm: we no longer completely handle one node before we start handling edges from other nodes. From s, an edge is followed to one of its neighbors v_1. Before visiting the other neighbors of s, the current path to v_1 is extended to $v_2,\ v_3, \ldots$. (See Figure 8.6b.) We keep track of what has been handled by storing an integer i_u for each node u. We maintain that for each u, the first i_u edges of u have already been handled.

$foundNotHandled$ will be implemented as a stack of tuples $\langle v, i_v \rangle$. At each iteration, we pop the most recently pushed tuple $\langle u, i_u \rangle$ and handle the $(i_u+1)^{st}$ edge from u. Try this out on a graph (or on a tree). Figure 8.6c shows the pattern in which nodes are found.

Loop Invariants:

LI1: The nodes in the stack $foundNotHandled$ are ordered such that they define a path starting at s.

LI2: $foundNotHandled$ is a stack of tuples $\langle v, i_v \rangle$ such that for each v, the first i_v edges of v have been handled. (Each node v appears no more than once.)

Code:

algorithm $DepthFirstSearch\,(G, s)$

$\langle$***pre−cond***$\rangle$**:** G is a (directed or undirected) graph and s is one of its nodes.

$\langle$***post−cond***$\rangle$**:** The output is a depth-first search tree of G rooted at s.

begin
    $foundHandled = \emptyset$
    $foundNotHandled = \{\langle s, 0\rangle\}$
    loop
        $\langle$***loop−invariant***$\rangle$**:** See above.
        exit when $foundNotHandled = \emptyset$
        pop $\langle u, i\rangle$ off the stack $foundNotHandled$
        if u has an $(i{+}1)^{st}$ edge $\langle u, v\rangle$
            push $\langle u, i+1\rangle$ onto $foundNotHandled$
            if v has not previously been found then
                $\pi(v) = u$
                $\langle u, v\rangle$ is a tree edge
                push $\langle v, 0\rangle$ onto $foundNotHandled$
            else if v has been found but not completely handled then
                $\langle u, v\rangle$ is a back edge
            else (v has been completely handled)
                $\langle u, v\rangle$ is a forward or cross edge
            end if
        else
            move u to $foundHandled$
        end if
    end loop
    return $foundHandled$
end algorithm

Example:

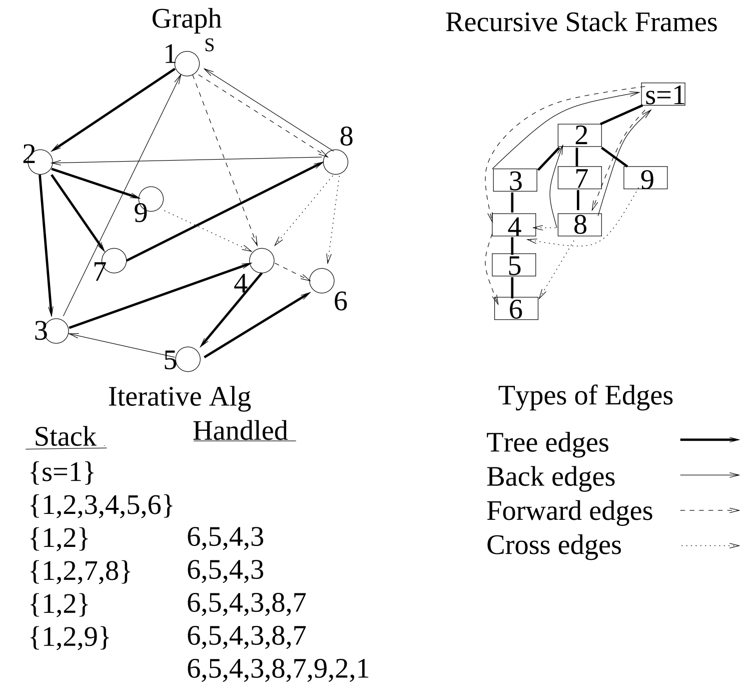


Figure 8.7: Depth-First Search of a Graph. The numbers give the order in which the nodes are found. The contents of the stack are given at each step.

Establishing and Maintaining the Loop Invariant: It is easy to see that with $foundNotHandled = \{\langle s, 0\rangle\}$, the loop invariant is established. If the stack does contain a path from s to u and u has an unhandled edge to v, then u is kept on the stack and v is pushed on top. This extends the path from u onward to v. If u does not have an unhandled edge, then u is popped off the stack. This decreases the path from s by one.

Classification of Edges: The depth-first search algorithm can be used to classify edges.

Tree Edges: Tree edges are the edges $\langle u, v\rangle$ in the depth-first search tree. When such edges are handled, v has not yet been found.

Back Edges: Back edges are the edges $\langle u, v\rangle$ such that v is an ancestor of u in the depth-first search tree. When such edges are handled, v is in the stack and is found but not completely handled.

Cyclic: A graph is cyclic if and only if it has a back-edge.
Proof ($\Leftarrow$): The loop invariant of the depth-first search algorithm ensures that the contents of the stack forms a path from s through v and onward to u. Adding on the edge $\langle u, v\rangle$ creates a cycle back to v.
Proof ($\Rightarrow$): Later we prove that if the graph has no back edges then there is a total ordering of the nodes respecting the edges and hence the graph has no cycles.

Bipartite: A graph is bipartite if and only if there is no back-edge between any two nodes with the same level-parity, i.e., iff it has no odd length cycles.

Forward Edges and Cross Edges: Forward edges are the edges $\langle u, v\rangle$ such that v is a descendent of u in the depth-first search tree.

Cross edges $\langle u, v\rangle$ are such that u and v are in different branches of the depth-first search tree (i.e. are neither ancestors or descendents of each other) and v's branch is traversed before (to the "left" of) u's branch.

When forward edges and cross edges are are handled, v has been completely handled. The depth-first search algorithm does not distinguish between forward edges and cross edges.

Exercise 8.4.1 *Prove that when doing DFS on undirected graphs there are never any forward or cross edges.*

Time Stamping: Some implementations of depth-first search time stamp each node u with a start time $s(u)$ and a finish time $f(u)$. Here "time" is measured by starting a counter at zero and incrementing it every time a node is found for the first time or a node is completely handled. $s(u)$ is the time at which node u is first found and $f(u)$ is the time at which it is completely handled. The time stamps are useful in the following way. (Some texts use $d(u)$ instead of $f(u)$. We, however, prefer to reserve $d(u)$ to be the distance from s.)

- v is descendent of u if and only if the time interval $[s(v), f(v)]$ is completely contained in $[s(u), f(u)]$.
- If u and v are neither ancestor or descendent of each other, then the time intervals $[s(u), f(u)]$ and $[s(v), f(v)]$ are completely disjoint.

Using the time stamps, this can be determined in constant time.

A Recursive Implementation: There is a recursive implementation of this depth-first search algorithm that is easier to understand (assuming that you understand recursion). See Section 15.3.1. In fact, recursion itself is implemented by a stack. See Section 11.1.5. Hence, any recursive program can be converted into an iterative algorithm that uses a stack as done above. (In fact most compliers do this automatically).

8.5 Linear Ordering of a Partial Order

Finding a linear order consistent with a given partial order is one of many applications of a depth-first search. Hint: If a question ever mentions that a graph is acyclic always start by running this algorithm.

Total and Partial Orders:

Def^n: A *total order* of a set of objects V specifies for each pair of objects $u, v \in V$ either (1) that u is *before* v or (2) that v is *before* u. It must be *transitive*, in that if u is before v and v is before w, then u is before w.

Def^n: A *partial order* of a set of objects V supplies only some of the information of a total order. For each pair of objects $u, v \in V$, it specifies either that u is before v, that v is before u, or that the order of u and v is undefined. It must also be transitive.

For example, you must put on your underwear before your pants, and you must put on your shoes *after* both your pants and your socks. According to transitivity, this means you must put your underwear on *before* your shoes. However, you do have the freedom to put your underwear and your socks on in either order. My son, Josh, when six mistook this partial order for a total order and refuses to put on his socks before his underwear. When he was eight, he explained to me that the reason that he could get dressed faster than me was that he had a "short-cut", consisting of putting his socks on before his pants. I was thrilled that he had at least partially understood the idea of a partial order.

```
underwear
     \
    pants   socks
       \   /
       shoes
```

A partial order can be represented by a directed acyclic graph G. The vertices consist of the objects V, and the directed edge $\langle u, v \rangle$ indicates that u is before v. It follows from transitivity that if there is a directed path in G from u to v, then we know that u is before v. A cycle in G from u to v and back to u presents a contradiction because u cannot be both before *and* after v.

Specifications of the Problem: *Topological Sort*:

Preconditions: The input is a directed acyclic graph G representing a partial order.

Postconditions: The output is a total order consistent with the partial order given by G, i.e., $\forall$ edges $\langle u, v \rangle \in G$, i appears before v in the total order.

An Easy but Slow Algorithm:

The Algorithm: Start at any node v of G. If v has an outgoing edge, walk along it to one of its neighbors. Continue walking until you find a node t that has no outgoing edges. Such a node is called a *sink*. This process cannot continue forever because the graph has no cycles.

The sink t can go after every node in G. Hence, you should put t last in the total order, delete t from G, and recursively repeat the process on $G - v$.

Running Time: It takes up to n time to find the first sink, $n - 1$ to find the second, and so on. The total time is $\Theta(n^2)$.

Algorithm Using a Depth-First Search:

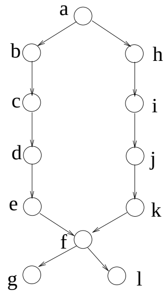
The Algorithm: Start at any node s of G. Do a depth-first search starting at node s. After this search completes, nodes that are considered found will *continue* to be considered found, so should not be considered again. Let s' be any unfound node of G. Do a depth-first search starting at node s'. Repeat the process until all nodes have been found.

Use the time stamp $f(u)$ to keep track of the order in which nodes are *completely handled*, i.e., removed from the stack. Output the nodes in the *reverse* order.

If you ever find a back edge, then stop and report that the graph has a cycle.

Proof of Correctness:

Lemma: For every edge $\langle u, v \rangle$ of G, node v is completely handled before u.



Stack	Handled
{d}	
{d,e,f,g}	
{d,e,f}	g
{d,e,f,l}	g
{}	g,l,f,e,d
{i}	g,l,f,e,d
{i,j,k}	g,l,f,e,d
{}	g,l,f,e,d,k,j,i
{a}	g,l,f,e,d,k,j,i
{a,b,c}	g,l,f,e,d,k,j,i
{a}	g,l,f,e,d,k,j,i,c,b
{a,h}	g,l,f,e,d,k,j,i,c,b
{}	g,l,f,e,d,k,j,i,c,b,h,a

(Topplogical Sort) = a,h,b,c,i,j,k,d,e,f,l,g

Figure 8.8: A Topological Sort Is Found Using a Depth-First Search.

Exercise 8.5.1 *Prove that the lemma is sufficient to prove that the reverse order to that in which the nodes were completely handled is a correct topological sort.*

Proof of Lemma: Consider some edge $\langle u, v\rangle$ of G. Before u is completely handled, it must be put onto the stack $foundNotHandled$. At this point in time, there are three cases:

Tree Edge: v has not yet been found. Because u has an edge to v, v is put onto the top of the stack above u *before* u has been completely handled. No more progress will be made towards handling u until v has been completely handled and removed from the stack.

Back Edge: v has been found, but not completely handled, and hence is on the stack somewhere below u. Such an edge is a back edge. This contradicts the fact that G is acyclic.

Forward or Cross Edge: v has already been completely handled and removed from the stack. In this case, we are done: v was completely handled before u.

Running Time: As with the depth-first search, no edge is followed more than once. Hence, the total time is $\Theta(|E|)$.

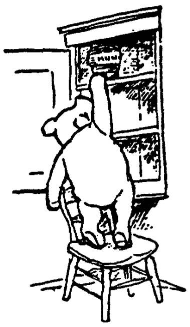
Shortest-Weighted Path: Suppose you want to find the shortest-weighted path for a graph G that you know is acyclic. You could use Dijkstra's algorithm from Section 8.3. However, as hinted above, when ever a question mentions that a graph is acyclic always start by finding a linear order consistent with the edges of the graph. Once this has been completed, you can handle the nodes (as done in Dijkstra's algorithm) in this linear order of the nodes.

Proof of Correctness: The shortest path to node v will not contain any nodes u that appear after it in the total order because by the requirements of the total order there is not path from u to v. Hence, it is fine to handle v, committing to a shortest path to v, before considering node u. Hence, it is fine to handle the nodes in the order given by the total order.

Running Time: The advantage of this algorithm is that you do not need to maintain a priority queue, as done in Dijkstra's algorithm. This decreases the time from $\Theta(|E| \log |V|)$ to $\Theta(|E|)$.

Chapter 10

Greedy Algorithms



Every two year old knows the greedy algorithm. In order to get what you want, just start grabbing what looks best.

10.1 The Techniques and the Theory

Optimization Problems: An important and practical class of computational problems is referred to as *optimization problems*. For most such problems, the fastest known algorithms run in exponential time. (There may be faster algorithms; we simply do not know.) For most of those that have polynomial time algorithms, the algorithm is either greedy, recursive back tracking, dynamic programming, network flow, or linear programming. (See later chapters.)

Most of the optimization problems that are solved using the greedy method have the following form. (A more complete definition of an optimization problem is given in Section 15.1.1.)

Instances: An instance consists of a set of objects and a relationship between them. Think of the objects as being prizes that you must choose between.

Solutions for Instance: A solution requires the algorithm to make a choice about each of the objects in the instance. Sometimes, this choice is more complex, but usually it is simply whether or not to keep it. In this case, a solution is the subset of the objects that you have kept. The catch is that some subsets are not allowed because these objects conflict somehow with each other.

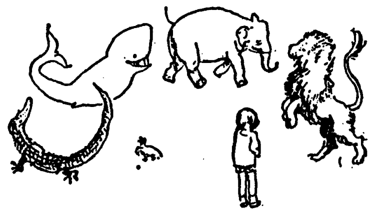
Cost of Solution: Each non-conflicting subset of the objects is assigned a cost. Often this cost is the number of objects in the subset or the sum of the costs of its individual objects. Sometimes the cost is a more complex function of the subset.

Goal: Given an instance, the goal is to find one of the valid solutions for this instance with optimal (minimum or maximum as the case may be) cost. (The solution to be outputted might not be unique.)

The Brute Force Algorithm (Exponential Time): The brute force algorithm for an optimization problem considers each possible solution for the given instance, computes its cost, and outputs the cheapest. Because each instance has an exponential number of solutions, this algorithm takes exponential time.

The Greedy Choice: The greedy step is the first that would come to mind when designing an algorithm for a problem. Given the set of objects specified in the input instance, the greedy step chooses and commits to one of these objects because, according to some simple criteria, it seems to be the "best". When proving that the algorithm works, we must be able to prove that this locally greedy choice does not have negative global consequences.

The Game Show Example: Suppose the instance specifies a set of prizes and an integer m and allows you to choose m of the prizes. The criteria, according to which some of these prizes appear to be "better" than others, may be its dollar price, the amount of joy it would bring you, how practical it is, or how much it would impress the neighbors. At first it seem obvious that you should choose your first prize using the greedy approach. However, some of these prizes conflict with each other and as is often the case in life there are compromises that need to be made. For example, if you take the pool, then your yard is too full to be able to take many of the other prizes. Or if you take the lion, then are many other animals that would not appreciate living together with it. As is also true in life, it is sometimes hard to look into the future and predict the ramifications of the choices made today.



Making Change Example: The goal of this optimization problem is to find the minimum number of quarters, dimes, nickels, and pennies. that total to a given amount. The above format states that an instance consists of a set objects and a relationship between them. Here, the set is a huge pile of coins and the relationship is that the chosen coins must total to the given amount. The cost of a solution, which is to be minimized, is the number of coins in the solution.

The Greedy Choice: The coin that appear to be best to take is a quarter, because it makes the most progress towards making our required amount while only incurring a cost of one.

A Valid Choice: Before committing to a quarter, we must make sure that it does not conflict with the possibility of arriving at a valid solution. If the sum to be obtained happens to be less than \$0.25, then this quarter should be rejected, even though it appears at first to be best. On the other hand, if the amount is at least \$0.25, then we can commit to the quarter without invalidating the solution we are building.

Leading to an Optimal Solution: A much more difficult and subtle question is whether or not committing to a quarter leads us towards obtaining an optimal solution. In this case, it happens that it does, though this not at all obvious.

Going Wrong: Suppose that the previous Making Change Problem is generalized to include as part of the input the set of coin denominations available. This problem is identical to Integer-Knapsack Problem given in Section 16.3.4. With general coin denominations, the greedy algorithm does not work. For example, suppose we have 4, 3, and 1 cent coins. If the given amount is 6, than the optimal solution contains two 3 cent coins. One goes wrong by greedily committing to a 4 cent coin.

Exercise 10.1.1 *What restrictions on the coin denominations ensure that the greedy algorithm works?*

Have Not Gone Wrong: Committing to the pool, to the lion, or to the 4 cent coin in the previous examples, though they locally appear to be the "best" objects, do not lead to an optimal solution. However, for some problems and for some definitions of "best", the greedy algorithm does work. Before committing to the seemingly "best" object, we need to prove that we do not go wrong by doing it.

- **"The" Optimal Solution Contains the "Best" Object:** The first attempt at proving this might try to prove that for every set of objects that might be given as an instance, the "best" of these objects is definitely in its optimal solution. The problem with this is that there may be more than one optimal solution. It might not be the case that all of them contain the chosen object. For example, if all the objects were the same, then it would not matter which subset of objects were chosen.
- **At Least One Optimal Solution Remaining:** Instead of requiring all optimal solutions to contain the "best" object, what needs to be proven is that at least one does. The effect of this is that though committing to the "best" object may eliminate the possibility of some of the optimal solutions, it does not eliminate all of them. There is the saying, "Do not burn your bridges behind you." The message here is slightly different. It is ok to burn a few of your bridges as long as you do not burn all of them.

The Second Step: After the "best" object has been chosen and committed to, the algorithm must continue and choose the remaining objects for the solution. There are two different abstractions within which one can think about this process, iterative and recursive. Though the resulting algorithm is (usually) the same, having the different paradigms at your disposal can be helpful.

- **Iterative:** In the iterative version, there is a main loop. Each iteration, the "best" is chosen from amongst the objects that have not yet been considered. The algorithm then commits to some choice about this object. Usually, this involves deciding whether to commit to putting this chosen object in the solution or to commit to rejecting it.
  - **A Valid Choice:** The most common reason for rejecting an object is that it conflicts with the objects committed to previously. Another reason for rejecting an object is that the object fills no requirements that are not already filled by the objects already committed to.
  - **Cannot Predict the Future:** At each step, the choice that is made can depend on the choices that were made in the past, but it cannot depend on the choices that will be made in the future. Because of this, no back tracking is required.
  - **Making Change Example:** The greedy algorithm for finding the minimum number of coins summing to a given amount is as follows. Commit to quarters until the next quarter increases your current sum above the required amount. Then reject the remain quarters. Then do the same with the dimes, the nickels, and pennies.
- **Recursive:** A recursive greedy algorithm makes a greedy first choice and then recurses once or twice in order to solve the remaining subinstance.
  - **Making Change Example:** After committing to a quarter, we could subtract \$0.25 from the required amount and ask a friend to find the minimum number of coins to make this new amount. Our solution, will be his solution plus our original quarter.
  - **Binary Search Tree Example:** The recursive version of a greedy algorithm is more useful when you need to recurse more than once. For example, suppose you want to construct a binary search tree for a set of keys that minimizes the total height of the tree, i.e. a balanced tree. The greedy algorithm will commit to the middle key being at the root. Then it will recurse once for the left subtree and once for the right.

  To learn more about how to recurse after the greedy choice has been made see recursive back-tracking algorithms in Section 15.

Proof of Correctness: Greedy algorithms themselves are very easy to understand and to code. If your intuition is that it should not work, then your intuition is correct. For most optimization search problems, all greedy algorithms tried do not work. By some miracle, however, for some problems there is a greedy algorithm that works. The proof that they work, however, is very subtle and difficult. As with all iterative algorithms, we prove that it works using loop invariants.

The Loop Invariant: The loop invariant maintained is that we have not gone wrong. There is at least one optimal solution consistent with the choices made so far, i.e. containing the objects committed to so far and not containing the objects rejected so far.

Three Players: To help understand this proof, we will tell a story involving three characters: the algorithm, the prover, and a fairy god mother. Having these three helps us keep track of what each does, knows, and is capable of. Using the analogy of a relay race used to describe iterative algorithms in Section 3.2.1, we could consider a separate algorithm executor, prover, a fairy god mother for each iteration of the algorithm. Doing this helps us keep track of what information is passed by each of these players from one iteration to the next.

The Algorithm: At the beginning of an iteration, the algorithm has a set *Commit* of objects committed to so far and the set of jobs rejected so far. The algorithm then chooses the "best" object from amongst those not considered so far and either commits to it or rejects it.

The Prover: At the beginning of an iteration, the prover knows that the loop invariant is true. The job of the prover is to make sure that it has been maintained when the algorithm commits to or rejects the next best object. We separate his role from that of the algorithm to emphasize that his actions are not a part of the algorithm and hence do not need to be coded or executed. By the loop invariant, the prover knows that there is at least one optimal solution consistent with the choices made by the algorithm so far. He, however, does not know any of them. If, he did, then perhaps the algorithm could too, and we would not need to be going through all of this work. On the other hand, as part of a thought experiment, the prover does have a fairy god mother to "help" him.

The Fairy God Mother: At the beginning of an iteration, the fairy god mother is holding for the prover one of the optimal solutions that is known to exist. This solution is said to *witness* to the fact such a solution exists. Though the fairy god mother is all powerful, practically speaking she is not all that helpful, because she is unable to communicate anything back to either the prover or the algorithm. The prover does, however, receive moral support by speaking in a one way communication to her.

Initially (i.e., $\langle pre \rangle \rightarrow \langle LI \rangle$): Initially, the algorithm has made no choices, neither committing to nor rejecting any objects. The prover then establishes the loop invariant as follows. Assuming that there is at least one legal solution, he knows that there must be an optimal solution. He goes on to note that this optimal solution by default is consistent with the choices made so far, because no choices have been made so far. Knowing that such a solution exists, the prover kindly asks his fairy good mother to find one. She being all powerful has no problem doing this. If there are more than one equally good optimal solutions, then she chooses one arbitrarially.

Maintaining the Loop Invariant (i.e., $\langle LI' \rangle$ & not $\langle exit \rangle$ & $code_{loop} \rightarrow \langle LI'' \rangle$): Now consider an arbitrary iteration.

What We Know: At the beginning of this iteration, the algorithm has a set *Commit* of objects committed to so far and the set of jobs rejected so far. The prover knows that the loop invariant is true, i.e. that there is at least one optimal solution consistent with these choices made so far. Witnessing this fact, the fairy god mother is holding one such optimal solution. We will use $optS_{LI}$ to denote the solution that she holds. In addition to containing those objects in *Commit* and not those in *Reject*, this solution may contain objects that the algorithm has not considered yet. Neither the algorithm nor the prover know what these objects are.

Taking a Step: During the iteration, the algorithm proceeds to choose the "best" object from amongst those not considered so far and either commits to it or rejects it. In order to prove that the loop invariant has been maintained, the prover must prove that there is at least one optimal solution consistent with both the choices made previously and with this new choice. He is going to accomplish this by getting his fairy good mother to witness this fact by switching to such an optimal solution.

Weakness in Communication: It would be great if the prover could simply ask the fairy god mother whether such a solution exists. However, she is unable to reply to his questions. More over he cannot ask her to find such a solution if he is not already confident that it exists, because he does not want to ask her to do anything that is impossible.

Massage Instructions: The prover accomplishes his task by giving his fairy god mother detailed instructions. He starts by saying, "If it happens to be the case that the optimal solution that you hold is consistent with this new choice that was made, then we are done, because this will witness the fact that there is at least one optimal solution consistent with both the choices made previously and with this new choice." " Otherwise," he says, "you must massage (modify) the optimal solution that you have in the following ways." The fairy god mother follows the detailed instructions that he gives her, but of course gives him no feed back as to how they go. We will use $optS_{ours}$ to denote what she constructs.

Making Change Example: If the remaining amount required is at least \$0.25, then the algorithm commits to another quarter. The prover must prove that there exists an optimal solution consistent with all the choices made. His fairy god mother has an optimal solution $optS_{LI}$ that contains all the coins committed to so far. He considers the following cases. If this solution, happens to contain the newly committed to quarter, then he is done. If it contains another quarter (other than those committed to previously), but not the exact quarter that the algorithm happened to commit to, then though what his fairy god mother holds is a perfectly good optimal solution, it is not exactly consistent with the choices made by the algorithm. The prover instructs his fairy god mother that if this case arises to kindly swap the extra quarter that she has for the quarter that the algorithm has. In another case, $optS_{LI}$ does not contain an additional quarter at all. The prover proves in this case that in order to make up the required remaining amount, $optS_{LI}$ must either contains three dimes, two dimes and a nickel, one dime and three nickels, five nickels, or combinations with at least five pennies. He tells her that in such cases she must replace the three dimes with the newly committed to quarter and a nickel and the other options with just the quarter. Note that the prover gives these instructions without gaining any information. There is another case to consider. If the remaining amount required is less than \$0.25, then the algorithm rejects the next (and later all remaining) quarters. The prover is confident that optimal solution held by his fairy god mother cannot contain additional quarters either, so he knows he is safe.

Proving That She Has A Witness: It is the job of the prover to prove that the thing $optS_{ours}$ that his fairy god mother now holds is a valid, consistent, and optimal solution.

Proving A Valid Solution: First he must prove that what she now holds is a valid solution. Because he knows that what she had been holding, $optS_{LI}$, at the beginning of the iteration was a valid solution, he know that the objects in it did not conflict in any way. Hence, all he needs to do is to prove that he did not introduce any conflicts that he did not fix.

Making Change Example: The prover was careful that the changes he made did not change the total amount that she was holding.

Proving Consistent: He must also prove that the solution she is now holding is consistent with both the choices made previously by the algorithm and with this new choice. Because he knows that what she had been holding was consistent with the previous choices, he need only prove that he modified it to be consistent with the new choices without messing this earlier fact.

Making Change Example: Though the prover may have removed some of the coins that the algorithm has not considered yet, he was sure not to have her remove any of the previously committed to coins. He also managed to add the newly committed to quarter.

Proving Optimal: The prover must also prove that the solution, $optS_{ours}$, she holds is optimal. One might think that proving it is optimal would be hard, given that we do not even know the cost of an optimal solution. However, the prover can be assured that it is optimal as long as its cost is the same as the optimal solution, $optS_{LI}$, from which it was derived. If there is case in which prover manages to improve the solution, then this contradicts the fact that $optS_{LI}$ is optimal. This contradiction only proves that such a case will not occur. However, the prover does not need to concern himself with this problem.

Making Change Example: Each change that the prover instructs his fairy god mother to make either keeps the number of coins the same or decreases the number. Hence, because $optS_{LI}$ is optimal, $optS_{ours}$ is as well.

This completes the provers proof that his fairy good mother now has an optimal solution consistent with both the previous choices and with the latest choice. This witnesses the fact that such a solution exists.

This proves that the loop invariant has been maintained.

Continuing: This completes everybody's requirements for this iteration. This is all repeated over and over again. Each iteration, the algorithm commits to more about the solution and the fairy god mother's solution is changed to be consistent with these commitments.

Exiting Loop (i.e., $\langle LI \rangle$ & $\langle exit \rangle \rightarrow \langle post \rangle$): After the algorithm has considered every object in the instance and each has either been committed to or rejected, the algorithm exits. We still know that the loop invariant is true. Hence, the prover knows that there is an optimal schedule $optS_{LI}$ consistent with all of these choices. Previously, this optimal solution was only imaginary. However, now we concretely know what this imagined solution is. It must consist of those objects committed to. Hence, the algorithm can return this set as the solution.

Running Time: Greedy algorithms are very fast because they take only a small amount of time per object in the instance.

Fixed vs Adaptive Priority: Iterative greedy algorithms come in two flavors, *fixed priority* and *adaptive priority*.

Fixed Priority: A fixed priority greedy algorithm begins by sorting the objects in the input instance from best to worst according to a fixed greedy criteria. For example, it might sort the objects based on the cost of the object or the arrival time of the object. The algorithm then considers the objects one at a time in this order.

Adaptive Priority: In an adaptive priority greedy algorithm, the greedy criteria is not fixed, but depends on which objects have been committed to so far. At each step, the next "best" object is chosen according to the current greedy criteria. Blindly searching the remaining list of objects each iteration for the next best object would be too time consuming. So would re-sorting the objects each iteration according to the new greedy criteria. A more efficient implementation uses a priority queue to hold the remaining objects prioritized according to the current greedy criteria. This can be implemented using a Heap. (See Section 6.1.)

Code:

algorithm *AdaptiveGreedy*(set of objects)

$\langle pre-cond \rangle$: The input consists of a set of objects.

$\langle post-cond \rangle$: The output consists of an optimal subset of them.

begin

  Put objects in a Priority Queue according to the initial greedy criteria

  $Commit = \emptyset$ % set of objects previously committed to

  loop

    $\langle loop-invariant \rangle$: See above.

    exit when the priority queue is empty

    Remove "best" object from priority queue

If this object does not conflict with those in *Commit* and is needed then
Add object to *Commit*
end if
Update the priority queue to reflect the new greedy criteria.
This is done by changing the priorities of the objects effected.
end loop
return (*Commit*)
end algorithm

Example: Dijkstra's shortest weighted path algorithm can be considered to be a greedy algorithm with an adaptive priority criteria. See Section 8.3. It chooses the next edge to include in the optimal *shortest-weighted-paths tree* based on which node currently seems to be the closest to s. Those yet to be chosen are organized in a priority queue. Even Breadth-First and Depth-First Search can be considered to be adaptive greedy algorithms. In fact, they very closely resembles Prim's Minimal Spanning Tree algorithm, Section 10.2.3, in how a tree is grown from a source node. They are adaptive because as the algorithm proceeds, the set from which the next edge is chosen changes.

10.2 Examples of Greedy Algorithms

This completes the presentation of the general techniques and the theory behind greedy algorithms. We now will provide a few examples.

10.2.1 A Fixed Priority Greedy Algorithm for The Job/Event Scheduling Problem

Suppose that many people want to use your conference room for events and you must schedule as many of these as possible. (The version in which some events are given a higher priority is considered in Section 16.3.3.)

The Event Scheduling Problem:

Instances: An instance is $\langle\langle s_1, f_1\rangle, \langle s_2, f_2\rangle, \ldots, \langle s_n, f_n\rangle\rangle$, where $0 \leq s_i \leq f_i$ are the starting and finishing times for the i^{th} event.

Solutions: A solution for an instance is a schedule S. This consists of a subset $S \subseteq [1..n]$ of the events that don't conflict by overlapping in time.

Cost of Solution: The cost $C(S)$ of a solution S is the number of events scheduled, i.e., $|S|$.

Goal: Given a set of events, the goal of the algorithm is to find an *optimal solution*, i.e., one that maximizes the number of events scheduled.

Possible Criteria for Defining "Best":

The Shortest Event $f_i - s_i$: It seems that it would be best to schedule short events first because they increase the number of events scheduled without booking the room for a long period of time. This greedy approach does not work.

Counter Example: Suppose that the following lines indicate the starting and completing times of three events to schedule.

The only optimal schedule includes the two long events. There are no optimal schedules containing the short event in the middle, because both the other two events conflict with it. Hence, if in a greedy way we committed to scheduling this short event, then we would be going wrong. The conclusion is that using the shortest event as a criteria does not work as a greedy algorithm.

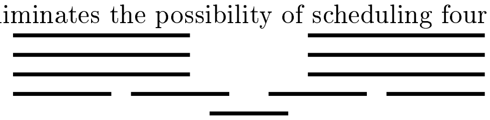
The Earliest Starting Time s_i or the Latest Finishing Time f_i: First come first serve, which is a common scheduling algorithm, does not work either.

Counter Example:

The long event is both the earliest and the latest. Committing to scheduling it would be a mistake.

Event Conflicting with the Fewest Other Events: Scheduling an event that conflicts with other events prevents you from scheduling these events. Hence a reasonable criteria would be to first schedule the event with the fewest conflicts.

Counter Example: In the following example, the middle event would be committed to first. This eliminates the possibility of scheduling four events.



Earliest Finishing Time f_i: This criteria may seem a little odd at first, but it too makes sense. It says to schedule the event who will free up your room for someone else as soon as possible. We will see that this criteria works for every set of events.

Exercise 10.2.1 *See how it works on the above three examples.*

Code: The resulting greedy algorithm for the Event Scheduling problem is as follows:

algorithm $Scheduling\left(\langle\langle s_1, f_1\rangle, \langle s_2, f_2\rangle, \ldots, \langle s_n, f_n\rangle\rangle\right)$

$\langle$***pre−cond***$\rangle$**:** The input consists of a set of events.

$\langle$***post−cond***$\rangle$**:** The output consists of a schedule that maximizes the number of events scheduled.

```
begin
    Sort the events based on their finishing times f_i
    Commit = ∅       % The set of events committed to be in the schedule
    loop i = 1...n   % Consider the events in sorted order.
        if( event i does not conflict with an event in Commit ) then
            Commit = Commit ∪ {i}
    end loop
    return(Commit)
end algorithm
```

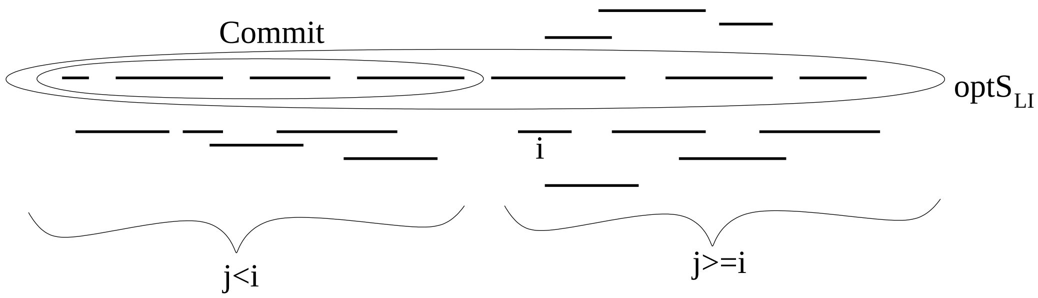


Figure 10.1: A set of events, those committed to at the current point in time, those rejected, those in the optimal solution assumed to exist, and the next event to be considered

The Loop Invariant: The loop invariant is that we have not gone wrong. There is at least one optimal solution consistent with the choices made so far, i.e. containing the objects committed to so far and not containing the objects rejected so far, i.e. the previous events $j < i$ not in $Commit$.

Initial Code (i.e., $\langle pre \rangle \rightarrow \langle LI \rangle$): Assuming that there is at least one solution, there is at least one that is optimal. Initially, no choices have been made and hence trivially this optimal solution is consistent with these choices.

Maintaining the Loop Invariant (i.e., $\langle LI' \rangle$ & not $\langle exit \rangle$ & $code_{loop} \rightarrow \langle LI'' \rangle$): Suppose that LI' which denotes the statement of the loop invariant before the iteration is true, the exit condition $\langle exit \rangle$ is not, and we have executed another iteration of the algorithm.

Hypothetical Optimal Solution: Let $optS_{LI}$ denote one of the hypothetical optimal schedules assumed to exist by the loop invariant.

Event i Not Added: If event i conflicts with an event in $Commit$, then it is not committed to. After going around the loop, event i will also be considered a previous event. Hence, the loop invariant will require (in addition to the previous conditions) that this event is not in the optimal schedule stated to exist. The same optimal schedule $optS_{LI}$ meets this condition. It cannot contain event i because it contains all the events in $Commit$ and event i conflicts with an event in $Commit$. Hence, the loop invariant is maintained.

Event i Is Added: From here on, let us assume that i does not conflict with any event in $Commit$ and hence will be added to it.

At Least One Optimal Solution Left: To prove that the loop invariant is maintained after the loop, we must prove that there is at least one optimal schedule containing the events in $Commit$ and containing i.

Massaging Optimal Solutions: If we are lucky, the schedule $optS_{LI}$ already contains event i. In this case, we are done. Otherwise, we will massage the schedule $optS_{LI}$ into another schedule $optS_{ours}$ containing $Commit \cup \{i\}$ and prove that this is also an optimal schedule. Note that this massaging is NOT part of the algorithm. We do not actually have an optimal schedule yet. We have only maintained the loop invariant that such an optimal schedule exists.

Constructing $optS_{ours}$: There are four types of events to consider.

Event i: Because the new massaged schedule $optS_{ours}$ must contain the current event i and the hypothetical optimal schedule $optS_{LI}$ happens not to contain it, the first step in massaging $optS_{LI}$ into $optS_{ours}$ is simply to add event i.

Events in $Commit$: We know that $optS_{LI}$ already contains the events in $Commit$ because of LI'. In order to maintain this loop invariant, these events must remain in the new schedule $optS_{ours}$. However, this is not a problem because we know that event i does not conflict with any event in $Commit$. Recall that the code tests whether i conflicts with any event in $Commit$.

Past Events not in $Commit$: By LI', we know that the events that were seen in the past yet not added to $Commit$ are not contained in $optS_{LI}$. Neither will they be in $optS_{ours}$.

Future Events: What remains to be considered are the events that are yet to be seen in the future i.e. $j > i$. Though the algorithm has not yet considered them, some of them are likely in the hypothetical optimal schedule $optS_{LI}$, yet we do not know which ones. The issue at hand is determining how adding event i to this schedule changes which of these future events should be scheduled. Adding event i may create conflicts with these future jobs. Such future jobs are deleted from the schedule. In summary, $optS_{ours} = optS_{LI} \cup \{i\} - \{\ j \in optS_{LI} \mid j > i$ and event j conflicts with event $i\}$.

$optS_{ours}$ is a Solution: Our massaged set $optS_{ours}$ contains no conflicts, because $optS_{LI}$ contained none and we were careful to introduce none. Hence, it is a valid solution for our instance.

$optS_{ours}$ is Optimal: To prove that $optS_{ours}$ has the optimal number of events in it, we need only to prove that it has at least as many as $optS_{LI}$. We added one event to the schedule. Hence, we must prove that we have not removed more than one of the future events, i.e. that $|\{\ j \in optS_{LI} \mid j > i$ and event j conflicts with event $i\}| \leq 1$.

Consider some event j in this set. Because the events are sorted based on their finishing time, $j > i$ implies that event j finishes after event i finishes, i.e. $f_j \geq f_i$. If event j conflicts with event

i, it follows that it also starts before it finishes, i.e. $s_j \leq f_i$. (In Figure 10.1, there are three future events conflicting with i.)

Combining $f_j \geq f_i$ and $s_j \leq f_i$, gives that such an event j is running at the finishing time f_i of event i. This completes the proof that there can only be one event j in $optS_{LI}$ with these properties. By way of contradiction, suppose that there are two such events, j_1, j_2. We showed that each of them must be running at time f_i. Hence, they conflict with each other. Therefore, they cannot both be in the schedule $optS_{LI}$ because it contains no conflicts.

Loop Invariant Has Been Maintained: In conclusion, we constructed a legal schedule $optS_{ours}$ that contains the events in $Commit \cup \{i\}$. We proved that it is of optimal size because it has the same size as the optimal schedule $optS_{LI}$. This proves that the loop invariant is maintained because it proves that there is an optimal schedule that contains the events in $Commit \cup \{i\}$.

Exiting Loop (i.e., $\langle LI\rangle$ & $\langle exit\rangle \rightarrow \langle post\rangle$): Suppose that both $\langle LI\rangle$ and $\langle exit\rangle$ are true. We must prove that the events in $Commit$ form an optimal schedule. By LI, there is an optimal schedule $optS_{LI}$ containing the events in $Commit$ and not containing the previous events not in $Commit$. Because all events are previous events, it follows that $Commit = optS_{LI}$ is in an optimal schedule for our instance.

Running Time: The loop is iterated once for each of the n events. The only work is determining whether event i conflicts with a event within $Commit$. Because of the ordering of the events, event i finishes after all the events in $Commit$. Hence, it conflicts with an event in $Commit$ if and only if it starts before the last finishing time of an event in it. It is easy to remember this last finishing time, because it is simply the finishing time of the last event to be added to $Commit$. Hence, the main loop runs in $\Theta(n)$ time. The total time of the algorithm then is dominated by the time to sort the events.

10.2.2 An Adaptive Priority Greedy Algorithm for The Interval Cover Problem

The Interval Cover Problem:

Instances: An instance consists a set of points P and a set of intervals I on the real line. An interval consists of a starting and a finishing time $\langle f_i, s_i\rangle$.

Solutions: A solution for an instance is a subset S of the intervals that covers all the points.

Cost of Solution: The cost $C(S)$ of a solution S is the number of intervals required, i.e., $|S|$.

Goal: The goal of the algorithm is to find an *optimal cover*, i.e., a subset of the intervals that covers all the points and that contains the minimum number of intervals.

The Adaptive Greedy Criteria: The algorithm sorts the points and covers them in order from left to right. If the intervals committed to so far, i.e. those in $Commit$, cover all of the points in P, then the algorithm stops. Otherwise, let P_i denote the left most point in P that is not covered by $Commit$. The next interval committed to must cover this next uncovered point, P_i. Of the intervals that start to the left of the point, the algorithm greedily takes the one that extends as far to the right as possible. The hope in doing so is that the chosen interval, in addition to covering P_i, will cover as many other points as possible. Let I_j denote this interval. If there is no such interval I_j or it does not extend to the right far enough to cover the point P_i, then no interval covers this point and the algorithm reports that no subset of the intervals covers all of the points. Otherwise, the algorithm commits to this interval by adding it $Commit$. Note that this greedy criteria with which to select the next interval changes as the point P_i to be covered changes.

The Loop Invariant: The loop invariant is that we have not gone wrong. There is at least one optimal solution consistent with the choices made so far, i.e. containing the objects committed to so far and not containing the objects rejected so far.

Maintaining the Loop Invariant (i.e., $\langle LI'\rangle$ & not $\langle exit\rangle$ & $code_{loop} \rightarrow \langle LI''\rangle$): Assume that we are at the top of the loop and that the loop invariant is true, i.e. there exists an optimal cover that contains all of the intervals in $Commit$. Let $optS_{LI}$ denote such a cover that is assumed to exist. If we are lucky and $optS_{LI}$ already contains the interval I_j being committed to this iteration, then we automatically know that there exists an optimal cover that contains all of the intervals in $Commit \cup \{I_j\}$ and hence the loop invariant has been maintained. If on the other hand, the interval I_j being committed to is not in $optS_{LI}$, then we must massage this optimal solution into another optimal solution that does contain it.

Massaging $optS_{LI}$ into $optS_{ours}$: The optimal solution $optS_{LI}$ must cover point P_i. Let $I_{j'}$ denote one of the intervals in $optS_{LI}$ that covers P_i. Our solution $optS_{ours}$ is the same as $optS_{LI}$ except $I_{j'}$ is removed and I_j is added. We know that $I_{j'}$ is not in $Commit$, because the point P_i is not covered by $Commit$. Hence as constructed $optS_{ours}$ contains all the intervals in $Commit \cup \{I_j\}$.

$optS_{ours}$ is an Optimal Solution: Because $optS_{LI}$ is an optimal cover, we can prove that $optS_{ours}$ in an optimal cover, simply by proving that it covers all of the points covered by $optS_{LI}$ and that it contains the same number of intervals. (The later is trivial.)

The algorithm considered the point P_i because it is the left most uncovered point. It follows that the intervals in $Commit$ cover all the points to the left of point P_i.

The interval $I_{j'}$, because it covers point P_i, must be one of these intervals that starts to the left of point P_i. The algorithm's greedy choice of intervals chose I_j because of those intervals that start to the left of point P_i, it is the one that extends as far to the right as possible. Hence, I_j extends further to the right than $I_{j'}$ and hence I_j covers as many points to the right as $I_{j'}$ covers. It follows that $optS_{ours}$ covers all of the points covered by $optS_{LI}$.

Because $optS_{ours}$ is an optimal solution containing $Commit \cup \{I_j\}$, we have proved such a solution exists. Hence, the loop invariant has been maintained.

Maintaining the Greedy Criteria: As the point P_i to be covered changes, the greedy criteria according to which the next interval is chosen changes. Blindly searching for the interval that is best according to the current greedy criteria would be too time consuming. The following data structures help to make the algorithm more efficient.

An Event Queue: The progress of the algorithm can be viewed as an *event marker* moving along the real line. An *event* in the algorithm occurs when this marker either reaches the start of an interval or a point to be covered. This is implemented not with a marker, but by *event queue*. The queue is constructed initially by sorting the intervals according to their start time, the points according to their position, and merging these two lists together. The algorithm removes and processes these events one at a time.

Additional Loop Invariants: The following additional loop invariants relate the current position event marker with the current greedy criteria.

LI1 Points Covered: All the points to the left of the event marker have been covered by the intervals in $Commit$.

LI2 The Priority Queue: A priority queue contains all intervals (except possibly those in $Commit$) that start to the left of the event marker. The priority according to which they are organized is how far f_j to the right the interval extends. This priority queue can be implemented using a Heap. (See Section 6.1.)

LI3 Last Place Covered: A variable *last* indicates the right most place covered by an interval in $Commit$.

Maintaining the Additional Loop Invariants:

A Start Interval Event: When the event marker passes the starting time s_j of an interval I_j, this interval from then on will start to the left of the event marker and hence is added to the priority queue with its priority being its finishing time f_j.

An End Interval Event: When the event marker passes the finishing time f_j of an interval I_j, we learn that this interval will not be able to cover future points P_i. Though the algorithm no longer wants to consider this interval, the algorithm will be lazy and leave it in the priority queue. Its priority will be lower than those actually covering P_i. There being nothing useful to do, the algorithm does not consider these events.

A Point Event: When the event marker reaches a point P_i in P, the algorithm uses $last \geq P_i$ to check whether the point is already covered by an interval in *Commit*. If it is covered, then nothing needs to be done. If not, then LI1 assures us that this point is the left most uncovered point. LI2 ensures that the priority queue is organizing the intervals according to the current greedy criteria, namely it contains all intervals that start to the left of the point P_i sorted according to how far to the right the interval extends. Let I_j denote the highest priority interval in the priority queue. Assuming that it cover P_i, the algorithm commits to it. As well, if it covers P_i, then it must extend further to the right than other intervals in *Commit* and hence *last* is updated to f_j. (The algorithm can either remove the interval I_j committed to from the priority queue or not. It will not extend far enough to the right to cover the next uncovered point, and hence its priority will be low in the queue.)

Code:

algorithm $IntervalPointCover\ (P, I)$

$\langle$***pre−cond***$\rangle$**:** P is a set of points and I is a set of intervals on a line.

$\langle$***post−cond***$\rangle$**:** The output consists of the smallest set of intervals that covers all of the points.

```
begin
    Sort P = {P_1, ..., P_n} in ascending order of p_i's.
    Sort I = {⟨s_1, f_1⟩, ..., ⟨s_m, f_m⟩} in ascending order of s_j's.
    Events = Merge(P, I)        % sorted in ascending order
    consideredI = ∅             % the priority queue of intervals being considered
    Commit = ∅                  % solution set: covering subset of intervals
    last = −∞                   % rightmost point covered by intervals in Commit
    for each event e ∈ Events, in ascending order do
        if(e = ⟨s_j, f_j⟩) then
            Insert interval ⟨s_j, f_j⟩ into the priority queue consideredI with priority f_j
        else (e = P_i)
            if(P_i > last) then                          % P_i is not covered by Commit
                ⟨s_j, f_j⟩ = ExtractMax(consideredI)     % f_j is max in consideredI
                if(consideredI was empty or P_i > f_j) then
                    return (P_i cannot be covered)
                else
                    Commit = Commit ∪ {j}
                    last = f_j
                end if
            end if
        end if
    end for
    return (Commit)
end algorithm
```

Running Time: The initial sorting takes $\mathcal{O}((n+m)\log(n+m))$ time. The main loop iterates $n+m$ times, once per event. Since H contains a subset of I, the priority queue operations *Insert* and *ExtractMax* each takes $\mathcal{O}(\log m)$ time. The remaining operations of the loop take $\mathcal{O}(1)$ time per iteration. Hence, the loop takes a total of $\mathcal{O}((n+m)\log m)$ time. Therefore, the running time of the algorithm is $\mathcal{O}((n+m)\log(n+m))$.

10.2.3 The Minimum-Spanning-Tree Problem

Suppose that you are building network of computers. You need to decide between which pairs of computers to run a communication line. You want all of the computers to be connected via the network. You want to do it in a way that minimizes your costs. This is an example of the Minimum Spanning Tree Problem.

Definitions: Consider a subset S of the edges of an undirected graph G.

A Tree: S is said to be *tree* if it contains no cycles and is connected.

Spanning Set: S is said to *span* the graph iff every pair nodes is connected by a path through edges in S.

Spanning Tree: S is said to be a *spanning tree* of G iff it is a tree that spans the graph. Note cycles will cause redundant paths between pairs of nodes.

Minimal Spanning Tree: S is said to be a *minimal spanning tree* of G iff it is a spanning tree with minimal total edge weight.

Spanning Forest: If the graph G is not connected then it cannot have a spanning tree. S is said to be a *spanning forest* of G iff it is a collection of trees that spans each of the connected components of the graph. In other words, pairs of nodes that are connected by a path in G are still connected by a path when considering only the edges in S.

The Minimum Spanning Tree (Forest) Problem:

Instances: An instance consists of an undirected graph G. Each edge $\{u, v\}$ is labeled with a real-valued (possibly negative) weight $w_{\{u,v\}}$.

Solutions: A solution for an instance is a spanning tree (forest) S of the graph G.

Cost of Solution: The cost $C(S)$ of a solution S is sum of its edge weights.

Goal: The goal of the algorithm is to find a minimum spanning tree (forest) of G.

Possible Criteria for Defining "Best":

Cheapest Edge (Kruskal's Algorithm): The obvious greedy algorithm simply commits to the cheapest edge that does not create a cycle with the edges committed to already.

Code: The resulting greedy algorithm is as follows.

```
algorithm KruskalMST (G)
⟨pre−cond⟩: G is an undirected graph.
⟨post−cond⟩: The output consists of a minimal spanning tree.
begin
    Sort the edges based on their weights w_{u,v}.
    Commit = ∅      % The set of edges committed to
    loop i = 1...n  % Consider the edges in sorted order.
        if( edge i does not create a cycle with the edges in Commit ) then
            Commit = Commit ∪ {i}
    end loop
    return(Commit)
end algorithm
```

Checking For a Cycle: One task that this algorithm must be able to do quickly is to determine whether the new edge i creates a cycle with the edges in *Commit*. As a task in itself, this would take a while. However, if we maintain an extra data structure, this this task can be done very quickly.

Connected Components of *Commit*: The edges in *Commit* connect some pairs of nodes with a path and do not connect other pairs. Using this, we can partition the nodes in the graph into sets so that between any two nodes in the same set, *Commit* provides a path between them and between any two nodes in the different sets, *Commit* does not connect them. These sets are referred to as *components* of the subgraph induced by the edges in *Commit*. The algorithm can determine whether the new edge i creates a cycle with the edges in *Commit* by checking whether the endpoints of the edge $i = \{u, v\}$ are contained in the same component. The required operations on components are handled by the following *Union-Find* data structure.

Union-Find: The *Union-Find* data structure maintains a number of disjoint sets of elements and allows three operations: 1) $Makeset(v)$ which creates a new set containing the specified element v; 2) $Find(v)$ which determines the "name" of the set containing a specified element; and 3) $Union(u, v)$ which merges the sets containing the specified elements u and v. On average, for all practical purposes, each of these operations can be competed in a constant amount of time. More formally, the total time to do m of these operations on n elements is $\Theta(m\alpha(n))$ where α is *inverse Ackermann's function*. This function is so slow growing that even if n equals the number of atoms in the universe, then $\alpha(n) \leq 4$.

Code: The union-find data structure is integrated into the MST algorithm as follows.

algorithm $KruskalMST\,(G)$

$\langle$***pre−cond***$\rangle$**:** G is an undirected graph.

$\langle$***post−cond***$\rangle$**:** The output consists of a minimal spanning tree.

```
begin
    Sort the edges based on their weights w_{u,v}.
    Commit = ∅      % The set of edges committed to
    for each v,
        % With no edges in Commit, each node is in a component by itself.
        MakeSet(v)
    end for
    loop i = 1...n   % Consider the edges in sorted order.
        u and v are end points of edge i.
        if( Find(u) ≠ Find(v) ) then
            % The end points of edge i are in different components and hence do
              not create a cycle with edges in Commit.
            Commit = Commit ∪ {i}
            Union(u, v)
        % Edge i connects the two components, hence they
          are merged into one component.
        end if
    end loop
    return(Commit)
end algorithm
```

Running Time: The initial sorting takes $\mathcal{O}((m)\log(m))$ time when G has m edges. The main loop iterates m times, once per edge. Checking for a cycle takes time $\alpha(n) \leq 4$ on average. Therefore, the running time of the algorithm is $\mathcal{O}(m \log m)$.

Cheapest Connected Edge (Prim's Algorithm): The following greedy algorithm expands out a tree of edges from a source node as done in the generic search algorithm of Section 8.1. Each iteration it commits to the cheapest edge of those that expand this tree, i.e., cheapest from amongst those edges that are connected to the tree *Commit* and yet do not create a cycle with it.

Advantage: If you are, for example, trying to find a MST of the world wide web, then you may not know about an edge until you have expanded out to it.

Adaptive: Note that this is an adaptive greedy algorithm. The priorities of the edges change as the algorithm proceeds, not because their weights $w_{\{u,v\}}$ change, but because an edge is not always allowed. The adaptive algorithm maintains a priority queue of the allowed edges with priorities given by their weights $w_{\{u,v\}}$. The edges allowed are those that are connected to the tree *Commit*. When a new edge i is added to *Commit*, the edges connected to i are added to the queue.

Code:

algorithm $PrimMST\,(G)$

$\langle$***pre−cond***$\rangle$**:** G is an undirected graph.

$\langle$***post−cond***$\rangle$**:** The output consists of a minimal spanning tree.

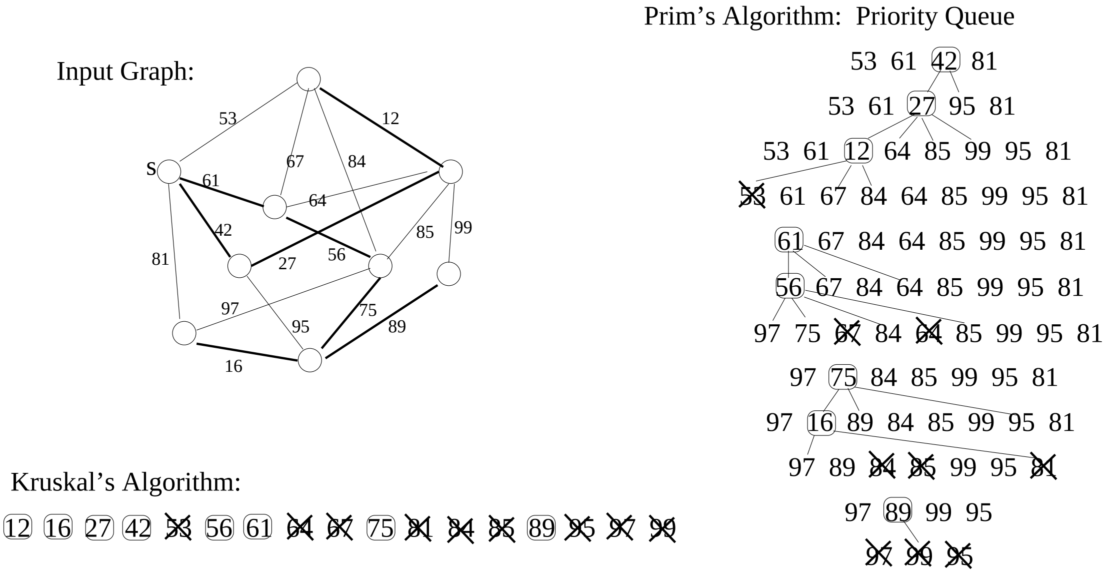


Figure 10.2: Both Kruskal's and Prim's algorithms are run on the given graph. For Kruskal's algorithm the sorted order of the edges is shown. For Prim's algorithm the running contents of the priority queue is shown (edges are in no particular order). Each iteration, the best edge is considered. If it does not create a cycle, it is add to the MST. This is shown by circling the edge weight and darkening the graph edge. For Prim's algorithm, the lines out of the circles indicate how the priority queue is updated. If the best edge does create a cycle, then the edge weight is Xed.

begin
 Let s be the start node in G.
 $Commit = \emptyset$ % The set of edges committed to
 $Queue =$ edges adjacent to s % Priority Queue
 Loop until $Queue = \emptyset$
  $i =$ cheapest edge in $Queue$
  if(edge i does not create a cycle with the edges in $Commit$) then
   $Commit = Commit \cup \{i\}$
   Add to $Queue$ edges adjacent to edge i that have not been added before
  end if
 end loop
 return($Commit$)
end algorithm

Checking For a Cycle: Because one tree is grown, like in the generic search algorithm of Section 8.1, one end of the edge i will have been found already. It will create a cycle iff the other end has also been found already.

Running Time: The main loop iterates m times, once per edge. The priority queue operations $Insert$ and $ExtractMax$ each takes $\mathcal{O}(\log m)$ time. Therefore, the running time of the algorithm is $\mathcal{O}(m \log m)$.

A More General Algorithm: When designing an algorithm it is best to leave as many implementation details unspecified as possible. One reason is that this gives more freedom to anyone who may want to implement or to modify your algorithm. Another reason is that it provides better intuition as to why the algorithm works. The following is a greedy criteria that is quite general.

Cheapest Connected Edge of Some Component: Partition the nodes of the graph G into connected *components* of nodes that are reachable from each other only through the edges in $Commit$. Nodes with no adjacent edges will be in a component by themselves. Each iteration, the algorithm is free to chooses however it likes one of these components. Denote this component with C. (If fact, if it prefers C can be the union of a number of different components.) Then the algorithm greedily commits to the cheapest edge of those that expand this component, i.e., cheapest from amongst those edges that are connected to the component and yet do not create a cycle with it. When ever it likes, the algorithm also has the freedom to throw away uncommitted edges that create a cycle with the edges in *Commit*.

Generalizing: This greedy criteria is general enough to include both Kruskal's and Prim's algorithms. Therefore, if we prove that this greedy algorithm works no matter how it is implemented, then we automatically prove that both Kruskal's and Prim's algorithms work.

Cheapest Edge (Kruskal's Algorithm): This general algorithm may choose the component that is connected to the cheapest uncommitted edge that does not create a cycle. Then when it chooses the cheapest edge out of this component, it gets the over all cheapest edge.

Cheapest Connected Edge (Prim's Algorithm): This general algorithm may always choose the component that contains the source node s. This amounts to Prim's Algorithm.

The Loop Invariant: The loop invariant is that we have not gone wrong. There is at least one optimal solution consistent with the choices made so far, i.e. containing the objects committed to so far and not containing the objects rejected so far.

Maintaining the Loop Invariant (i.e., $\langle LI' \rangle$ & not $\langle exit \rangle$ & $code_{loop} \rightarrow \langle LI'' \rangle$): If we are unlucky and the optimal MST $optS_{LI}$ that is assumed to exist by the loop invariant does not contain the edge i being committed to this iteration, then we must massage this optimal solution $optS_{LI}$ into another one $optS_{ours}$ that contains all of the edges in $Commit \cup \{i\}$. This proves that the loop invariant has been maintained.

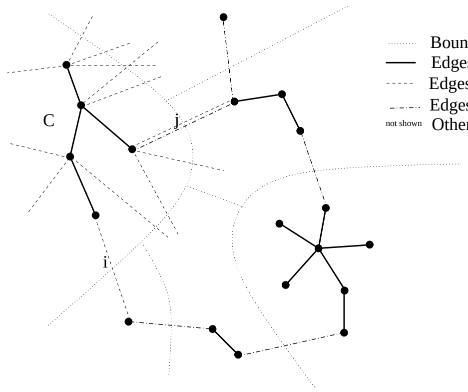


Figure 10.3: Let C be one of the components of the graph induced by the edges in *Commit*. Edge i is chosen to be the cheapest out of C. An optimal solution $optS_{LI}$ is assumed to exist that contains the edges in *Commit*. Let j be any edge in $optS_{LI}$ that is out of C. Form $optS_{ours}$ by removing j and adding i.

Massaging $optS_{LI}$ into $optS_{ours}$: Let u and v denote the two nodes of the edge i being committed to this iteration. As usual, the first massaging step is to add the edge i to the optimal solution $optS_{LI}$. The problem is that because $optS_{LI}$ spans the graph G, there is some path P within it from node u to node v. This path along with the edge $i = \{u, v\}$ creates a cycle. Cycles, however, are not allowed in our solution. Hence, we will have to break this cycle by removing one of its other edges.

Let C denote the component of *Commit* that the general greedy algorithm chooses the edge i from. Because i expands the component without creating a cycle with it, one and only one of the edge i's nodes u and v are within the component. Hence, this path P starts within C and ends outside of C. Hence, there must be some edge j in the path that leaves C. We will delete this edge from our optimal MST, namely $optS_{ours} = optS_{LI} \cup \{i\} - \{j\}$.

$optS_{ours}$ is an Optimal Solution:

$optS_{ours}$ has no cycles: Because $optS_{LI}$ has no cycles, we create only one cycle by adding edge i and we destroyed this cycle by deleting edge j.

$optS_{ours}$ Spans G: Because $optS_{LI}$ spans G, $optS_{ours}$ spans it as well. Any path between two nodes that goes through edge j in $optS_{LI}$ will now follow the remaining edges of path P together with edge i in $optS_{ours}$.

$optS_{ours}$ has minimum weight: Because $optS_{LI}$ has minimum weight, it is sufficient to prove that edge i is at least as cheap as edge j. Note that by construction edge j leaves component C and does not create a cycle with it. Because edge i was chosen because it was the cheapest (or at least one of the cheapest) such edges, it is true that edge i is at least as cheap as edge j.

Exercise 10.2.2 *In Figure 10.2, suppose we decide to change the weight 61 to any real number from $-\infty$ to $+\infty$. What is the interval of values that it could be changed to, for which the MST remains the same? Explain your answer. Similarly for the weight 95.*

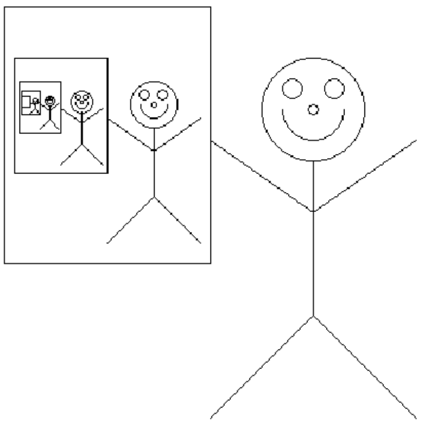
Part III

Recursive Algorithms

Chapter 11

An Introduction to Recursion

The previous chapters covered iterative algorithms. These start at the beginning and take one step at a time towards the final destination. Another technique used in many algorithms is to slice the given task into a number of disjoint pieces, to solve each of these separately, and then to combine these answers into an answer for the original task. This is the *divide and conquer* method. When the subtasks are different, this leads to different subroutines. When they are instances of the original problem, it leads to recursive algorithms.



11.1 Recursion - Abstractions, Techniques, and Theory

People often find recursive algorithms very difficult. To understand them, it is important to have a good solid understanding of the theory and techniques presented in this section.

11.1.1 Different Abstractions

There are a number of different abstractions within which one can view a recursive algorithm. Though the resulting algorithm is the same, having the different paradigms at your disposal can be helpful.

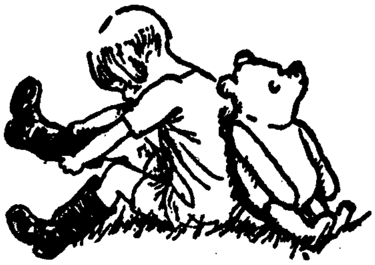
Code: Code is useful for implementing an algorithm on a computer. It is precise and succinct. However, as said, code is prone to bugs, language dependent, and often lacks higher level of intuition.

Stack of Stack Frames: Recursive algorithms are executed using a stack of stack frames. Though this should be understood, tracing out such an execution is painful.

Tree of Stack Frames: This is a useful way of viewing the entire computation at once. It is particularly useful when computing the running time of the algorithm. However, the structure of the computation tree might be very complex and difficult to understand all at once.

Friends & Strong Induction: The easiest method is to focus separately on one step at a time.

Suppose that someone gives you an instance of the computational problem. You solve it as follows. If it is sufficiently small, solve it yourself. Otherwise, you have a number of friends to help you. You construct for each friend an instance of the same computational problem that is *smaller* then your own. We refer to these as *subinstances*. Your friends magically provide you with the solutions to these. You then combine these *subsolutions* into a solution for your original instance.



I refer to this as the *friends* level of abstraction. If you prefer, you can call it the *strong induction* level of abstraction and use the word "recurse" instead of "friend". Either way, the key is that you concern yourself only about your stack frame. Do not worry about how your friends solve the subinstances that you assigned them. Similarly, do not worry about whom ever gave you your instance and what he does with your answer. Leave these things up to them.

Though students resist it, I strongly recommend using this method when designing, understanding, and describing a recursive algorithm.

Faith in the Method: As said for the loop invariant method, you do not want to be rethinking the issue of whether or not you should steal every time you walk into a store. It is better to have some general principles with which to work. Every time you consider a hard algorithm, you do not want to be rethinking the issue of whether or not you believe in recursion. Understanding the algorithm itself will be hard enough. Hence, while reading this chapter you should once and for all come to understand and believe to the depth of your soul how the above mentioned steps are sufficient to describing a recursive algorithm. Doing this can be difficult. It requires a whole new way of looking at algorithms. However, at least for the duration of this course, adopt this as something that you believe in.

11.1.2 Circular Argument? Looking Forwards vs Backwards

Circular Argument?: Recursion involves designing an algorithm by using it as if it already exists. At first this look paradoxical. Consider the following related problem. You must get into a house, but the door is locked and the key is inside. The magical solution is as follows. If I could get in, I could get the key. Then I could open the door, so that I could get in. This is a circular argument. However, it is not a legal recursive program because the sub-instance is not smaller.

One Problem and a Row of Instances: Consider a row of houses. The recursive problem consists of getting into any specified house. Each house in the row is a separate instance of this problem. Each house is bigger than the next. Task is to get into the biggest one. You are locked out of all the houses. However, the key to a house is locked in the house of the next smaller size.

The Algorithm: The task is no longer circular. The algorithm is as follows. The smallest house is small enough that one can use brute force to get in. For example, one could simply lift off the roof. Once in this house, we can get the key to the next house, which is then easily opened. Within this house, we can get the key to the house after that and so on. Eventually, we are in the largest house as required.

Focus On One Step: Though this algorithm is quite simple to understand, more complex algorithms are harder to understand all at once. Instead we want to focus on one step at a time. Here one step consists of the following. We are required to open house i. We ask a friend to open house $i-1$, out of which we take the key with which we open house i.

Working Forwards vs Backwards: An iterative algorithm works forward. It knows about house $i-1$. It uses a loop invariant to assume that this house has been opened. It searches this house and learns that the key within it is that for house i. Because of this, it decides that house i would be a good one to go to next.

A recursion algorithm works backwards. It knows about house i. It wants to get it open. It determines that the key for house i is contained in house $i-1$. Hence, opening house $i-1$ is a subtask that needs to be accomplished.

There are two advantages of recursive algorithms over iterative ones. The first is that sometimes it is easier to work backwards than forwards. The second is that a recursive algorithm is allowed to have more than one subtask to be solved. This forms a tree of houses to open instead of a row of houses.

Do Not Trace: When designing a recursive algorithm it is tempting to trace out the entire computation. "I must open house n, so I must open house $n-1$, ... The smallest house I rip the roof off. I get the key for house 1 and open it. I get the key for house 2 and open it. ... I get the key for house n and open it." Such an explanation is bound to be incomprehensible.

Solving Only Your Instance: An important quality of any leader is knowing how to delegate. Your job is to open house i. Delegate to a friend the task of opening house $i-1$. Trust him and leave the responsibility to him.

11.1.3 The Friends Recursion Level of Abstraction

The following are the steps to follow when developing a recursive algorithm within the friends level of abstraction.

Specifications: Carefully write the specifications for the problem.

Preconditions: The preconditions state any assumptions that must be true about the input instance for the algorithm to operate correctly.

Postconditions: The postconditions are statements about the output that must be true when the algorithm returns.

This step is even more important for recursive algorithms than for other algorithms, because there must be a tight agreement between what is expected from you in terms of pre and post conditions and what is expected from your friends.

Size: Devise a measure of the "size" of each instance. This measure can be anything you like and corresponds to the measure of progress within the loop invariant level of abstraction.

General Input: Consider a large and general instance of the problem.

Magic: Assume that by "magic" a friend is able to provide the solution to any instance of your problem as long as the instance is strictly smaller than the current instance (according to your measure of size). More specifically, if the instance that you give the friend meets the stated preconditions, then his solution will meet the stated post conditions. Do **not**, however, expect your friend to accomplish more than this. (In reality, the friend is simply a mirror image of yourself.)

Subinstances: From the original instance, construct one or more *subinstances*, which are smaller instances of the same problem. Be sure that the preconditions are met for these smaller instances. Do not refer to these as "subproblems". The problem does not change, just the input instance to the problem.

Subsolutions: Ask your friend to (recursively) provide solutions for each of these subinstances. We refer to these as *subsolutions* even though it is not the solution, but the instance that is smaller.

Solution: Combine these subsolutions into a solution for the original instance.

Generalizing the Problem: Sometimes a subinstance you would like your friend to solve is not a legal instance according to the preconditions. In such a case, start over redefining the preconditions in order to allow such instances. Note, however, that now you too must be able to handle these extra instances. Similarly, the solution provided by your friend may not provide enough information about the subinstance for you to be able to solve the original problem. In such a case, start over redefining the post condition by increasing the amount of information that your friend provides. Note again that now you too must also provide this extra information. See Section 12.3.

Minimizing the Number of Cases: You must ensure that the algorithm that you develop works for *every* valid input instance. To achieve this, the algorithm often will require many separate pieces of code to handle inputs of different types. Ideally, however, the algorithm developed has as few such cases as possible. One way to help you minimize the number of cases needed is as follows. Initially, consider an instance that is as large and as general as possible. If there are a number of different types of instances, choose one whose type is as general as possible. Design an algorithm that works for this instance. Afterwards, if there is another type of instance that you have not yet considered, consider a general instance of this type. Before designing a separate algorithm for this new instance, try executing your existing algorithm on it. You may be surprised to find that it works. If, on the other hand, it fails to work for this instance, then repeat the above steps to develop a separate algorithm for this case. This process may need to be repeated a number of times.

For example, suppose that the input consists of a binary tree. You may well find that the algorithm designed for a tree with a full left child and a full right child also works for a tree with a missing child and even for a child consisting of only a single node. The only remaining case may be the empty tree.

Base Cases: When all the remaining unsolved instances are sufficiently small, solve them in a brute force way.

Running Time: Use a recurrence relation or the tree of stack frames to estimate the running time.

11.1.4 Proving Correctness with Strong Induction

Whether you give your subinstances to friends or you recurse on them, this level of abstraction considers only the algorithm for the "top" stack frame. We must now prove that this suffices to produce an algorithm that successfully solves the problem on every input instance. When proving this, it is tempting to talk about stack frames. This stack frame calls this one, which calls that one, until you hit the base case. Then the solutions bubble back up to the surface. These proofs tend to make little sense and get very low marks. Instead, we use strong induction to prove formally that the friends level of abstraction works.

Strong Induction: Strong induction is similar to induction except instead of assuming only $S(n-1)$ to prove $S(n)$, you must assume all of $S(0), S(1), S(2), \ldots, S(n-1)$.

A Statement for each n: For each value of $n \geq 0$, let $S(n)$ represent a boolean statement. For some values of n, this statement may be true and for others it may be false.

Goal: Our goal is to prove that it is true for every value of n, namely that $\forall n \geq 0,\ S(n)$.

Proof Outline: Proof by strong induction on n.

Induction Hypothesis: "For each $n \geq 0$, let $S(n)$ be the statement that ...". (It is important to state this clearly.)

Base Case: Prove that the statement $S(0)$ is true.

Induction Step: For each $n \geq 0$, prove $S(0), S(1), S(2), \ldots, S(n-1) \Rightarrow S(n)$.

Conclusion: "By way of induction, we can conclude that $\forall n \geq 0,\ S(n)$."

Exercise 11.1.1 *Give the "process" of strong induction as we did for regular induction.*

Exercise 11.1.2 *(See solution in Section 20) As a formal statement, the base case can be eliminated because it is included in the formal induction step. How is this? (In practice, the base cases are still proved separately.)*

Proving The Recursive Algorithm Works:

Induction Hypothesis: For each $n \geq 0$, let $S(n)$ be the statement, "The recursive algorithm works for *every* instance of size n."

Goal: Our goal is to prove that $\forall n \geq 0,\ S(n)$, namely that "The recursive algorithm works for *every* instance."

Proof Outline: The proof is by strong induction on n.

Base Case: Proving $S(0)$ involves showing that the algorithm works for the base cases of size $n = 0$.

Induction Step: The statement $S(0), S(1), S(2), \ldots, S(n-1) \Rightarrow S(n)$ is proved as follows. First assume that the algorithm works for every instance of size strictly smaller than n and then proving that it works for every instance of size n. This mirrors exactly what we do in the friends level of abstraction. To prove that the algorithm works for every instance of size n, consider an arbitrary instance of size n. The algorithm constructs subinstances that are strictly smaller. By our induction hypothesis we know that our algorithm works for these. Hence, the recursive calls return the correct solutions. Within the friends level of abstraction, we proved that the algorithm constructs the correct solutions to our instance from the correct solutions to the subinstances. Hence, the algorithm works for this arbitrary instance of size n. $S(n)$ follows.

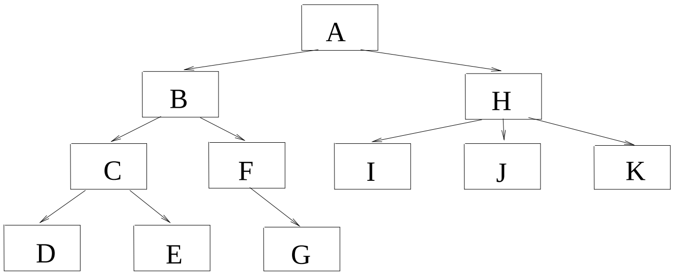
Conclusion: By way of strong induction, we can conclude that $\forall n \geq 0,\ S(n)$, i.e., The recursive algorithm works for every instance.

11.1.5 The Stack Frame Levels of Abstraction

Tree Of Stack Frames Level Of Abstraction: Tracing out the entire computation of a recursive algorithm, one line of code at a time, can get incredibly complex. This is why the friend's level of abstraction, which considers one stack frame at a time, is the best way to understand, explain, and design a recursive algorithm. However, it is also useful to have some picture of the entire computation. For this, the tree of stack frames level of abstraction is best.

The key thing to understand is the difference between a particular routine and a particular execution of a routine on a particular input instance. A single routine can at one moment in time have many instances of it being "executed". Each such execution is referred to as a *stack frame*. You can think of each as the task given to a separate friend. Note that, even though each may be executing exactly the same routine, each may currently be on a different line of code and have different values for the local variables.

If each routine makes a number of subroutine calls (recursive or not), then the stack frames that get executed form a tree.



In this example, instance A is called first. It executes for a while and at some point recursively calls B. When B returns, A then executes for a while longer before calling H. When H returns, A executes for a while before completing. We skipped over the details of the execution of B. Let's go back to when instance A calls B. B then calls C, which calls D. D completes, then C calls E. After E, C completes. Then B calls F which calls G. G completes, F completes, B completes and A goes on to call H. It does get complicated.

Stack Of Stack Frames Level Of Abstraction: Both the friend level of abstraction and the tree of stack frames level are only abstract ways of understanding the computation. The algorithm is actually implemented on a computer by a stack of stack frames. What is actually stored in the computer memory at any given point in time is a only single path down the tree. The tree represents what occurs through out time.

In the above example, when instance G is active, the stack frames that are in memory waiting are A, B, F, and G; the stack frames for C, D and E have been removed from memory as these have completed. H, I, J, and K have not been started yet.

Although we speak of many separate stack frames executing on the computer, the computer is not a parallel machine. Only the most recently created instance is actively being executed. The other instances are on hold waiting for a sub-routine call that it made to return.

It is useful to understand how memory is managed for the simultaneous execution of many instances of the same routine. The routine itself is described only once by a block of code that appears in static memory. This code declares a set of variables. Each instance of this routine that is currently being executed, on the other hand, may be storing different values in these variables and hence needs to have its own separate copy of these variables. The memory requirements of each of these instances are stored in a separate "stack frame". These frames are stacked on top of each other within stack memory. The first stack frame in the stack is that for the main routine. The second is that created when the execution of the main routine made a subroutine call. The third is that created when the execution of this subroutine call made a subroutine call, and so on.

Recall that a stack is a data structure in which either the last element to be added is *popped* off or a new element is *pushed* onto the top. Similarly here. The last stack frame to enter the stack is the instance that is currently being executed and on its completion popped off the stack. Let us denote this stack frame by A. When the execution of A makes a subroutine call to a routine with some input values, a stack frame is created for this new instance. This frame denoted B is pushed onto the stack after that for A. In addition to a separate copy of the local variables for the routine, it contains a pointer to the next line code that A must execute when B returns. When B returns, its stack frame is popped and A continues to execute at the line of code that had been indicated within B.

Silly Example: The following is a silly example that demonstrates how difficult it is to trace out the full stack-frame tree, yet how easy it is to determine the output using the friends/strong-induction method.

algorithm $Fun(n)$

$\langle$***pre−cond***$\rangle$**:** n is an integer.

$\langle$***post−cond***$\rangle$**:** Outputs a silly string.

```
begin
    if( n > 0 ) then
        if( n = 1 ) then
            put "X"
        else if( n = 2 ) then
            put "Y"
        else
            put "A"
            Fun(n − 1)
            Put "B"
            Fun(n − 2)
            Put "C"
        end if
    end if
end algorithm
```

Exercise 11.1.3 *Attempt to trace out the tree of stack frames for this algorithm for* $n = 5$*.*

Exercise 11.1.4 *(See solution in Section 20) Now try the following simpler approach. What is the output with* $n = 1$*? What is the output with* $n = 2$*? Trust the answers to all previous questions; do not recalculate them. (Assume a trusted friend gave you the answer.) Now, what is the output with* $n = 3$*? Repeat this approach for* $n = 4$*, 5, and 6.*

11.2 Some Simple Examples of Recursive Algorithms

I will now give some simple examples of recursive algorithms. Even if you have already seen them before, study them again, keeping the abstractions, techniques, and theory learned in this chapter in mind. For each example, look for the key steps of the friend paradigm. What are the subinstances given to the friend? What is the size of an instance? Does it get smaller? How are the friend's solutions combined to give your solution? As well, what does the tree of stack frames look like? What is the time complexity of the algorithm?

11.2.1 Sorting and Selecting Algorithms

The classic divide and conquer algorithms are merge sort and quick sort. They both have the following basic structure.

General Recursive Sorting Algorithm:

- Take the given list of objects to be sorted (numbers, strings, student records, etc.)
- Split the list into two sublists.
- Recursively have a friend sort the two sublists.
- Combine the two sorted sublists into one entirely sorted list.

This process leads to four different algorithms, depending on whether you:

Sizes: Split the list into two sublists each of size $\frac{n}{2}$ or one of size $n-1$ and the one of size one.

Work: Put minimal effort into splitting the list but put lots of effort into recombining the sublists or put lots of effort into splitting the list but put minimal effort into recombining the sublists.

Exercise 11.2.1 *(See solution in Section 20) Consider the algorithm that puts minimal effort into splitting the list into one of size* $n-1$ *and the one of size one, but put lots of effort into recombining the sublists. Also consider the algorithm that puts lots of effort into splitting the list into one of size* $n-1$ *and the one of size one, but put minimal effort into recombining the sublists. What are these two algorithms?*

Merge Sort (Minimal work to split in half): This is the classic recursive algorithm.

Friend's Level of Abstraction: Recursively give one friend the first half of the input to sort and another friend the second half to sort. You then combine these two sorted sublists into one completely sorted list. This combining process is referred to as *merging*. A simple linear time algorithm for it can be found in Section 3.2.2.

Size: The size of an instance is the number of elements in the list. If this is at least two, then the sublists are smaller than the whole list. Hence, it is valid to recurse on them with the reassurance that your friends will do their parts correctly. On the other hand, if the list contains only one element, then by default it is already sorted and nothing needs to be done.

Generalizing the Problem: If the input is assumed to be received in an array indexed from 1 to n, then the second half of the list is not a valid instance. Hence, we redefine the preconditions of the sorting problem to require as input both an array A and a subrange $[i, j]$. The postcondition is that the specified sublist be sorted in place.

Running Time: Let $T(n)$ be the total time required to sort a list of n elements. This total time consists of the time for two subinstance of half the size to be sorted, plus $\Theta(n)$ time for merging the two sublists together. This gives the recurrence relation $T(n) = 2T(n/2) + \Theta(n)$. See Section 1.6 to learn how to solve recurrence relations like these. In this example, $\frac{\log a}{\log b} = \frac{\log 2}{\log 2} = 1$ and $f(n) = \Theta(n^1)$ so $c = 1$. Because $\frac{\log a}{\log b} = c$, the technique concludes that time is dominated by all levels and $T(n) = \Theta(f(n) \log n) = \Theta(n \log n)$.

Tree of Stack Frames: The following is a tree of stack frames for a concrete example.

```
               --------------------------------------------
               | In:  100 21 40 97 53 9 25 105 99 8 45 10 |
               | Out: 8 9 10 21 25 40 45 53 97 99 100 105 |
               |__________________________________________|
                             /            \
         ------------------/-------   -----\--------------------
         | In:  100 21 40 97 53 9 |   | In:  25 105 99 8 45 10 |
         | Out: 9 21 40 53 97 100 |   | Out: 8 10 25 45 99 105 |
         |________________________|   |________________________|
               /         \                     /         \
 --------------/---  ----\-----------  ---------/--------  --\-------------
 | In:  100 21 40 |  | In:  97 53 9 |  | In:  25 105 99 |  | In:  8 45 10 |
 | Out: 21 40 100 |  | Out: 9 53 97 |  | Out: 25 99 105 |  | Out: 8 10 45 |
 |________________|  |______________|  |________________|  |______________|
          /     \          /     \          /      \          /      \
     -----/---- --\--- ---/----- -\--- ----/----- --\--- ---/---- --\---
In:  | 100 21 | | 40 | | 97 53 | | 9 | | 25 105 | | 99 | | 8 45 | | 10 |
Out: | 21 100 | | 40 | | 53 97 | | 9 | | 25 105 | | 99 | | 8 45 | | 10 |
     |________| |____| |_______| |___| |________| |____| |______| |____|
        /   \          /   \             /   \              /   \
     ---/--- -\----  --/--- -\----    --/--- --\----     --/-- --\---
In:  | 100 | | 21 |  | 97 | | 53 |    | 25 | | 105 |     | 8 | | 45 |
Out: | 100 | | 21 |  | 97 | | 53 |    | 25 | | 105 |     | 8 | | 45 |
     |_____| |____|  |____| |____|    |____| |_____|     |___| |____|
```

Quick Sort (Minimal work to recombine the halves): The following is one of the fastest sorting algorithms. Hence, the name.

Friend's Level of Abstraction: The first step in the algorithm is to choose one of the elements to be the "pivot element". How this is to be done is discussed below. The next step is to *partition* the list into two sublists using the Partition routine defined below. This routine rearranges the elements so that all the elements that are less than or equal to the pivot element are to the left of the pivot element and all the elements that are greater than it are to the right of it. (There are no requirements on the order of the elements in the sublists.) Next, recursively have a friend sort those elements before the pivot and those after it. Finally, (without effort) put the sublists together, forming one completely sorted list.

Tree of Stack Frames: The following is a tree of stack frames for a concrete example.

```
               ____________________________________________
               | In:  100 21 40 97 53 9 25 105 99 8 45 10 |
               | Out: 8 9 10 21 25 40 45 53 97 99 100 105 |
               |__________________________________________|
                            /   |   \
         __________________/    |    \_______________________________
         | In:  21 9 8 10 |    25     | In:  100 40 97 53 105 99 45 |
         | Out: 8 9 10 21 |           | Out: 40 45 53 97 99 100 105 |
         |________________|           |_____________________________|
                / | \                        /          |    \
   ____________/  |  \___________   _________/_______   |  ___\_______________
   | In:  9 8 |  10   | In:  21 |   | In:  40 53 45 |  97  | In:  100 105 99 |
   | Out: 8 9 |       | Out: 21 |   | Out: 40 45 53 |      | Out: 99 100 105 |
   |__________|       |_________|   |_______________|      |_________________|
        /    | \                         /   |  \               /   |   \
     __/__   | _\__               ______/__  | __\_         ___/__  |  __\____
In:  | 8 |   9 |  |               | 40 45 | 53 |  |         | 99 | 100 | 105 |
Out: | 8 |     |  |               | 40 45 |    |  |         | 99 |     | 105 |
     |___|     |__|               |_______|    |__|         |____|     |_____|
                                    / | \
                                ___/  | _\____
                           In:  |  | 40 | 45 |
                           Out: |  |    | 45 |
                                |__|    |____|
```

Running Time: The computation time depends on the choice of the pivot element.

Median: If we are lucky and the pivot element is close to being the median value, then the list will be split into two sublists of size approximately $n/2$. We will see that partitioning the array according to the pivot element can be done in time $\Theta(n)$. In this case, the timing is $T(n) = 2T(n/2) + \Theta(n) = \Theta(nlogn)$.

Reasonable Split: The above timing is quite robust with respect to the choice of the pivot. For example, suppose that the pivot always partitions the list into one sublist of $\frac{1}{5}th$ the original size and one of $\frac{4}{5}th$ the size. The total time is then the time to partition plus the time to sort the sublists of these sizes. This gives $T(n) = T(\frac{1}{5}n) + T(\frac{4}{5}n) + \Theta(n)$. Because $\frac{1}{5} + \frac{4}{5} = 1$, this evaluates to $T(n) = \Theta(n \log n)$. (See Section 1.6.) More generally, suppose that the pivot always partitions the list so that the first half is between $\frac{1}{5}th$ and $\frac{4}{5}th$ the original size. The time will still be $\Theta(n \log n)$.

Worst Case: On the other hand, suppose that the pivot always splits the list into one of size $n-1$ and one of size 1. In this case, $T(n) = T(n-1) + T(1) + \Theta(n)$, which evaluates to $T(n) = \Theta(n^2)$. This is the worst case scenario.

We will return to Quick Sort after considering the following related problem.

Finding the k^{th} Smallest Element: Given an unsorted list and an integer k, this problem finds the k^{th} smallest element from the list. It is not clear at first that there is an algorithm for doing this that is any faster than sorting the entire list. However, it can be done in linear time using the subroutine Pivot.

Friend's Level of Abstraction: The algorithm is like that for binary search. Ignoring input k, it proceeds just like quick sort. A pivot element is chosen randomly, and the list is split into two sublists, the first containing all elements that are all less than or equal to the pivot element and the second those that are greater than it. Let ℓ be the number of elements in the first sublist. If $\ell \geq k$, then we know that the k^{th} smallest element from the entire list is also the k^{th} smallest element from the first sublist. Hence, we can give this first sublist and this k to a friend and ask him to find it. On the other hand, if $\ell < k$, then we know that the k^{th} smallest element from the entire list is the $(k-\ell)^{th}$ smallest element from the second sublist. Hence, giving the second sublist and $k-\ell$ to a friend, he can find it.

Tree of Stack Frames: The following is a tree of stack frames for a concrete example.

```
                 ----------------------------------------------
                 |     In: 100 21 40 97 53 9 25 105 99 8 45 10 |
                 | Sorted: 8 9 10 21 25 40 45 53 97 99 100 105 |
                 | k=7 Out:45                                  |
                 |_____________________________________________|
                           Pivot:25   \
                                       \_________________________________
                                       | In:      100 40 97 53 105 99 45 |
                                       | Sorted: 40 45 53 97 99 100 105 |
                                       | k=7-5=2 Out:45                 |
                                       |________________________________|
                                                 /   Pivot:97
                                     _________/_____________
                                     | In:     40 53 45 97 |
                                     | Sorted: 40 45 53 97 |
                                     | k=2 Out:45          |
                                     |_____________________|
                                          /  Pivot:45
                                   ______/__________
                                   | In:     40 45 |
                                   | Sorted: 40 45 |
                                   | k=2 Out:45    |
                                   |_______________|
                                         \
                             Pivot:40   _\________________
                                         | In:     45     |
                                         | Sorted: 45     |
                                         | k=2-1=1 Out:45 |
                                         |________________|
```

Running Time: Again, the computation time depends on the choice of the pivot element.

Median: If we are lucky and the pivot element is close to being the median value, then the list will be split into two sublists of size approximately $n/2$. Because the routine recurses on only one of the halves, the timing is $T(n) = T(n/2) + \Theta(n) = \Theta(n)$.

Reasonable Split: If the pivot always partitions the list so that the larger half is at most $\frac{4}{5}n$, then the total time is at most $T(n) = T(\frac{4}{5}n) + \Theta(n)$, which is still linear time, $T(n) = \Theta(n)$.

Worst Case: In the worst case, the pivot splits the list into one of size $n-1$ and one of size 1. In this case, $T(n) = T(n-1) + \Theta(n)$, which is $T(n) = \Theta(n^2)$.

Choosing the Pivot: In the last two algorithms, the timing depends on choosing a good pivot element quickly.

Fixed Value: If you knew that you were sorting elements that are numbers within the range [1..100], then it reasonable to partition these elements based on whether they are smaller or larger than 50. This is often referred to as *bucket sort*. See below. However, there are two problems with this technique. The first is that in general we do not know what range the input elements will lie. The second is that at every level of recursion another pivot value is needed with which to partition the elements. The solution is to use the input itself to choose the pivot value.

Use $A[1]$ as the Pivot: The first thing one might try is to let the pivot be the element that happens to be first in the input array. The problem with this is that if the input happens to be sorted (or almost sorted) already, then this first element will split the list into one of size zero and one of size $n-1$. This gives the worst case time of $\Theta(n^2)$. Given random data, the algorithm will execute quickly. On the other hand, if you forget that you sorted the data and you run it a second time, then second time will take a long time to complete.

Use $A[\frac{n}{2}]$ as the Pivot: Motivated by the last attempt, one might use the element that happens to be located in the middle of the input array. For all practical purposes, this would likely work great. It would work exceptionally well when the list is already sorted. However, the worst case time complexity assumes that the input instance is chosen by some nasty adversary. The adversary will provide an input for which the pivot is always bad. The worst case time complexity is still $\Theta(n^2)$.

A Randomly Chosen Element: In practice, what is often done is to choose the pivot element randomly from the input elements. See Section 19.2. The advantage of this is that the adversary who is choosing the worst case input instance, knows the algorithm, but does not know the random coin tosses. Hence, all input instances are equivalently good and equivalently bad.

We will prove that the expected computation time is $\Theta(nlogn)$. What this means is that if you ran the algorithm 1000000 times on the same input, then the average running time would be $\Theta(nlogn)$. We prove this as follows. Suppose that the randomly chosen pivot element happens to be the i^{th} smallest element. The list is split into one of size i and one of size $n-i$. This gives the recursive relation $T(n) = \text{Avg}_{i=1}^{n}\left[\, T(i) + T(n-i) + \Theta(n)\,\right]$. With a little work, this evaluates to $\Theta(nlogn)$.

Even though the expected time is good, it would be bad, if once in a while the running time is $\Theta(n^2)$. We will now prove that the probability that the running time is not $\Theta(n\log n)$ is $2^{-\Theta(n)}$, i.e. for reasonable n it is not going to happen within your lifetime. We prove this as follows. Let us say progress is made if pivot element is chosen such that the split is between $\frac{1}{5}$ and $\frac{4}{5}$. This occurs if the pivot happens to be one of the elements between the $\frac{1}{5}n^{th}$ and the $\frac{4}{5}n^{th}$ smallest. The probability of this is $\frac{3}{5}$, which means that we expect progress to occur $\frac{3}{5}ths$ of the time. Hence, if we recurse H times, the expected number of times progress made is $h = \frac{3}{5}H$. The probability is exponentially small, $2^{-\Theta(n)}$, that we are unlucky and the progress is less than say $h = \frac{1}{2}H$ times. When h progress has been made, the subinstance are no larger than $n/(\frac{4}{5})^h$. This size becomes 1 when $H = 2\log(n)/\log(4/5)$ and hence $h \geq \log(n)/\log(4/5)$. The work at each level of recursion is $\Theta(n)$. Hence, the total time is $\Theta(n\log n)$.

Randomly Choose 3 Elements: Another option is to randomly select three elements from the input list and use the middle one as the pivot. Doing this greatly increases the probability that the pivot is close to the middle and hence decreases the probability of the worst case occurring. However, doing so also takes time. All in all, the expected running time is worse.

A Deterministic Algorithm: The following is a deterministic method of choosing the pivot that leads to the worst case running time of $\Theta(n)$ for finding k^{th} smallest element. First group the n elements into $\frac{n}{5}$ groups of 5 elements each. Within each group of 5 elements, do $\Theta(1)$ work to find the median of the group. Let S_{median} be the set of $\frac{n}{5}$ elements that is the median from each group. Recursively ask a friend to find the median element from the set S_{median}. This element will be used as our pivot.

We claim that this pivot element has at least $\frac{3}{10}n$ elements that are less than or equal to it and another $\frac{3}{10}n$ elements that are greater or equal to it. The proof of the claim is as follows. Because the pivot is the median within S_{median}, there are $\frac{1}{10}n = \frac{1}{2}|S_{median}|$ elements within S_{median} that are less than or equal to the pivot. Consider any such element $x_i \in S_{median}$. Because x_i is the median within its group of 5 elements, there are 3 elements within this group (including x_i itself) that are less than or equal to x_i and hence in turn less than or equal to the pivot. Counting all these gives $3 \cdot \frac{1}{10}n$ elements. A similar argument counts this many that are greater or equal to the pivot.

The algorithm to find the k^{th} largest element proceeds as stated originally. A friend is either asked to find the k^{th} smallest element within all elements that are less than or equal to the pivot or the $(k-\ell)^{th}$ smallest element from all those that are greater than it. The claim insures that the size of the sublist given to the friend is at most $\frac{7}{10}n$.

Unlike the first algorithm for the finding k^{th} smallest element, this algorithm recurses twice. Hence, one would initially assume that the running time is $\Theta(n\log n)$. However, careful analysis shows that it is only $\Theta(n)$. Let $T(n)$ denote the running time. Finding the median of each of the $\frac{1}{5}n$ groups takes $\Theta(n)$ time. Recursively finding the median of S_{median} takes $T(\frac{1}{5}n)$ time. Recursing on the remaining at most $\frac{7}{10}n$ elements takes at most $T(\frac{7}{10}n)$ time. This gives a total of $T(n) = T(\frac{1}{5}n) + T(\frac{7}{10}n) + \Theta(n)$ time. Because $\frac{1}{5} + \frac{7}{10} < 1$, this evaluates to $T(n) = \Theta(n)$. (See Section 1.6).

A deterministic Quick Sort algorithm can use this deterministic $\Theta(n)$ time algorithm for the finding k^{th} smallest element, to find the median of the list to be the pivot. Because partitioning the elements according to the pivot already takes $\Theta(n)$ time, the timing is still $T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$.

Partitioning According To The Pivot Element: The input consists of a list of elements $A[I], \ldots, A[J]$ and a pivot element. The output consists of the rearranged elements and an index i, such that the elements $A[I], \ldots, A[i-1]$ are all less than or equal to the pivot element, $A[i]$ is the pivot element, and the elements $A[i+1], \ldots, A[J]$ are all greater than it.

The loop invariant is that there are indexes $I \le i \le j \le J$ for which

1. The values in $A[I], \ldots, A[i-1]$ are less than or equal to the pivot element.
2. The values in $A[j+1], \ldots, A[J]$ are greater than the pivot element.
3. The pivot element has been removed and is on the side, leaving an empty entry either at $A[i]$ or at $A[j]$.
4. The other elements in $A[i], \ldots, A[j]$ have not been considered.

The loop invariant is established by setting $i = I$ and $j = J$, making $A[i]$ empty by putting the element in $A[i]$ where the pivot element is and putting the pivot element aside.

If the loop invariant is true and $i < j$, then there are four possible cases:

Case A) $A[i]$ is empty and $A[j] \le$ pivot: $A[j]$ belongs on the left, so move it to the empty $A[i]$. $A[j]$ is now empty. Increase the left side by increasing i by one.

Case B) $A[i]$ is empty and $A[j] >$ pivot: $A[j]$ belongs on the right and is already there. Increase the right side by decreasing j by one.

Case C) $A[j]$ is empty and $A[i] \le$ pivot: $A[i]$ belongs on the left and is already there. Increase the left side by increasing i by one.

Case D) $A[j]$ is empty and $A[i] >$ pivot: $A[i]$ belongs on the right, so move it to the empty $A[j]$. $A[i]$ is now empty. Increase the right side by decreasing j by one.

In each case, the loop invariant is maintained. Progress is made because $j - i$ decreases.

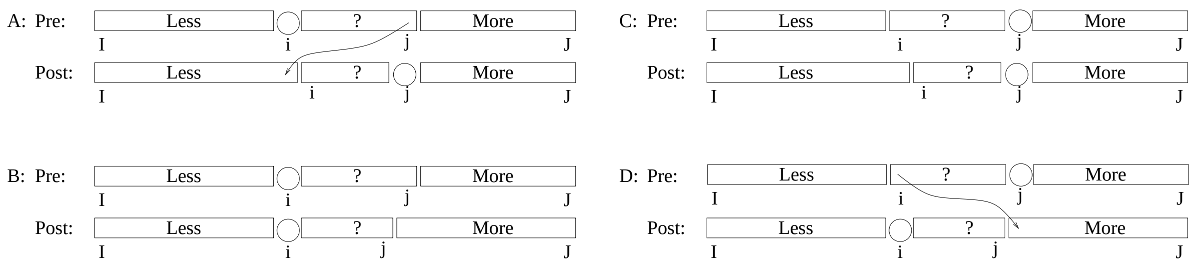


Figure 11.1: The four cases of how to iterate are shown.

When $i = j$, the list is split as needed, leaving $A[i]$ empty. Put the pivot there. The postcondition follows.

Sorting By Hand: As a professor, I often have to sort a large stack of student's papers by the last name. Surprisingly enough, the algorithm that I use is a version of quick sort called *bucket sort*.

Fixed Pivot Values: When fixed pivot values are used with which to partition the elements, the algorithm is called bucket sort instead of quick sort. This works for this application because the list to be sorted consist of names whose first letters are fairly predictably distributed through the alphabet.

Partitioning Into 5 Buckets: Computers are good at using a single comparison to determine whether an element is greater than the pivot value or not. Humans, on the other hand, tend to be good at quickly determining which of 5 buckets an element belongs in. I first partition the papers based on which of the following ranges the first letter of the name is within: [A-E], [F-K], [L-O], [P-T], or [U-Z]. Then I partition the [A-E] bucket into the sub-buckets [A], [B], [C], [D], and [E]. Then I partition the [A] bucket based on the second letter of the name.

A Stack of Buckets: One difficulty with this algorithm is keeping track of all the buckets. For example, after the second partition, we will have nine buckets: [A], [B], [C], [D], [E], [F-K], [L-O], [P-T], and [U-Z]. After the third, we will have 13. On a computer, the recursion of the algorithm is implemented with a stack of stack frames. Correspondingly, when I sort the student's papers, I have a stack of buckets.

Loop Invariant: I use the following loop invariant to keep track of what I am doing. I always have a pile (initially empty) of papers that are sorted. These papers belong before all other papers. I also have a stack of piles. Though the papers within each pile are out of order, each paper in a pile belongs before each paper in a later pile. For example, at some point in the algorithm, the papers starting with [A-C] will be sorted and the piles in my stack will consist of [D], [E], [F-K], [L-O], [P-T], and [U-Z].

Maintain the Loop Invariant: I make progress while maintaining this loop invariant as follows. I take the top pile off the stack, here the [D]. If it only contains a half dozen or so papers, I sort them using insertion sort. These are then added to the top of the sorted pile, [A-C], giving [A-D]. On the other hand, if the pile [D] taken off the stack is larger then this, I partition it into 5 piles, [DA-DE], [DF-DK], [DL-DO], [DP-DT], and [DU-DZ], which I push back onto the stack. Either way, my loop invariant is maintained.

Exiting: When the last bucket has been removed from the stack, the papers are sorted.

11.2.2 Operations on Integers

Raising an integer to a power b^N, multiplying $x \times y$, and matrix multiplication each have surprising divide and conquer algorithms.

$\boldsymbol{b^N}$**:** Suppose that you are given two integers b and N and want to compute b^N.

The Iterative Algorithm: The obvious iterative algorithm simply multiplies b together N times. The obvious recursive algorithm recurses with $Power(b, N) = b \times Power(b, N-1)$. This requires the same N multiplications.

The Straightforward Divide and Conquer Algorithm: The obvious divide and conquer technique cuts the problem into two halves using the property that $b^{\lceil \frac{N}{2} \rceil} \times b^{\lfloor \frac{N}{2} \rfloor} = b^{\lceil \frac{N}{2} \rceil + \lfloor \frac{N}{2} \rfloor} = b^N$. This leads to the recursive algorithm $Power(b, N) = Power(b, \lceil \frac{N}{2} \rceil) \times Power(b, \lfloor \frac{N}{2} \rfloor)$. Its recurrence relation gives $T(N) = 2T(\frac{N}{2}) + 1$ multiplications. The technique in Section 1.6 notes that $\frac{\log a}{\log b} = \frac{\log 2}{\log 2} = 1$ and $f(N) = \Theta(N^0)$ so $c = 0$. Because $\frac{\log a}{\log b} > c$, the technique concludes that time is dominated by the base cases and $T(N) = \Theta(N^{\frac{\log a}{\log b}}) = \Theta(N)$. This is no faster than the standard iterative algorithm.

Reducing the Number of Recursions: This algorithm can be improved by noting that the two recursive calls are almost the same and hence need only to be called once. The new recurrence relation gives $T(N) = 1T(\frac{N}{2}) + 1$ multiplications. Here $\frac{\log a}{\log b} = \frac{\log 1}{\log 2} = 0$ and $f(N) = \Theta(N^0)$ so $c = 0$. Because $\frac{\log a}{\log b} = c$, we conclude that time is dominated by all levels and $T(N) = \Theta(f(N) \log N) = \Theta(\log N)$ multiplications.

algorithm $Power(b, N)$
$\langle$***pre−cond***$\rangle$**:** $N \geq 0$ (N and b not both 0)
$\langle$***post−cond***$\rangle$**:** Outputs b^n.

```
begin
    if( N = 0 ) then
        result 1
    else
        half = ⌊N/2⌋
        p = Power(b, half)
        if( 2 · half = N ) then
            result( p · p )      % if N is even, b^N = b^{N/2} · b^{N/2}
        else
            result( p · p · b )  % if N is odd, b^N = b · b^{N/2} · b^{N/2}
        end if
    end if
end algorithm
```

Tree of Stack Frames:

```
      <-| return value
        | 32 = 4x4x2
  -----
 | b=2 |   <-| return value
 |_N=5_|     | 4 = 2x2
       \-----
       | b=2 |  <-| return value
       |_N=2_|    | 2 = 1x1x2
             \-----
             | b=2 |   <-| return value
             |_N=1_|     | 1
                   \-----
                   | b=2 |
                   |_N=0_|
```

Running Time: One is tempted to say that the first two $\Theta(N)$ algorithms are linear time and that the last $\Theta(\log N)$ one is logarithmic time. However, in fact the first two are exponential $\Theta(2^n)$ time and the last is linear $\Theta(n)$. Recall that time complexity is measured as a function of the "size" of the input, which is typically the number of bits $n = \log N$ to represent the number.

$x \times y$: The time complexity of the last algorithm was measured in terms of the number of multiplications. This begs the question of how quickly one can multiply.

The input for the next problem consists of two strings of n digits each. These are viewed as two integers x and y either in binary or in decimal notation. The problem is to multiply them.

The Iterative Algorithm: The standard elementary school algorithm considers each pair of digits, one from x and the other from y, and multiplies them together. These n^2 products are shifted appropriately and summed. The total time is $\Theta(n^2)$. It is hard to believe that one could do faster.

			8	2	7
			5	9	6
				4	2
			1	2	
		4	8		
			6	3	
		1	8		
	7	2			
		3	5		
	1	0			
4	0				
4	9	2	8	9	2

The Straightforward Divide and Conquer Algorithm: Let us see how well the divide and conquer technique can work. Split each sequence of digits in half and consider each half as an integer. This gives $x = x_1 \cdot 10^{\frac{n}{2}} + x_0$ and $y = y_1 \cdot 10^{\frac{n}{2}} + y_0$. Multiplying these symbolically gives

$$\begin{aligned} x \times y &= \left(x_1 \cdot 10^{\frac{n}{2}} + x_0\right) \times \left(y_1 \cdot 10^{\frac{n}{2}} + y_0\right) \\ &= (x_1 y_1) \cdot 10^n + (x_1 y_0 + x_0 y_1) \cdot 10^{\frac{n}{2}} + (x_0 y_0) \end{aligned}$$

The obvious divide and conquer algorithm would recursively compute the four subproblems x_1y_1, x_1y_0, x_0y_1, and x_0y_0, each of $\frac{n}{2}$ digits. This would take $4T(\frac{n}{2})$ time. Then these four products are shifted appropriately and summed. Note that additions can be done in $\Theta(n)$ time. See Section 7. Hence, the total time is $T(n) = 4T(\frac{n}{2}) + \Theta(n)$. Here $\frac{\log a}{\log b} = \frac{\log 4}{\log 2} = 2$ and $f(n) = \Theta(n^1)$ so $c = 1$. Because $\frac{\log a}{\log b} > c$, the technique concludes that time is dominated by the base cases and $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^2)$. This is no improvement in time.

Reducing the Number of Recursions: Suppose that we could find a trick so that we only need to recurse three times instead of four. One's intuition might be that this would only provide a linear time savings, but in fact the savings is much more. $T(n) = 3T(\frac{n}{2}) + \Theta(n)$. Now $\frac{\log a}{\log b} = \frac{\log 3}{\log 2} = 1.58..$, which is still bigger than $c =$. Hence, time is still dominated by the base cases, but now this is $T(n) = \Theta(n^{\frac{\log a}{\log b}}) = \Theta(n^{1.58..})$. This is a significant improvement from $\Theta(n^2)$.

The trick for requiring only three recursive multiplications is as follows. The first step is to multiply x_1y_1 and x_0y_0 recursively as required. This leaves us only one more recursive multiplication.

If you review the symbolic expansion of $x \times y$, you will see that we do not actually need to know the value of x_1y_0 and x_0y_1. We only need to know their sum. Symbolically, we can observe the following.

$$\begin{aligned} & x_1 y_0 + x_0 y_1 \\ &= [x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0] - x_1 y_1 - x_0 y_0 \\ &= [(x_1 + x_0)(y_1 + y_0)] - x_1 y_1 - x_0 y_0 \end{aligned}$$

Hence, the sum $x_1y_0 + x_0y_1$ that we need can be computed by adding x_1 to x_0 and y_1 to y_0; multiplying these sums; and subtracting off the values x_1y_1 and x_0y_0 that we know from before. This requires only one additional recursive multiplication. Again we use the fact that additions are fast, requiring only $\Theta(n)$ time.

algorithm $Multiply(x, y)$

$\langle$***pre−cond***$\rangle$**:** x and y are two integers represented as an array of n digits

$\langle$***post−cond***$\rangle$**:** The output consists of their product represented as an array of $n + 1$ digits

```
begin
    if(n=1) then
        result( x × y        ) % product of single digits
    else
        ⟨x1, x0⟩ = high and low order n/2 digits of x
        ⟨y1, y0⟩ = high and low order n/2 digits of y
        A = Multiply(x1, y1)
        C = Multiply(x0, y0)
        B = Multiply(x1 + x0, y1 + y0) − A − C
        result( A · 10^n + B · 10^(n/2) + C )
    end if
end algorithm
```

It is surprising that this trick reduces the time from $\Theta(n^2)$ to $\Theta(n^{1.58})$.

Dividing into More Parts: The next question is whether the same trick can be extended to improve the time even further. Instead of splitting each of x and y into two pieces, lets split them each into

d pieces. The straightforward method recursively multiplies each of the d^2 pairs of pieces together, one from x and one from y. The total time is $T(n) = d^2T(\frac{n}{d}) + \Theta(n)$. Here $a = d^2$, $b = d$, $c = 1$, and $\frac{\log d^2}{\log d} = 2 > c$. This gives $T(n) = \Theta(n^2)$. Again, we are back where we began.

Reducing the Number of Recursions: The trick now is to do the same with fewer recursive multiplications. It turns out it can be done with only $2d-1$ of them. This gives time of only $T(n) = (2d-1)T(\frac{n}{d}) + \Theta(n)$. Here $a = 2d-1$, $b = d$, $c = 1$, and $\frac{\log(2d-1)}{\log(d)} \approx \frac{\log(d)+1}{\log(d)} = 1 + \frac{1}{\log(d)} \approx c$. By increasing d, the time for the top stack frame and for the base cases becomes closer and closer to being equal. Recall that when this happens, we must add an extra $\Theta(\log n)$ factor to account for the $\Theta(\log n)$ levels of recursion. This gives $T(n) = \Theta(n \log n)$, which is a surprising running time for multiplication.

Fast Fourier Transformations: We will not describe the trick for reducing the number of recursive multiplications from d^2 to only $2d-1$. Let it suffice that it involves thinking of the problem as the evaluation and interpolation of polynomials. When d becomes large, other complications arise. These are solved by using the $2d$-roots of unity over a finite field. Performing operations over this finite field require $\Theta(\log\log n)$ time. This increases the total time from $\Theta(n \log n)$ to $\Theta(n \log n \log\log n)$. This algorithm is used often for multiplication and many other applications such as signal processing. It is referred to as *Fast Fourier Transformations*.

Exercise 11.2.2 *Design the algorithm and compute the running time when $d = 3$.*

Strassen's Matrix Multiplication: The next problem is to multiply two $n \times n$ matrices.

The Iterative Algorithm: The obvious iterative algorithm computes the $\langle i, j\rangle$ entry of the product matrix by multiplying the i^{th} row of the first matrix with the j^{th} column of the second. This requires $\Theta(n)$ scalar multiplications. Because there are n^2 such entries, the total time is $\Theta(n^3)$.

The Straightforward Divide and Conquer Algorithm: When designing a divide and conquer algorithm, the first step is to divide these two matrices into four submatrices each. Multiplying these symbolically gives the following.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} ae+bf & ag+bh \\ ce+df & cg+dh \end{pmatrix}$$

Computing the four $\frac{n}{2} \times \frac{n}{2}$ submatrices in this product in this way requires recursively multiplying eight pairs of $\frac{n}{2} \times \frac{n}{2}$ matrices. The total computation time is given by the recurrence relation $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^{\frac{\log 8}{\log 2}}) = \Theta(n^3)$. As seen before, this is no faster than the standard iterative algorithm.

Reducing the Number of Recursions: Strassen found a way of computing the four $\frac{n}{2} \times \frac{n}{2}$ submatrices in this product using only seven such recursive calls. This gives $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\frac{\log 7}{\log 2}}) = \Theta(n^{2.8073})$. We will not include the details of the algorithm.

11.2.3 Akerman's Function

If you are wondering just how slowly a program can run, consider the code below. Assume the input parameters n and k are natural numbers. Let $T_k(n)$ denote the value returned by $A(k, n)$.

```
algorithm A(k, n)
    if( k = 0) then
        return( n+1+1 )
    else
        if( n = 0) then
            if( k = 1) then
                return( 0 )
```

```
                else
                        return( 1 )
        else
                return( A(k − 1, A(k, n − 1)))
        end if
    end if
end algorithm
```

1. Write out an expression using recurrence relations to express $T_k(n)$. Be sure to include all base cases.

   - Answer: $T_0(n) = 2 + n$, $T_1(0) = 0$, $T_k(0) = 1$ for $k \geq 2$, and $T_k(n) = T_{k-1}(T_k(n-1))$ for $k > 0$ and $n > 0$.

2. Solve each of the following exactly (not the Θ approximation): $T_0(n)$, $T_1(n)$, $T_2(n)$, and $T_3(n)$. Compute $T_4(1)$, $T_4(2)$, $T_4(3)$, and $T_4(4)$. Explain your work.

   - Answer:
   $T_0(n) = 2 + n$
   $T_1(n) = T_0(T_1(n-1)) = 2 + T_1(n-1) = 4 + T_1(n-2) = 2i + T_1(n-i) = 2n + T_1(0) = 2n.$
   $T_2(n) = T_1(T_2(n-1)) = 2 \cdot T_2(n-1) = 2^2 \cdot T_2(n-2) = 2^i \cdot T_2(n-i) = 2^n \cdot T_2(0) = 2^n$
   $T_3(n) = T_2(T_3(n-1)) = 2^{T_2(n-1)} = 2^{2^{T_3(n-2)}} = \left[\underbrace{2^{2^{\cdots^{2}}}}_{i}\right]^{T_3(n-i)} = \left[\underbrace{2^{2^{\cdots^{2}}}}_{n}\right]^{T_3(0)} = \underbrace{2^{2^{\cdots^{2}}}}_{n}$
   $T_4(0) = 1$. $T_4(1) = T_3(T_4(0)) = T_3(1) = \underbrace{2^{2^{\cdots^{2}}}}_{1} = 2.$
   $T_4(2) = T_3(T_4(1)) = T_3(2) = \underbrace{2^{2^{\cdots^{2}}}}_{2} = 2^2 = 4.$
   $T_4(3) = T_3(T_4(2)) = T_3(4) = \underbrace{2^{2^{\cdots^{2}}}}_{4} = 2^{2^{2^{2}}} = 2^{2^4} = 2^{16} = 65,536.$
   Note $\underbrace{2^{2^{\cdots^{2}}}}_{5} = 2^{65,536} \approx 10^{21,706}$, while the number of atoms in the universe is less than 10^{100}.
   $T_4(4) = T_3(T_4(3)) = T_3(65,536) = \underbrace{2^{2^{\cdots^{2}}}}_{65,536}.$

   Akerman's function is defined to be $A(n) = T_n(n)$. As seen $A(4)$ is is bigger than any number in the natural world. $A(5)$ is unimaginable.

3. Give a lower bound on the number of times the computation on input $A(k, n)$ adds one twice on line $n + 1 + 1$ of the code.

   - Answer: The only way that the program builds up a big number is by continually incrementing it by one. Hence, the number of times one is added is at least as huge as the value $T_k(n)$ returned.

4. Programs can stop at run time because of: 1) over flow in an integer value; 2) running out of memory; 3) running out of time. Which is likely to happen first?

   - Answer: If the machine's integers are 32 bits, then they hold a value that is about 10^{10}. Incrementing up to this value will take a long time. However, much worse than this, each two increments needs another recursive call creating a stack of about this many recursive stack frames. The machine is bound to run out of memory first.

Chapter 15

Recursive Back Tracking Algorithms

Recursive back tracking is an algorithmic technique in which all options are systematically enumerated, trying as many as required and pruning as many as possible. It is a precursor to understanding the more complex algorithmic technique of *dynamic programming algorithms* covered in Chapter 16. It is also useful for better understanding *greedy algorithms* covered in Chapter 10. Most known algorithms for a large class of computational problems referred to as *Optimization Problems* arise from these three techniques. Section 15.1 provides some theory and Sections 15.2 and 15.3 give examples.

15.1 The Theory of Recursive Back Tracking

Section 15.1.1 defines the class of optimization problems, Section 15.1.2 introduces recursive back tracking algorithms, Section 15.1.3 gives an example, Sections 15.1.5 and 15.1.4 deal with two of the technical challenges in designing such an algorithm, and finally Section 15.1.6 discusses ways of speeding them up.

15.1.1 Optimization Problems

An important and practical class of computational problems is the class of *optimization problems*. For most of these, the best known algorithm runs in exponential time for worst case inputs. Industry would pay dearly to have faster algorithms. On the other hand, the recursive backtracking algorithms designed here sometimes work sufficiently well in practice. We now formally define this class of problems.

Ingredients: An optimization problem is specified by defining instances, solutions, and costs.

- **Instances:** The *instances* are the possible inputs to the problem.
- **Solutions for Instance:** Each instance has an exponentially large set of *solutions*. A solution is *valid* if it meets a set of criteria determined by the instance at hand.
- **Cost of Solution:** Each solution has an easy to compute *cost* or *value*.

Specification of an Optimization Problem:

- **Preconditions:** The input is one instance.
- **Postconditions:** The output is one of the valid solutions for this instance with optimal (minimum or maximum as the case may be) cost. (The solution to be outputted might not be unique.)

Be Clear About These Ingredients: A common mistake is to mix them up.

Examples:

- **Longest Common Subsequence:** This is an example for which we have polynomial time algorithm.
  - **Instances:** An instance consists of two sequences, e.g., $X = \langle A, B, C, B, D, A, B\rangle$ and $Y = \langle B, D, C, A, B, A\rangle$.

Solutions: A subsequence of a sequence is a subset of the elements taken in the same order. For example, $Z = \langle B, C, A\rangle$ is a subsequence of $X = \langle A, \underline{B}, \underline{C}, B, D, \underline{A}, B\rangle$. A solution is a sequence, Z, that is a subsequence of both X and Y. For example, $Z = \langle B, C, A\rangle$ is solution because it is a subsequence common to both X and Y ($Y = \langle \underline{B}, D, \underline{C}, \underline{A}, B, A\rangle$).

Cost of Solution: The cost of a solution is the length of the common subsequence, e.g., $|Z| = 3$.

Goal: Given two sequences X and Y, the goal is to find the longest common subsequence (LCS for short). For the example given above, $Z = \langle B, C, B, A\rangle$ is the longest common subsequence.

Course Scheduling: This is an example for which we do not have polynomial time algorithm.

Instances: An instance consists of the set of courses specified by a university, the set of courses that each student requests, and the set of time slots in which courses can be offered.

Solutions: A solution for an instance is a schedule which assigns each course a time slot.

Cost of Solution: A conflict occurs when two courses are scheduled at the same time even though a student requests them both. The cost of a schedule is the number of conflicts that it has.

Goal: Given the course and student information, the goal is to find the schedule with the fewest conflicts.

15.1.2 Classifying the Solutions and the Best of the Best

Recall that in Section 11.1, we described the same recursive algorithms from a number of different abstractions: code, stack of stack frames, tree of stack frames, and friends & Strong Induction. This introductory section will do the same for a type of algorithms called *recursive back tracking algorithms*. Each abstraction focuses on a different aspect of the algorithm, providing different understandings, language, and tools from with which to draw.

Searching A Maze: To begin, imagine that, while searching a maze, we come to a fork in the road. Not knowing where any of the options lead, we accept the fact that we will have to try all of them. Bravely we head off along one of the paths. When we have completely exhausted searching in this direction, we remember the highlights of this sub-adventure and backtrack to the fork in the road where we began this discussion. Then we repeat this process, trying each of the other options from this fork. After completing them all, we determine which of these options was the best overall. This algorithm is complicated by the fact that there are many such forks for us to deal with. Hence, it is best to have friends to help us, i.e. recursion. For each option, we can get a friend to find for us the best answer for this option. We must collect together all of this information and determine which these best answers is overall best. This is will be our answer. Note that our friends will have their own forks to deal with. However, it is best not to worry about this.

Searching For the Best Animal: Now suppose we are searching for the best animal from some very large set of animals. A divide and conquer algorithm would break the search into smaller searches and delegate each smaller search to a friend. We might, for example, assign to one friend the subtask of finding for us the best vertebrate and another the best invertebrate. We will take the best of these best as our answer. This algorithm is recursive. The friend that must search for the best vertebrate also has a big task. Hence, he asks a friend to find for him the the best mammal, another the best bird, and so on. He gives us the best of his friends' best answers as his answer.

A Classification Tree of Solutions: If we were to unwind this algorithm into the tree of stackframes, we would see that it directly mirrors a tree that classifies the solutions in a way similar to the way taxonomy systematically organize animals. The root class, consisting of all of the solutions, is broken into subclasses, which are further broken into sub-subclasses. Each solution is identified with a leaf of this classification tree.

A Tree of Questions About the Solution: Another useful way of looking at this classification of solutions is that it can be defined by a tree of questions to ask about the solution. Remember the game of

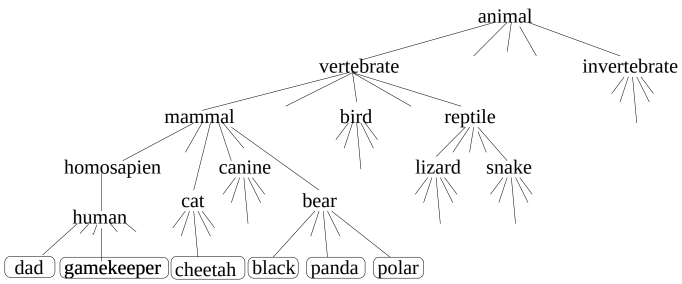


Figure 15.1: Classification Tree of Animals

20 questions? See Section 17.3. The goal is to find the optimal animal solution. The first question to ask is whether it is a vertebrate or an invertebrate. Suppose momentarily that we had a *little bird* to give us the answer, vertebrate. Then our second question would be whether the animal is a mammal, bird, reptile, or one of the other subclassifications. In the end, the sequence of answers vertebrate-mammal-cat-cheetah uniquely specifies the animal. Of course in reality, we do not have a little bird to answer our questions. Hence, we must try all answers by traversing through this entire tree of answers.

Iterating Through The Solutions To Find The Optimal One: The algorithm for iterating through all the solutions associated with the leaves of this classification tree is a simple depth-first search traversal of the tree. See Sections 8.4 and 12. The reason the algorithmic technique is called *back-tracking* is because it first tries the line of classification vertebrate-mammal-homosapiens-human-dad and later it back tracks to try vertebrate-bird-.... Even later it back tracks to tries invertebrate-....

Speeding Up the Algorithm: In the end, some friend looks at each animal. Hence, this is algorithm is not any faster than the *brute force* algorithm that simply compares the animals. However, the advantage of this algorithm is that the structure that the recursive back tracking adds can possibly be exploded to speed up the algorithm. For example, sometimes entire branches of the solution classification tree can be pruned off, perhaps because some entire classes of solutions are not highly valued or because we know that there is always at least one optimal solution does not have this particular local structure. This is similar to a greedy algorithm that knows that at least one optimal solution contains the greedy choice. In fact, greedy algorithms are recursive back tracking algorithms that prune off, in this way, all branches except the one that looks best. Memoization is another technique for speeding up the algorithm. It reuses the solutions found for similar subinstances, effectively pruning off this later branch of the classification tree. Later we will see that dynamic programming algorithms carry this idea of memoization even farther.

The Little Bird Abstraction: Personally, I like to use a "little bird" abstraction to help focus on two of the most difficult and creative parts of designing a recursive backtracking algorithm. Doing so certainly is not necessary. It is up to you.

> Little blue bird on my shoulder. It's the truth. It's actual. Every thing is satisfactual.
> Zippedy do dah, zippedy ay; wonderful feeling, wonderful day.
> From a 193? movie ??? with Shirley Temple.

The Little Bird & Friend Algorithm: Suppose I am searching for the best vertebrate. I ask the little bird "Is the best animal, a bird, mammal, reptile, or fish?" She tells me mammal. I ask my friend for the best mammal. Trusting the little bird and the friend, I give this as the best animal.

Classifying the Solutions: There is a key difference between the searching the maze example first given and the searching for the best animal example. In both, the structure of the algorithm is tied to how the maze forks or how the animals are classified. However, in the first, the forks in the maze are fixed by the problem. In the second, the algorithm designer is able to choose how to classify the animals. This is one of the two most difficult and creative parts of the algorithm design process. It dictates the entire structure of the algorithm, which in turns dictates how well the algorithm can be sped up. I abstract the task of deciding how to classify the solutions by asking *a little bird* a question. When I am searching for best vertebrate, I classify them, by asking "Is the best vertebrate, a mammal, a bird, a reptile, or a fish?" We are allowed to ask the bird any question about the solution that we are looking for as long as the question does not have too many possible answers. Not having a little bird, we must try all these possible answers. But it can be fun to temporarily pretend that the little bird gives us the correct answer.

Constructing a Sub-Instance for a Friend: We must get a friend to find us the best mammal. Because a friend is really a recursive call, we must construct a smaller instance to the same search problem to give him. The second creative part of designing a recursive backtracking algorithm is how to express the problem "Find the best mammal" as a sub-instance. The little bird abstraction helps again. We pretend that she answered "mammal". Trusting (at least temporarily) in her answer, helps us focus, on the fact that we are now only considering mammals. This helps us to design a related sub-instance.

Non-Determinism: You may have previously studied Non-Deterministic Finite Automaton (NFA) and Non-Deterministic Turing Machines. One way of viewing these machines is that a higher power provides them help telling them which way to go. The little bird can be viewed as a little higher power. In all of these cases, the purpose is to provide another level of abstraction within which it is easier to design and to understand the creative parts of the algorithm.

A Flock of Birds: I sometimes imagine that instead of a single trustworthy bird, we have a flock of K birds that are less trustworthy. Each gives us a different answer $k \in [1, K]$. We give each the benefit of doubt and try his answer. Because there does exist an answer that the little bird would have given us, at least one of these birds must have been telling us the truth. The remaining question is whether we are able to pick out the bird that is trust worthy.

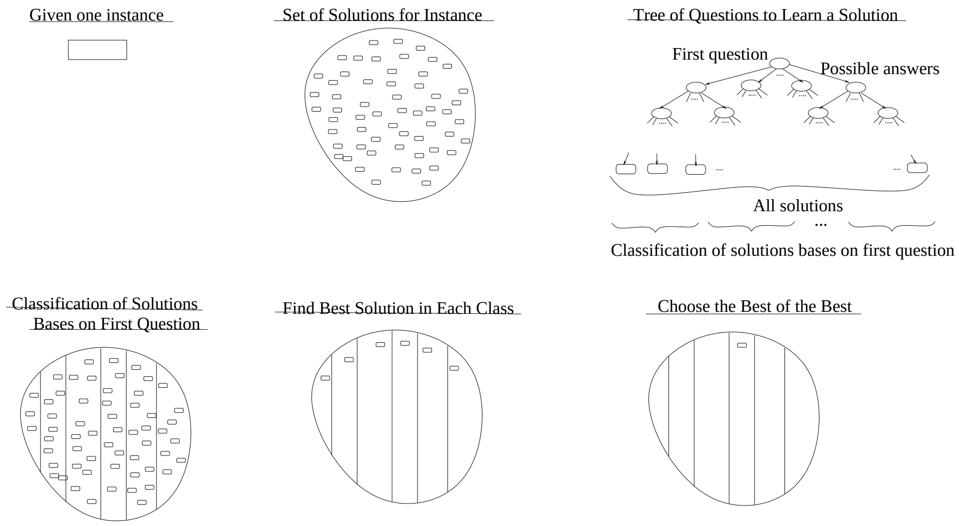


Figure 15.2: Classifying solutions and taking the best of the best

The Recursive Back-Tracking Algorithm: The following are the steps for you to follow in order to find an optimal solution for the instance that you have been given and the solution's cost.

Base Cases and Recursive Friends: If your instance is sufficiently small or has sufficiently few solutions find an optimal solution for it in a brute-force way. Otherwise, assume that you have friends who can magically find an optimal solution for any instance to the problem that is strictly "smaller" than your instance.

Question for The Little Bird: Formulate one small question about the optimal solution that you are searching for. The question should be such that if you had a powerful *little bird* to give you a correct answer, then this answer would greatly reduce your search. Without a little bird, the different answers to this question partitions the class of all solutions for your instance into subclasses.

Try All Answers: Given that you do not have little a bird, you must try all possible answers. One at a time, for each of the K answers that the little bird may have given, i.e. for each $k \in [1..K]$, do the following.

Pretend: Pretend that the little bird gave you the answer k. This narrows your search.

Help From Your Friend: Your task now is to find the best solution from among those solutions consistent with the k^{th} bird's answer. You get help from a friend to find such a solution and its cost.

Best of the Best: Your remaining task is simply to select and return the best of these best solutions along with its cost.

Code:

algorithm $Alg(I)$

$\langle$***pre−cond***$\rangle$**:** I is an instance to the problem.

$\langle$***post−cond***$\rangle$**:** $optSol$ is one of the optimal solutions for the instance I and $optCost$ is its cost.

```
begin
    if( I is small ) then
        return( brute force answer )
    else
        % Loop over the possible bird answers
        for k = 1 to K
            % Find the best solution consistent with the answer k.
            ⟨optSol_k, optCost_k⟩ = Alg_try(I, k)
        end for
        % Find the best of the best.
        k_min = "a k that minimizes optCost_k"
        optSol = optSol_{k_min}
        optCost = optCost_{k_min}
        return ⟨optSol, optCost⟩
    end if
end algorithm
```

Note that the above code is *not* recursive, because the same routine Alg in not called. In order for it to be recursive, the routine Alg_{try} will have to be written that recursively calls Alg again. The following code includes the main steps. An example of these steps will be given in Section 15.1.3 and then the more theoretical aspects of this task will be discussed in Section 15.1.4

algorithm $Alg_{try}(I,\ k)$

$\langle$***pre−cond***$\rangle$**:** I is any instance to the problem and k is any answer to the question asked about its solution.

$\langle$***post−cond***$\rangle$**:** $optSol$ is one of best solutions for the instance I from amongst those consistent with the answer k and $optCost$ is its cost.

```
begin
    subI is constructed from I and k
```

$\langle optSubSol, optSubCost\rangle = Alg\,(subI)$
$optSol = \langle k, optSubSol\rangle$
$optCost$ is computed from k and $optSubCost$
return $\langle optSol, optCost\rangle$
end algorithm

Exercise 15.1.1 *(See solution in Section 20) An instance may have many optimal solutions with exactly the same cost. The postcondition of the problem allows any one of these to become output. Which line of code in the above algorithm chooses which of these optimal solutions will be selected?*

15.1.3 Example: The Shortest Path within a Leveled Graph

Let us now continue our example of searching mazes started at the beginning of Section 15.1.2. We represent a maze by a graph. The nodes act as forks in the road and the edges act as the corridors between them. There are many versions of this graph search problem. We have already seen a few of them in Section 8, namely breadth-first search, depth-first search, and shortest-weighted paths. Here we will develop a recursive backtracking algorithm for a version of the shortest-weighted paths problem. This same example will be used to introduce memoization in Section 16.2.2 and dynamic programming algorithms in Chapter 16.

Shortest Weighted Path within a Directed Leveled Graph: We generalize the shortest-weighted paths problem by allowing negative weights on the edges and simplify it by requiring the input graph to be leveled.

Instances: An instance (input) consists of $\langle G, s, t\rangle$, where G is a weighted directed layered graph G, s is a specified source node and t is a specified destination node. See Figure 15.3.a. The graph G has n nodes. Each node has maximum in and out degree d. Each edge $<v_i, v_j>$ is labeled with a real-valued (possibly negative) weight $w_{<v_i,v_j>}$. The nodes are partitioned into levels so that each edge is directed from some node to a node in a lower level. (This prevents cycles.) It is easiest to assume that the nodes are ordered such that an edge can only go from node v_i to node v_j if $i < j$.

Solutions: A solution for an instance is a path from the source node s to destination node t.

Cost of Solution: The cost of a solution is the sum of the weights of the edges within the path.

Goal: Given a graph G, the goal is to find a path with minimum total weight from s to t.

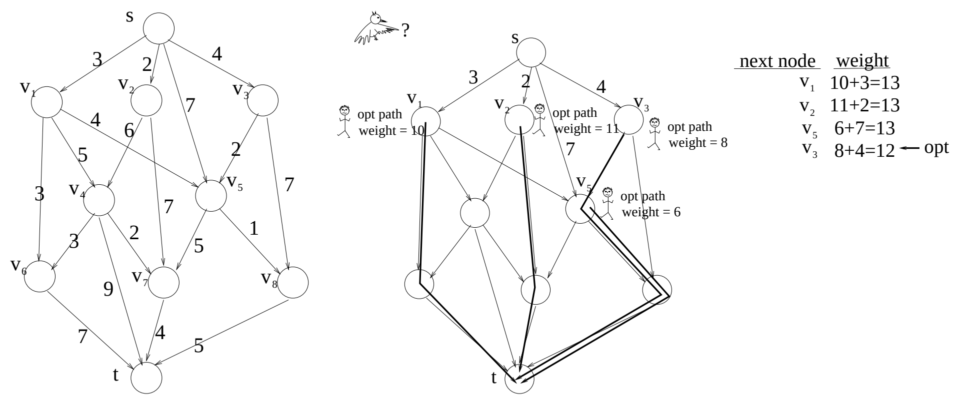


Figure 15.3: a: The directed layered weighted graph G used as a running example. b: The recursive backtracking algorithm

Brute Force Algorithm: The problem with simply trying all paths is that there may be an exponential number of them.

Exercise 15.1.2 *Give a directed leveled graph on n nodes that has a small number of edges and as many paths from s to t as possible.*

Bird & Friend Algorithm: The first step in developing a recursive backtracking algorithm is to classify the solutions of the given instance $I = \langle G, s, t\rangle$. I classify paths by asking the little bird, "Which edge should we take first?" Suppose for now that the bird answers $<s, v_1>$. I then go on to ask a friend, "Which is the best path from s to t that starts with $<s, v_1>$?" Friend answers $\langle s, v_1, v_6, t\rangle$. If I trust the little bird and the friend, I give this as the best path. If I don't trust the little bird, I can try all of the possible answers that the bird may have given. This, however, is not the immediate issue.

Constructing Sub-Instances: A friend is really a recursive call to the same search problem. Hence, I must express my question to him as a small instance of the same computational problem. I formulate this question as follows. I start by (at least temporarily) trusting the little bird and committing to the first edge being $<s, v_1>$. Given this, it is quite natural for me to take step from node s along the edge $<s, v_1>$ to the node v_1. Standing here, the natural question to ask my friend is how to get to t from v_1. Luckily, "Which is the best path from v_1 to t?", expressed as $\langle G, v_1, t\rangle$ is a subinstance to the same computational problem. We will denote this by $subI$.

Constructing a Solution for My Instance: My friend faithfully will give me the path $optSubSol = \langle v_1, v_6, t\rangle$, this being a best path from v_1 to t. The problem is that this is not a solution for my instance $\langle G, s, t\rangle$, because it is not, in itself, a path from s to t. The path from s is formed by first taking the step from s to v_i and then following the best path from there to t. Hence, I construct a solution *optSol* for my instance from this optimal subsolution *optSubSol* by reattaching the answer given by the bird. In other words, the path $\langle s, v_1, v_6, t\rangle$ from s to t is formulated by combining the edge $<s, v_1>$ with the subpath $\langle v_1, v_6, t\rangle$.

Costs of Solutions and Sub-Solutions: We must also return the cost of our solution. The cost of our path is the cost of the edge $<s, v_1>$ plus the cost of the path from v_1 to t. Luckily, our friend gives us that the cost of this subpath is 10. The cost of the edge $<s, v_1>$ is 3. Hence, we conclude that the total weight of our path is $3 + 10 = 13$.

Optimality: If we trust both the bird and the friend, we conclude that this path from s to t is a best path. It turns out that because our little bird gave us the wrong first edge, this is not the best path from s to t. However, our work was not wasted, because we did succeed in finding the best path from amongst those that start with the edge $<s, v_1>$. Not trusting the little bird, I repeat this process finding a best path starting with each of $<s, v_2>$, $<s, v_3>$ and $<s, v_4>$. At least one of these four paths must be an over all best path. We give the best of these best as the over all best path.

Code:

algorithm $LeveledGraph\,(G, s, t)$

$\langle \boldsymbol{pre-cond}\rangle$: G is a weighted directed layered graph and s and t are nodes.

$\langle \boldsymbol{post-cond}\rangle$: *optSol* is a path with minimum total weight from s to t and *optCost* is its weight.

```
begin
    if(s = t) then
        return ⟨∅, 0⟩
    else
        % Loop over possible bird answers.
        for each node v_k with an edge from s
            % Get help from friend
            ⟨optSubSol, optSubCost⟩ = LeveledGraph(⟨G, v_k, t⟩)
            optSol_k = ⟨s, optSubSol⟩
            optCost_k = w_⟨s,v_k⟩ + optSubCost
        end for
```

```
            % Take the best bird answer.
            k_min = "a k that minimizes optCost_k"
            optSol = optSol_{k_min}
            optCost = optCost_{k_min}
            return ⟨optSol, optCost⟩
        end if
    end algorithm
```

Memoization and Dynamic Programming: This recursive backtracking algorithm faithfully enumerates all of the paths from s to t and hence requires exponential time. In Section 16.2.2, we will use memoization techniques to mechanically convert this algorithm into a dynamic programming algorithm that runs in polynomial time.

15.1.4 SubInstances and SubSolutions

Getting a trust worthy answer from the little bird, narrows our search problem down to the task of finding the best solution from among those solutions consistent with this answer. It would be great if we could simply ask a friend to find us such as solution, however, we are only allowed to ask our friend to solve subinstances of the original computational problem. Our task within this section is to formulate a subinstance to our computational problem such that the search for its optimal solutions some how parallels our narrowed search task. This will explain the algorithm $Alg_{try}(I,\ k)$ that is given at the end of Section 15.1.2.

The Recursive Structure of the Problem: In order to be able to design a recursive backtracking algorithm for a optimization problem, the problem needs to have a recursive structure. The key property is that in order for a solution of the instance to be optimal, some part of the solution must itself be optimal. The computational problem has a recursive structure if the task of finding an optimal way to construct this part of the solution is a subinstance of the same computational problem.

Leveled Graph: For a path from s to t to be optimal, the subpath from some v_i to some v_j along the path must itself be an optimal path between these nodes. The computational problem has a recursive structure because the task of finding an optimal way to construct this part of the path is a subinstance of the same computational problem.

Question from Answer: Sometimes it is a challenge to know what subinstance to ask our friend. It turns out that it is easier to know what answer (subsolution) that we want from him. Knowing the answer we want will be a huge hint as to what the question should be.

Each Solution as a Sequence of Answers: One task that you as the algorithm designer must do is to organize the information needed to specify a solution into a sequence of fields, $sol = \langle field_1, field_2, \ldots, field_m\rangle$.

Best Animal: Each solution consists of an animal, which we will identify with the sequence of answers to the little bird's questions, sol = *vertebrate-mammal-cat-cheetah*.

Leveled Graph: A solution consists of a path, $\langle s, v_1, v_6, t\rangle$, which we will identify with the sequence of edges $sol = \langle <s, v_1>, <v_1, v_6>, <v_6, t>\rangle = \langle field_1, field_2, \ldots, field_m\rangle$.

Bird's Question and Remaining Task: The algorithm asks the little bird for the first field $field_1$ of one of the instance's optimal solutions and asks the friend for the remaining fields $\langle field_2, \ldots, field_m\rangle$. We will let k denote the answer provided by the bird and $optSubSol$ that provided by the friend. Given both, the algorithm formulates the final solution by simply concatenating these two parts together, namely $optSol = \langle k, optSubSol\rangle = \langle field_1, \langle field_2, \ldots, field_m\rangle\rangle$. Note this is a line within the code for $Alg_{try}(I,\ k)$ in Section 15.1.2.

Leveled Graph: Asking for the first field of an optimal solution $optSol = \langle <s, v_1>, <v_1, v_6>, <v_6, t>\rangle$ amounts to asking for the first edge that the path should take. The bird answers $k = <s, v_1>$. The friend provides $optSubSol = \langle <v_1, v_6>, <v_6, t>\rangle$. Concatenating these forms our solution.

Formulating the SubInstance: We formulate the subinstance that we will give to our friend by finding an instance of the computational problem whose optimal solution is $optSubSol = \langle field_2, \ldots, field_m\rangle$. If one wants to get formal, the instance is that whose set of valid solutions is $setSubSol = \{subSol \mid \langle k, subSol\rangle \in setSol\}$.

Leveled Graph: The instance whose solution is $optSubSol = \langle <v_1, v_6>, <v_6, t> \rangle$ is $\langle G, v_1, t\rangle$ asking for the optimal solution from v_1 to t.

Costs Solutions: The algorithm, in addition to finding an optimal solution $optSol = \langle field_1, \langle field_2, \ldots, field_m\rangle\rangle$ for the instance I, must also produce the cost of this solution. To be helpful, the friend provides the cost of his solution, $optSubSol$. Do to the recursive structure of the problem, the costs of these solutions $optSol = \langle field_1, \langle field_2, \ldots, field_m\rangle\rangle$ and $optSubSol = \langle field_2, \ldots, field_m\rangle$ usually differ in some uniform way. For example, often the cost is the sum of the costs of the fields, i.e. $cost(optSol) = \sum_{i=1}^{m} cost(field_i)$. In this case we have that $cost(optSol) = cost(field_1) + cost(optSubSol)$.

Leveled Graph: The cost of of a path from s to t is the cost of the first edge plus the cost of the rest of the path.

Formal Proof of Correctness:

Recursive Structure of Costs: In order for this recursive back tracking method to solve an optimization problem, the costs that the problem allocates to the solutions must have the following recursive structure. Consider two solutions $sol = \langle k, subSol\rangle$ and $sol' = \langle k, subSol'\rangle$ both consistent with the same bird's answer k. If the given cost function dictates that the solution sol is better than the solution sol', then the subsolution $subSol$ of sol will also be better than the subsolution $subSol'$ of sol'. This ensures that any optimal subsolution of the subinstance leads to an optimal solution of the original instance.

Theorem: The solution $optSol$ returned is a best solution for I from amongst those that are consistent with the information k provided by the bird.

Proof: By way of contradiction, assume not. Then, there must be another solution $betterSol$ consistent with k whose cost is strictly better then that for $optSol$. From the way we constructed our friend's subinstance $subI$, this better solution must have the form $betterSol = \langle k, betterSubSol\rangle$ where $betterSubSol$ is a solution for $subI$. We ensured that the costs are such that because the cost of $betterSol$ is better than that of $optSol$, it follows that the cost of $betterSubSol$ is better than that of $optSubSol$. This contradicts the fact that $optSubSol$ is an optimal solution for the subinstance $subI$. Recall that within the friend level of abstraction, we can trust the friend to provide an optimal solution to the subinstance $subI$. (We proved this in Section 11.1.4 using strong induction.)

Size of an Instance: In order to avoid recursing indefinitely, the subinstance that you give your friend must be *smaller* than your own instance according to some measure of size. By the way that we formulated the subinstance, we know that its valid solutions $subSol = \langle field_2, \ldots, field_m\rangle$ are shorter than the valid solutions $sol = \langle field_1, field_2, \ldots, field_m\rangle$ of the instance. Hence, a reasonable measure of the size of an instance is the length of its longest valid solution. This measure only fails to work when an instance has valid solutions that are infinitely long.

Leveled Graph: The size of the instance $\langle G, s, t\rangle$ is the length of the longest path or simply the number of levels between s and t. Given this, the size of the subinstance $\langle G, v_i, t\rangle$, which is the number of levels between v_i and t, is smaller.

More Complex Algorithms: Most recursive backtracking, dynamic programming, and greedy algorithms fit into the structure defined above. However, a few have the following more complex structure. See Sections 16.3.6 and 16.3.7.

Each Solution as a Tree of Answers: Above, the information specifying a solution is partitioned into a number of fields and these fields are ordered into a sequence. For some more complex algorithms, these fields are organized into a tree instead of into a sequence. For example, if the problem is to find the best binary search tree then each solution is a binary search tree. Hence, it is quite reasonable that the fields are the nodes of the tree and these fields are organized as in the tree itself.

Bird's Question and Remaining Task: Instead of the first field, the algorithm asks the little bird to tell it the field at the root of one of the instance's optimal solutions. Given this information, the remaining task is to fill in each of the solution's subtrees. Instead of asking one friend to fill in this information, a separate friend will be asked for each of the solution's subtrees.

15.1.5 The Question For the Little Bird

Coming up with which question is used to classify the solutions is one of the main creative steps in designing either a recursive backtracking algorithm, a dynamic programming algorithm, or a greedy algorithm. This section examines some more advanced techniques that might be used.

Local vs Global Considerations: One of the reasons that optimization problems are difficult is that, although we are able to make what we call *local* observations and decisions, it is hard to see the *global* consequences of these decisions.

Leveled Graph: Consider the leveled graph example from Section 15.1.3. Which edge out of s is cheapest is a local question. Which path is the overall cheapest is a global question. We were tempted to follow the cheapest edge out of the source s. However, this greedy approach does not work. Sometimes one can arrive at a better over all path by starting with a first edge that is not the cheapest. This local *sacrifice* could globally lead to a better overall solution.

Ask About A Local Property: The question that we ask the bird is about some *local* property of the solution. For example:

- If the solution is a sequence of objects, a good question would be, "What is the first object in the sequence?"

  Example: If the solution is a path though a graph, we might ask, "What is the first edge in the path?"

- If the instance is a sequence of objects and a solution is a subset of these object, a good question would be, "Is the first object of the instance included in the optimal solution?"

  Example: In the Longest Common Subsequence problem, Section 16.3.2, we will ask, "Is the first character of either X or Y included in Z?"

- If a solution is a binary tree of objects, a good question would be, "What object is at the root of the tree?"

  Example: In the Best Binary Search Tree problem, Section 16.3.6, we will ask, "Which key is at the root of the tree?"

In contrast, asking the bird for the number of edges in the best path in the leveled graph is a global not a local question.

The Number K of Different Bird Answers: You can only ask the bird a little question. (It is only a *little* bird.) By little, I mean the following. Together with your question, you provide the little bird with a list $A_1, A_2, \ldots, A_K$ of possible answers. The little bird answers simply by returning the index $k \in [1..K]$ of her answer. In a little question, the number K of different answers that the bird might give must be small. The smaller K is, the more efficient the final algorithm will be.

Leveled Graph: In the leveled graph algorithm presented, we asked the bird which edge to take from node s. Because the maximum number of edges out of a node is d, we know that there are at most d different answers that the bird could give. This gives $K = d$.

Brute Force: The obvious question to ask the little bird is for her to tell you an entire optimal solution. However, the number of solutions for your instance I is likely exponential; each solution is a possible answer. Hence, K would be exponential. After getting rid of the bird, the resulting algorithm would be the usual brute force algorithm.

Repeated Questions: Although you want to avoid thinking about it, each of your recursive friends will have to ask his little bird a similar question. Hence, you should choose a question that provides a reasonable follow-up question of a similar form. For example:

- "What is the second object in the sequence?"
- "Is the second object of the instance included in the optimal solution?"
- "What is the root in the left/right subtree?"

In contrast, asking the bird for the number of edges in the best path in the leveled graph does not have good follow up question.

Reverse Order: We will see in Chapter 16 that the Dynamic Programming technique reverses the recursive backtracking algorithm by completing the subinstances from smallest to largest. In order to have the final algorithm move forward through the local structure of the solution, the recursive backtracking algorithm needs to go backwards through the local structure of the solution. To do this, we ask the bird about the last object in the optimal solution instead of about the first one. For example:

- "What is the last object in the sequence?"
- "Is the last object of the instance included in the optimal solution?"
- We still ask about the root. It is not useful to ask about the leaves.

The Required Creativity: Choosing the question to asked the little bird requires some creativity. Like any creative skill, it is learned by looking at other people's examples and trying a lot of your own. On the other hand, there are not that many different questions that you might ask the bird. Hence, you can design an algorithm using each possible question and use the best of these.

Common Pitfalls: In one version of the scramble game, an input instance consists of a set of letters and a board and the goal is to find a word that returns the most points. A student described the following recursive backtracking algorithm for it. The bird provides the best word out of the list of letters. The friend provides the best place on the board to put the word.

Bad Question for Bird: What is being asked of the bird is not a "little question". The bird is doing most of the work for you.

Bad Question for Friend: What is being asked of the friend needs to be a subinstance of the same problem as that of the given instance, not a subproblem of the given problem.

15.1.6 Speeding Up the Algorithm

Though structured, recursive back tracking algorithm as constructed so far examine each and every solution for the given instance. Hence, they are not any faster than the *brute force* algorithm that simply compares all solutions. We will now discuss and give examples of how this structure can be exploded to speed up the algorithm. The key ways of speeding up a recursive back tracking algorithm are the following: (1) pruning off entire branches of the tree; (2) taking only the greedy option; and (3) eliminating repeated subinstances with memorization.

Pruning: Sometimes entire branches of the solution classification tree can be pruned off. The following are typical reasons.

Not Valid or Highly Valued Solutions: When the algorithm arrives at the root of the reptile tree it might realize that all solutions within this subtree are not valid or are not rated sufficiently high to be optimal. Perhaps, we have already found a solution provably better than all of these. Hence, the algorithm prune this entire subtree from its search. See Section 15.2 for more examples.

Structure of An Optimal Solution: Some times we are able to prove that there is at least one optimal solution that has some particular local structure. This allows us to prune away any solution that does not have this local structure. Note that we do not require that all the optimal solutions of the instance have the property, only one. The reason is that ties can be broken arbitrarily. Hence, the algorithm can always choose the solution that has it. This is similar to a greedy algorithm that knows that at least one optimal solution contains the greedy choice.

Modifying Solutions: Suppose that it could be proven that for every reptile there exists at least one mammal that is rated at least as high. This would allow the algorithm to avoid iterating through all the reptiles, because if it happened to be the case that some reptile had the highest rating, then there would be an equivalent valued mammal that could be substituted. The general method for proving this is as follows. A scheme is found for taking an arbitrary reptile and modifying it, maybe by adding a wig of hair, and making it into a mammal. Then it is proved that this new mammal is valid and is rated at least as high as the original reptile. This proves that there exists a mammal, namely this newly created one, that is rated at least as high. See Section 15.3 for more examples.

Greedy Algorithms: This next way of speeding up a recursive back tracking algorithm is even more extreme than pruning off the odd branch. It is called a greedy algorithm. When ever the algorithm has a choice as which branch of the classification tree to go down, instead of iterating through all of the options it just goes only for the one that looks best according to some greedy criteria. In this way the algorithm follows only one path down the classification tree: animal-vertebrate-mammal-cat-cheetah. The reason that the algorithm works is because all branches that veer off this path have been pruned away as described above. Greedy algorithms are covered in Chapter 10.

Eliminating Repeated SubInstances With Memorization: The final way of speeding up a recursive back tracking algorithm remembers (memorization) the solutions for the subinstances solved by recursive calls that are made along the way so that if ever a sufficiently similar subinstance needs to be solved, the same answer can be used. This effectively prunes off this later branch of the classification tree.

Leveled Graph: Memorization is very useful for this problem as the same node gets reached many times. We cover this in Section 16.2.1.

Best Animal: For example, if we add color as an additional classification, the subclasses animal-green-vertebrate-reptile-lizard-chameleon and animal-brown-vertebrate-reptile-lizard-chameleon really are the same because each chameleon is able to take on either a green or a brown aspect, but this aspect is irrelevant to the computational problem.

Dynamic programming algorithms, studied in Chapter 16, take this idea one step further. Once the set of subinstance that need to be solved is determined, the algorithm no longer traverses recursively through the classification tree but simply solves each of the required subinstance in smallest to largest order.

15.2 Pruning Invalid Solutions

In this section, we give three examples of solving an optimization problem using recursive backtracking. Each will demonstrate, among other things, how the set of solutions can be organized into a classification tree and how branches of this tree containing invalid solutions can be pruned. See Section 15.1.2.

15.2.1 Satisfiability

A famous optimization problem is called *satisfiability* or *SAT* for short. It is one of the basic problems arising in many fields. The recursive backtracking algorithm given here is refereed to the *Davis Putnum* algorithm. It is an example of an algorithm whose running time is exponential time for worst case inputs, yet in many practical situations can work well. This algorithm is one of basic algorithms underlying automated theorem proving and robot path planning among other things.

The Satisfiability Problem:

Instances: An instance (input) consists of a set of constraints on the assignment to the binary variables $x_1, x_2, \ldots, x_n$. A typical constraint might be $\langle x_1 \text{ or } \overline{x_3} \text{ or } x_8 \rangle$, meaning ($x_1 = 1$ or $x_3 = 0$ or $x_8 = 1$) or equivalently that either x_1 is true, x_3 is false, or x_8 is true.

Solutions: Each of the 2^n assignments is a possible solution. An assignment is valid for the given instance, if it satisfies all of the constraints.

Cost of Solution: An assignment is assigned the value one if it satisfies all of the constraints and the value zero otherwise.

Goal: Given the constraints, the goal is to find a satisfying assignment.

Iterating Through the Solutions: The brute force algorithm simply tries each of the 2^n assignments of the variables. Before reading on, think about how you would non-recursively iterate through all of these solutions. Even this simplest of examples is surprisingly hard.

Nested Loops: The obvious algorithm is to have n nested loops each going from 0 to 1. However, this requires knowing the value of n before compile time, which is not too likely.

Incrementing Binary Numbers: Another options is to treat the assignment as an n bit binary number and then loop through the 2^n assignments by incrementing this binary number each iteration.

Recursive Algorithm: The recursive backtracking technique is able to iterate through the solutions with much less effort in coding. First the algorithm commits to assigning $x_1 = 0$ and recursively iterates through the 2^{n-1} assignments of the remaining variables. Then the algorithm back tracks repeating these steps with the choice $x_1 = 1$. Viewed another way, the first little bird question about the solutions is whether the first variable x_1 is set to zero or one, the second question asks about the second variable x_2, and so on. The 2^n assignments of the variables $x_1, x_2, \ldots, x_n$ are associated with the 2^n leaves of the complete binary tree with depth n. A given path from the root to a leaf commits each variable x_i to being either zero or one by having the path turn to either the left or to the right when reaching the i^{th} level.

Instances and SubInstances: Given an instance, the recursive algorithm must construct two subinstances for his friend to recurse with. There are two techniques for doing this.

Narrowing the Class of Solutions: A typical instance associated with some node of the classification tree will specify the original set of constraints and an assignment of a prefix $x_1, x_2, \ldots, x_k$ of the variables. A solution is an assignment of the n variables that is consistent with this prefix assignment. Traversing a step further down the classification tree further narrows the set of solutions.

Reducing the Instance: Given an instance consisting of a number of constraints on n variables, we first try assigning $x_1 = 0$. The subinstance to be given to the first friend will be the constraints on remaining variables given that $x_1 = 0$. For example, if one of our original constraints is $\langle x_1 \text{ or } \overline{x_3} \text{ or } x_8 \rangle$, then after assigning $x_1 = 0$, the reduced constraint will be $\langle \overline{x_3} \text{ or } x_8 \rangle$. This is because it is no longer possible for x_1 to be true, leaving that one of $\overline{x_3}$ or x_8 must be true. On the other hand, after assigning $x_1 = 1$, the original constraint is satisfied independent of the values of the other variables, and hence this constraint can be removed.

Pruning: This recursive backtracking algorithm for *SAT* can be speed up. This can either be viewed globally as a pruning off of entire branches of the classification tree or locally as seeing that some subinstances after they have been sufficiently reduced are trivial to solve.

Pruning Off Branches The Tree: Consider the node of the classification tree arrived at down the subpath $x_1 = 0$, $x_2 = 1$, $x_3 = 1$, $x_4 = 0, \ldots, x_8 = 0$. All of the assignment solutions consistent with this partial assignment fail to satisfy the constraint $\langle x_1 \textit{ or } \overline{x_3} \textit{ or } x_8\rangle$. Hence, this entire subtree can be pruned off.

Trivial SubInstances: When the algorithm tries to assign $x_1 = 0$, the constraint $\langle x_1 \textit{ or } \overline{x_3} \textit{ or } x_8\rangle$ is reduced to $\langle \overline{x_3} \textit{ or } x_8\rangle$. Assigning $x_2 = 1$, does not change this particular constraint. Assigning $x_3 = 1$, reduces this constraint further to simply $\langle x_8\rangle$ stating that x_8 must be true. Finally, when the algorithm is considering the value for x_8, it sees from this constraint that x_8 is *forced* to be one. Hence, the $x_8 = 1$ friend is called, but the $x_8 = 0$ friend is not.

Davis Putnum: The above algorithm branches on the values of each variable, $x_1, x_2, \ldots, x_n$, in order. However, there is no particular reason that this order needs to be fixed. Each branch of the recursive algorithm can dynamically use some heuristic to decide which variable to branch on next. For example, if there is a variable like x_8 above whose assignment is forced by some constraint, then clearly this assignment should be done immediately. Doing so removes this variable from all the other constraints, simplifying the instance. More over, if the algorithm branched on $x_4, \ldots, x_7$ before the forcing of x_8, then this same forcing would need to be repeated within all 2^4 of these branches.

If there are no variables to force, a common strategy is to branch on the variable that appears in the largest number of constrains. The thinking is that the removal of this variable may lead to the most simplification of the instance.

An example of how different branches may set the variables in a different order is the following. Suppose that $\langle x_1 \textit{ or } x_2\rangle$ and $\langle \overline{x_1} \textit{ or } x_3\rangle$ are two of the constraints. Assigning $x_1 = 0$ will simplify the first constraint to $\langle x_2\rangle$ and remove the second constraint. The next step would be to force $x_2 = 1$. On the other hand, assigning $x_1 = 1$ will simplify the second constraint to forcing $x_3 = 1$.

Code:

algorithm $DavisPutnum\,(c)$

$\langle$***pre−cond***$\rangle$**:** c is a set of constraints on the assignment to $\vec{x}$.

$\langle$***post−cond***$\rangle$**:** If possible, *optSol* is a satisfying assignment and *optCost* is one. Otherwise *optCost* is zero.

```
begin
    if( c has no constraints or no variables ) then
        % c is trivially satisfiable
        return ⟨∅, 1⟩
    else if( c has both a constraint forcing a variable x_i to 0 and one forcing the same variable to 1 ) then
        % c is trivially not satisfiable
        return ⟨∅, 0⟩
    else
        for any variable forced by a constraint to some value
            substitute this value into c.
        let x_i be the variable that appears the most often in c
        % Loop over the possible bird answers
        for k = 0 to 1
            % Get help from friend
            let c' be the constraints c with k substituted in for x_i
            ⟨optSubSol, optSubCost⟩ = DavisPutnum (c')
            optSol_k = ⟨forced values, x_i = k, optSubSol⟩
```

$optCost_k = optSubCost$
end for
% Take the best bird answer.
k_{max} = "a k that maximizes $optCost_k$"
$optSol = optSol_{k_{max}}$
$optCost = optCost_{k_{max}}$
return $\langle optSol, optCost\rangle$
end if
end algorithm

Running Time: If no pruning is done, then clearly the running time is $\Omega(2^n)$ as all 2^n assignments are tried. Considerable pruning needs to occur to make the algorithm polynomial time. Certainly in the worst case, the running time is $2^{\Theta(n)}$. In practice, however, the algorithm can be quite fast. For example, suppose that the instance is chosen randomly by choosing m constraints, each of which is the *or* of three variables or there negations, eg. $\langle x_1 \text{ or } \overline{x_3} \text{ or } x_8\rangle$. If few constraints are chosen, say m is less than about $3n$, then with very high probability there are many satisfying assignments and the algorithm quickly finds one of these assignments. If lots of constraints are chosen, say m is at least n^2, then with very high probability there are many conflicting constraints preventing there from being any satisfying assignments and the algorithm quickly finds one of these contradictions. On the other hand, if the number of constraints chosen is between these thresholds, then it has been proved that the Davis Putnum algorithm takes exponential time.

15.2.2 Scrabble

Consider the following scrabble problem. An instance consists of a set of letters and a dictionary. A solution consists of a permutation of a subset of the given letters. A solution is valid if it is in the dictionary. The value of a solution is given by its placement on the board. The goal is to find a highest point word that is in the dictionary.

The simple brute force algorithm searches the dictionary for each permutation of each subset of the letters. The back-tracking algorithm tries all of the possibilities for the first letter and then recurses. Each of these stack frames tries all of the remaining possibilities for the second letter, and so on. This can be pruned by observing that if the word constructed so far, eg. "xq", is not the first letters of any word in the dictionary, then there is no need for this stack frame to recurse any further. (Another improvement on the running time ensures that the words are searched for in the dictionary in alphabetical order.)

15.2.3 Queens

The following is a fun exercise to help you trace through a recursive back tracking algorithm. It is called the Queen's Problem. Physically get yourself (or make on paper) a chess board and eight tokens to act as queens. The goal is to place all eight queens on the board in a way such that no pieces moving like queen along a row, column, or diagonal is able to capture any other piece.

The recursive backtracking algorithm is as follows. First it observes that each of the eight rows can have at most one queen or else they will capture each other. Hence each row must have one of the eight queens. Given a placement of queens in the first few rows, a stack frame tries each of the legal placements of a queen in the next row. For each such placements, the algorithm recurses.

Code:

algorithm *Queens* (C, row)

$\langle$***pre−cond***$\rangle$: C is a chess board containing a queen on the first $row - 1$ rows in such a way that no two can capture each other. The remaining rows have no queen.

$\langle$***post−cond***$\rangle$: All legal arrangements of the 8 queens consistent with this initial placement are printed out.

```
begin
    if( row > 8 ) then
        print(C)
    else
        loop col = 1...8
            Place a queen on location C(row, col)
            if( this creates a conflict) then
                Do not pursue this option further. Do not recurse.
                Note this prunes off this entire branch of the recursive tree
            else
                Queens (C, row + 1)
            end if
            back track removing the queen from location C(row, col)
        end loop
    end if
end algorithm
```

Trace this algorithm. It is not difficult to do because there is an active stack frame for each queen currently on the board. You start by placing a queen on each row, one at a time, in the left most legal position until you get stuck. Then when ever a queen cannot be placed on a row or moves off the right of a row, you move the queen on the row above until it is in the next legal spot.

Exercise 15.2.1 *(See solution in Section 20) Trace this algorithm. What are the first dozen legal outputs for the algorithm. To save time record the positions stated in a given output by the vector* $\langle c_1, c_2, \ldots, c_8 \rangle$ *where for each* $r \in [1..8]$ *there is a queen at location* $C(r, c_r)$*. To save more time, note that the first two or three queens do not move so fast. Hence, it might be worth it to draw a board with all squares conflicting with these crossed out.*

Exercise 15.2.2 *Consider the same Queens algorithm placing* n *queens on an* n *by* n *board instead of only 8. Give a reasonable upper and lower bound on the running time of this algorithm after all the pruning occurs.*

15.3 Pruning Equally Valued Solutions

This section demonstrates how a branch of the classification tree can be pruned when for each solution contained in the branch, there is another solution outside of it whose value is at least as good.

15.3.1 Recursive Depth First Search

We will now consider another version of the graph search problem. But now instead of looking for the shortest weighted path for the source s to the sink t, we will only try to reach each node once. It makes sense that to accomplish this, many of the exponential number of possible paths do not need to be considered. The resulting recursive backtracking algorithm is called *recursive depth-first-search*. This algorithm directly mirrors iterative depth-first-search present in Section 8.4. The only difference is that the iterative version uses a stack to keep track of the route back to the start node, while the recursive version uses the stack of recursive stack frames instead.

The Optimization Problem: An instance to the search problem is a graph with a source node s. To make it more interesting each node in the graph will also be allocated a weight by the instance. A solution for this instance is a path starting at the source node s. The value of the solution is the weight of the last node in the path. The goal is to find a path to a highest weighted node.

The Recursive Backtracking Algorithm: The initial recursive backtracking algorithm for this problem is just as it was before. The algorithm tries each of the edges out of the source node s. Trying an edge consists of: traversing the edge; recursively searching from there; and backtracking back across the edge. At any point in the algorithm, the stack of recursive stack frames traces (like bread crumbs) indicates the path followed from s to the current node being considered.

Pruning Equally Valued Solutions: Recall the example given in Section 15.1.2. If it could be proven that for every reptile there exists at least one mammal that is rated at least as high, then the algorithm can avoid iterating through all the reptiles. This is proved by modifying each reptile into an mammal of at least equal value by adding a wig.

In our present example, two paths (solutions) to the same node have the same value. Hence, all second paths to the same node can be pruned from our consideration. We will accomplish this as follows. When a node has been recursed from once, we will mark it and never recurse from it again.

The Recursive Depth First Search Problem:

Precondition: An input instance consists of a (directed or undirected) graph G with some of its nodes marked $found$ and a source node s.

Postcondition: The output is the same graph G except all nodes v reachable from s without passing through a previously found node are now also marked as being found.

Code: The graph, which is implement as a global variable, is assumed to be both input and output to the routine.

algorithm $DepthFirstSearch\,(s)$

$\langle$***pre & post−cond***$\rangle$**:** See above. This algorithm implicitly acts on some graph ADT G.

```
begin
    if s is marked as found then
        do nothing
    else
        mark s as found
        for each v connected to s
            DepthFirstSearch (v)
        end for
    end if
end algorithm
```

Exercise 15.3.1 *Trace out the iterative and the recursive algorithms on the same graph and see how they compare.*

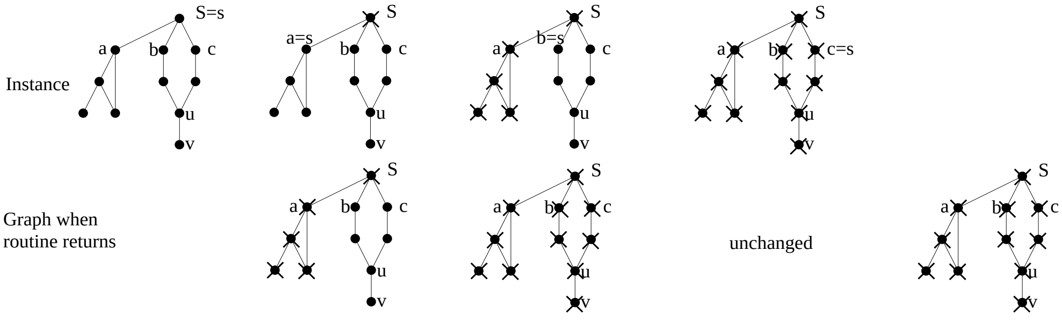


Figure 15.4: An Example Instance Graph

Example: Consider the instance graph in Figure 15.4.

Pruning Paths: There are two obvious paths from node S to node v. However, there are actually an infinite number of such paths. One path of interest is the one that starts at S, traverses around past u up to c and then down to v. All of these equally valued paths will be pruned from consideration, except the one that goes from S through b and u directly to v.

Three Friends: Given this instance, we first mark our source node S with an x and then we recurse three times, once from each of a, b, and c.

Friend *a*: Our first friend marks all nodes that are reachable from its source node $a = s$ without passing through a previously marked node. This includes only the nodes in the left most branch, because when we marked our source S, we block his route to the rest of the graph.

Friend *b*: Our second friend does the same. He finds, for example, the path that goes from b through u directly to v. He also finds and marks the nodes back around to c.

Friend *c*: Our third friend is of particular interest. He finds that his source node, c, has already been marked. Hence, he returns without doing anything. This prunes off this entire branch of the recursion tree. The reason that he can do this is because for any path to a node that he would consider, another path to the same node has already been considered.

Achieving The Postcondition: Consider the component of the graph reachable from our source s without passing through a previously marked nodes. (Because our instance has no marked nodes, this includes all the nodes.) To mark the nodes within this component, we do the following. First, we mark our source s. This partitions our component of reachable nodes into subcomponents that are still reachable from each other. Each such subcomponent has at least one edge from s into it. When we traverse the first such edge, this friend marks all the nodes within this subcomponent.

Size of a SubInstance: Section 15.1.4 stated that a reasonable measure of the size of an instance is the length of its longest valid solution. However, it also pointed out that this does not work when solutions might be infinitely long. This is the situation here, because solutions are paths that might wind infinitely often around cycles of the graph. Instead, we will measure the size of an instance to be the number of unmarked nodes. Each subinstance that we give to a friend is strictly "smaller" than our own instance because we marked our source node s.

Running Time: Marking a node before it is recursed from insures that each node is recursed from at most once. Recursing from a node involves traversing each edge from it. Hence, each edge is traversed at most twice: once from each direction. Hence, the running time is linear in the number of edges.

Chapter 16

Dynamic Programming

Dynamic Programming is a very powerful technique for designing polynomial time algorithms for optimization problems that are required every day to solve real life problems. Though it is not hard to give many examples, it has been a challenge to define exactly what the Dynamic Programming technique is. The previous chapter on recursive backtracking algorithms and this chapter attempt to do this. Dynamic programs can be thought of from two very different perspectives: as an iterative loop invariant algorithm that fills in a table and as an optimized recursive back tracking algorithm. It is important to understand both perspectives.

Optimization problems requires the algorithm to find the optimal solution from an exponentially large set of solutions for the given instance. (See Section 15.1.1.) The general technique of a dynamic program is as follows. Given an instance to the problem, the algorithm first determines its entire set of "subinstances", "sub-subinstances", "sub-sub-subinstances", and so on. Then it forms a table indexed by these subinstances. Within each entry of the table, the algorithm stores an optimal solution for the subinstance. These entries are filled in one at a time ordered from smallest to largest subinstance. When completed, the last entry will contain an optimal solution for the original instance.

From the perspective of an iterative algorithm, the subinstances are often prefixes of the instance and hence the algorithm iterates through the subinstances by iterating in some way through the elements of the instance. The loop invariant maintained is that the previous table entries have been filled in correctly. Progress is made while maintaining this loop invariant by filling in the next entry. This is accomplished using the solutions stored in the previous entries.

On the other hand, the technique for finding an optimal solution for a given subinstance is identical to the technique used in recursive backtracking algorithms, from Chapter 15.

16.1 The Iterative Loop Invariant Perspective of Dynamic Programming

Longest Increasing Contiguous SubSequence: To begin understanding dynamic programming from the iterative loop invariant perspective, let us consider a very simple example. Suppose that the input consists of a sequence $A[1,n]$ of integers and we want to find the longest contiguous subsequence $A[k_1,k_2]$ such that the elements are monotonically increasing. For example, the optimal solution for $[5,3,1,3,7,9,8]$ is $[1,3,7,9]$.

Deterministic Non-Finite Automata: The algorithm will read the input characters one at a time. Let $A[1,i]$ denote the subsequence read so far. The loop invariant will be that some information about this prefix $A[1,i]$ is stored. From this information about $A[1,i]$ and the element $A[i+1]$, the algorithm must be able to determine the required information about the prefix $A[1,i+1]$. In the end, the algorithm must be able to determine the solution, from this information about the entire sequence $A[1,n]$. Such an algorithm is called a Deterministic Finite Automata (DFA) if only a constant amount of information is stored at each point in time. However, in this chapter more memory then this will be required.

The Algorithm: After reading $A[1,i]$, remember the longest increasing contiguous subsequence $A[k_1,k_2]$ read so far and its size. In addition, so that you know whether the current increasing contiguous subsequence gets to be longer than the previous one, save the longest one ending in the value $A[i]$, and its size.

If you have this information about $A[1,i-1]$, then we learn it about $A[1,i]$ as follows. If $A[i-1] \leq A[i]$, then the longest increasing contiguous subsequence ending in the current value increases in length by one. Otherwise, it shrinks to being only the one element $A[i]$. If this subsequence increases to be longer than our previously longest, then it replaces the previous longest. In the end, we know the longest increasing contiguous subsequence.

The running time is $\Theta(n)$. As an aside, note that this is not a DFA because the amount of space to remember an index and a count is $\Theta(\log n)$.

Longest Increasing SubSequence: The following is a harder problem. Again the input consists of a sequence A of integers of size n. However now we want to find the longest (not necessarily contiguous) subsequence $S \subseteq [1,n]$ such that the elements, in the order that they appear in A, are monotonically increasing.

Dynamic Programming Deterministic Non-Finite Automata: Again the algorithm will read the input characters one at a time. But now the algorithm will store suitable information, not only about the current subsequence $A[1,i]$, but also about each previous subsequence $\forall j \leq i, A[1,j]$. Each of these subsequences $A[1,j]$ will be referred to as a *subinstance* of the original instance $A[1,n]$.

The Algorithm: As before, we will store both the longest increasing sequence seen so far and the longest one(s) that we are currently growing.

Suppose that the subsequence read so far is $10, 20, 1, 30, 40, 2, 50$. Then $10, 20, 30, 40, 50$ is the longest increasing subsequence so far. A shorter one is $1, 2, 50$. The problem with these ones is that they end in a large number, so we may not be able to extend them further. In case the rest of the string is $3, 4, 5, 6, 7, 8$, we will have to have remembered that $1, 2$ is the longest increasing subsequence that ends in the value 2. In fact, for many values v, we need to remember the longest increasing subsequence ending in this value v (or smaller), because in the end, it may be that the many of the remaining elements increase starting from this value. We only need to do this for values v that have been seen so far in the array. Hence, one possibility is to store, for each $j \leq i$, the longest increasing subsequence in $A[1,j]$ that ends with the value $A[j]$.

If you have this information for each $j \leq i-1$, then we learn it about $A[i]$ as follows. For each $j \leq i-1$, if $A[j] \leq A[i]$, then $A[i]$ can extend the subsequence ending with $A[j]$. Given this construction, the maximum length for i would then be one more than that for j. We get the longest one for i by taking the best over all such j. If there is no such j, then the count for i will be 1, namely simply $A[i]$ itself. The time for finding this best subsequence ending in $A[j]$ from which to extend to $A[i]$ would take $\Theta(i)$ time if each $j \in [1, i-1]$ needed to be checked. However, by storing this information in a heap, the best j can be found in $\Theta(logi)$ time. This gives a total time of $\Theta(\sum_{i=1}^{n} \log i) = \Theta(n \log n)$ for this algorithm.

In the end, the solution is the increasing subsequence ending in $A[j]$ where $j \in [1,n]$ is that for which this count is the largest.

Recursive Back Tracking: We can understand this same algorithm from the recursive back tracking perspective. See Chapter 15. Given the goal of finding the longest increasing subsequence of $A[1,i]$, one might ask the "little bird" whether or not the last element $A[i]$ should be included. Both options need to tried. If $A[i]$ is not to be included, then the remaining subtask is to find the longest increasing subsequence of $A[1,i-1]$. This is clearly a subinstance of the same problem. However, if $A[i]$ is to be included, then the remaining subtask is to find the longest increasing subsequence of $A[1,i-1]$ that ends in a value that is smaller or equal to this last value $A[i]$. This remotivates the fact that we need to learn both the longest increasing sequence of the prefix $A[1,i]$ and the longest one ends in $A[i]$.

16.2 The Recursive Back Tracking Perspective of Dynamic Programming

In addition to viewing dynamic programming algorithms as an iterative algorithm with a loop invariant, another useful technique for designing a dynamic programming algorithm for an optimization problem is to first design a recursive backtracking algorithm for the problem and then use memoization techniques to mechanically convert this algorithm into a dynamic programming algorithm.

A Recursive Back Tracking Algorithm: The first algorithm to design for the problem organizes the solutions for the given instance into a classification tree and then recursively traverses this tree looking for the best solution.

Memoization: Memoization speeds up such a recursive algorithm by saving the result for each subinstance and sub-subinstance in the recursion tree so that it does not need to be recomputed again if it is needed.

Dynamic programming: Dynamic programming takes the idea of memoization one step further. The algorithm first determines the set of subinstances within the entire tree of subinstances. Then it forms a table indexed by these subinstances. Within each entry of the table, the algorithm stores an optimal solution for the subinstance. These entries are filled out one at a time ordered from smallest to largest subinstance. In this way, when a friend needs an answer from one of his friends, this friend has already stored the answer in the table. When completed, the last entry will contain an optimal solution for the original instance.

16.2.1 Eliminating Repeated SubInstances With Memoization

Memoization is a technique to mechanically speed up a recursive algorithm. See Section 15.1.2. A recursive algorithm recurses on subinstances of a given input instance. Each of these *stack frames* recurses on sub-subinstances and so on, forming an entire tree of subinstances, called the *recursion tree*. Memoization speeds up such a recursive algorithm by saving the result for each subinstance considered in the recursion tree so that it does not need to be recomputed again if it is needed. This saves not only the work done by that stack frame, but also that done by the entire subtree of stack frames under it. Clearly, memoization speeds up a recursive algorithm if, and only if, the same subinstances are recalled many times.
A simple example is to compute the n^{th} Fibonacci number, where $Fib(0) = 0$, $Fib(1) = 1$, and $Fib(n) = Fib(n-1) + Fib(n-2)$. See Section 1.6.4.

The Recursive Algorithm: The obvious recursive algorithm is

algorithm $Fib(n)$

$\langle$***pre−cond***$\rangle$**:** n is a positive integer.

$\langle$***post−cond***$\rangle$**:** The output is the n Fibonacci number.

```
begin
    if( n = 0 or n = 1 ) then
        result( n )
    else
        result( Fib(n − 1) + Fib(n − 2) )
    end if
end algorithm
```

However, if you trace this recursive algorithm out, you will quickly see that that the running time is exponential, $T(n) = T(n-1) + T(n-2) + \Theta(1) = \Theta(1.61^n)$.

Saving Results: The recursive algorithm for Fib takes much more time than it should because many stack frames are called with the same input value. For example, $\langle 100\rangle$ calls $\langle 99\rangle$ and eventually calls $\langle 98\rangle$. (See Figure 16.1.) This $\langle 99\rangle$ stack frame calls $\langle 98\rangle$, which creates a large tree of stack frames under it. When the control gets back to the original $\langle 100\rangle$ stack frame, it calls $\langle 98\rangle$ and this entire tree is repeated.

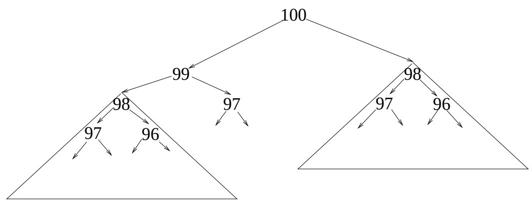


Figure 16.1: Stack frames called when starting with Fib(100)

Memoization avoids this problem by saving the results to previously computed instances. The following does not show how it is actually done, but it will give you an intuitive sense of what happens:

algorithm $Save(n, result)$

$\langle$***pre−cond***$\rangle$**:** n is an instance and $result$ is its solution.

$\langle$***post−cond***$\rangle$**:** This result is saved.

```
begin
    % Code not provided
end algorithm
```

algorithm $Get(n)$

$\langle$***pre−cond***$\rangle$**:** n is an instance.

$\langle$***post−cond***$\rangle$**:** if the result for n has been saved then returns true and result else returns false

```
begin
    % Code not provided
end algorithm
```

algorithm $Fib(n)$

$\langle$***pre−cond***$\rangle$**:** n is a positive integer.

$\langle$***post−cond***$\rangle$**:** The output is the n Fibonacci number.

```
begin
    ⟨saved, fib⟩ = Get(n)
    if( saved ) then
        result( fib )
    end if
    if( n = 0 or n = 1 ) then
        fib = n
    else
```

$fib = Fib(n-1) + Fib(n-2)$
end if
$Save(n, fib)$
result(fib)
end algorithm

The total computation time of this algorithm on instance n is only $\Theta(n)$. The reason is that the algorithm recurses at most once for each value $n' \in [0..n]$. Each such stack frame takes only $\Theta(1)$ time.

Save Only the Results of the SubInstances of the Given Instance: The goal is not to maintain a database of the results of all the instances of the problem the algorithm has ever seen. Such a database could be far too large. Instead, when given one instance that you want the result for, the program temporarily stores the results to all the subinstances that are encountered in the recursion tree for this instance.

Dynamic Programming: Dynamic programming takes the idea of memoization one step further. Instead of keeping track of which friends are waiting for answers from which friends, it is easier to first determine the complete set of subinstances for which solutions are needed and then to compute them in an order such that no friend must wait.

The set of subinstances in the recursive tree given by Fib on input n is $0, 1, \ldots, n$. The iterative algorithm sets up a one-dimensional table with entries indexed by $0, 1, \ldots, n$.

algorithm $Fib(n)$

$\langle$***pre−cond***$\rangle$**:** n is a positive integer.

$\langle$***post−cond***$\rangle$**:** The output is the n Fibonacci number.

begin
$table[0..n]\ fib$
$fib[0] = 0$
$fib[1] = 1$
loop $i = 2..n$
$fib[i] = fib[i-1] + fib[i-2]$
end loop
result($fib[n]$)
end algorithm

16.2.2 Redundancy in The Shortest Path within a Leveled Graph:

We will now continue with the shortest path within a leveled graph example started in Section 15.1.3.

The Redundancy: The recursive algorithm given traverses each of the exponentially many paths from s to t. The good news is that within this exponential amount of work, there is a great deal of redundancy. Different "friends" are assigned the exact same task. In fact, for each path from s to v_i, some friend is asked to solve the subinstance $\langle G, v_i, t\rangle$. See Figure 16.2.a.

The Memoization Algorithm: Let's improve the recursive algorithm as follows. We first observe that there are only $n = 10$ distinct subinstances the friends are solving. See Figure 16.2.b. Instead of having an exponential number of friends, we will have only ten friends each dedicated to one of the ten tasks, namely for $i \in [0..9]$, friend $Friend_i$ must find a best path from v_i to t.

Overhead for Recursion: This memoization algorithm still involves recursively traversing through a pruned version of the tree of subinstances. It requires the algorithm to keep track of which friends are waiting for answers from which friends. This is necessary if the algorithm does not know ahead of time which subinstances are needed. However, typically given an instance, the entire set of subinstances is easy to determine apriori.

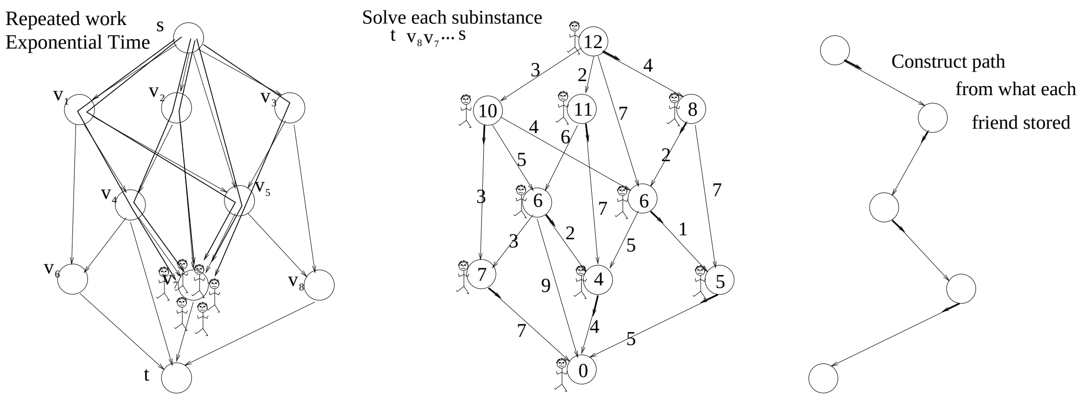


Figure 16.2: a) The recursive algorithm is exponential because different "friends" are assigned the same task. b) The dynamic programming algorithm: The value within the circle of node v_i gives the weight of a minimum path from v_i to t. The little arrow out of node v_i indicates the first edge a best path from v_i to t.

16.2.3 The Set of SubInstances

The first step in specifying, understanding, or designing a dynamic programming algorithm is to consider the set of subinstances that needs to be solved.

Obtaining The Set of SubInstances:

Recursive Structure: Before even designing the recursive backtracking algorithm for a problem, it is helpful to consider the recursive structure of the problem and try to guess what a reasonable set of subinstances for an instance might be.

Useful SubInstances: It is important that the subinstances are such that a solution to them is helpful to answering the original given instance.

Leveled Graph: In our running example, an instance to the problem asks for a shortest weighted path between specified nodes s and t within a given leveled graph G. If the nodes v_i and v_j both happen to be on route between s and t, then knowing a shortest path between these intermediate nodes would be helpful. However, we do not know apriori which nodes will be along the shortest path from s to t. Hence, we might let the set of subinstance consist of asking for a shortest path between v_i and v_j for every pair of nodes v_i and v_j.

Sometimes our initial guess either contains too many subinstances or too few. This is one of the reasons for designing a recursive backtracking algorithm.

The Set of Subinstances Called by the Recursive Back Tracking Algorithm: The set of subinstances needed by the dynamic program for a given instance is precisely the complete set of subinstances that will get called by the recursive backtracking algorithm, starting with this instance. (In Section 16.2.8, we discuss closure techniques for determining this set.)

Leveled Graph: Starting with the instance $\langle G, s, t\rangle$, the complete set of subinstances called will be $\{\langle G, v_i, t\rangle \mid \text{for each node } v_i \text{ above } t\}$, namely, the task of finding the best path from some new source node v_i to the original destination node t.

Base Cases and The Original Instance: The set of subinstances will contain subinstance with sophistication varying from base cases all the way to the original instance. Optimal solutions for the base case subinstances are trivial to find without any help from friends. When an optimal solution for the original instance is found, the algorithm is complete.

Leveled Graph: Because v_n is just another name for the original destination node t, the subinstance $\langle G, v_n, t\rangle$ within this set is in fact a trivial base case $\langle G, t, t\rangle$ asking for the best path from t to itself. On the other end, because v_0 is just another name for the original source node s, the subinstance $\langle G, v_0, t\rangle$ within this set is in fact the original instance $\langle G, s, t\rangle$.

The Number of Subinstances: A dynamic-programming algorithm is fast only if the given instance does not have many subinstances.

Sub-Instance is a Sub-Sequence: A common reason for the number subinstances of a given instance being polynomial is that the instance consists of a *sequence* of things rather than a *set* of things. In such a case, each subinstance can be a contiguous (continuous) subsequence of the things rather than an arbitrary subset of them. There are only $\mathcal{O}(n^2)$ contiguous subsequences of a sequence of length n, because one can be specified by specifying the two end points. Even better, there are even few subinstances if it is defined to be a prefix of the sequence. There are only n prefixes, because one can be specified by specifying the one end point. On the other hand, there are 2^n subsets of a set, because for each object you must decide whether or not to include it.

Leveled Graph: As stated in the definition of the problem, it is easiest to assume that the nodes are ordered such that an edge can only go from node v_i to node v_j if $i < j$ in this order. We initially guessed that the $\mathcal{O}(n^2)$ subinstances consisting of subsequences between two nodes v_i and v_j. We then decreased this to only the $\mathcal{O}(n)$ postfixes from some node v_i to the end t. Note that subinstances cannot be subsequences of the instance if the input graph is not required to be leveled.

Clique: In contrast, suppose that you want the largest clique S of a given graph G. A subinstance might specify a subset $V' \subseteq V$ of the nodes and ask for the largest clique S' in the subgraph G' on these nodes. The problem is that there are exponential number, 2^n, of such subinstances. Thus the obvious dynamic programming algorithm does not run in polynomial time. It is widely believed that this problem has no polynomial time algorithm exists.

16.2.4 Filling In The Table

Constructing a Table Indexed by Subinstances: Once you have determined the set of subinstances that needs to be considered, the next step is to construct the table. It must have one entry for each subinstance. Generally, the table will have one dimension for each "parameter" used to specify a particular subinstance. Each entry in the table is used to store an optimal solution for the subinstance along with its cost. Often we split this table into two tables: one for the solution and one for the cost.

Leveled Graph: The single parameter used to specify a particular subinstance is i. Hence, suitable tables would be $optSol[n..0]$ and $optCost[n..0]$, where $optSol[i]$ will store the best path from node v_i to node t and $optCost[i]$ will store its cost.

Solution from Sub-Solutions: The dynamic programming algorithm for finding an optimal solution to a given instance from an optimal solution to a subinstance is identical to that within the recursive backtracking algorithm.

Leveled Graph: $Friend_i$ find a best path from v_i to t as follows. For each of the edges $\langle v_i, v_k \rangle$ out of his node, he finds a best path from v_i to t from amongst those that take this edge. Then he takes the best of these best paths. A best path from v_i to t from amongst those that take the edge $\langle v_i, v_k \rangle$ is found by asking the $Friend_k$ for a best path from v_k and then tacking on the edge $\langle v_i, v_k \rangle$.

The Order in which to Fill the Table: Every recursive algorithm must guarantee that it recurses only on "smaller" instances. Hence, if the dynamic-programming algorithm fills in the table from smaller to larger instances, when an instance is being solved, the solution for each of its subinstances is already available. Alternatively, simply choose any order to fill the table that respects the dependencies between the instances and their subinstances.

Leveled Graph: The order, according to size, to fill the table is $t = v_n$, v_{n-1}, v_{n-2}, $\ldots$, v_2, v_1, $v_0 = s$. This order respects the dependencies on the answers because on instance $\langle G, v_i, t \rangle$,

$Friend_i$ depends on $Friend_k$ when ever there is an edge $\langle v_i, v_k \rangle$. Because edges must go from a higher level to a lower one, we know that $k > i$. This ensures that when $Friend_i$ does his work, $Friend_k$ has already stored his answer in the table.

16.2.5 Reversing the Order

Though this algorithm works well, it works backwards through the local structure of the solution. The Dynamic Programming technique reverses the recursive backtracking algorithm by completing the subinstances from smallest to largest. In order to have the final algorithm move forward, the recursive backtracking algorithm needs to go backwards. We will now start over and redevelop the algorithm so that the work is completed starting at the top of the graph.

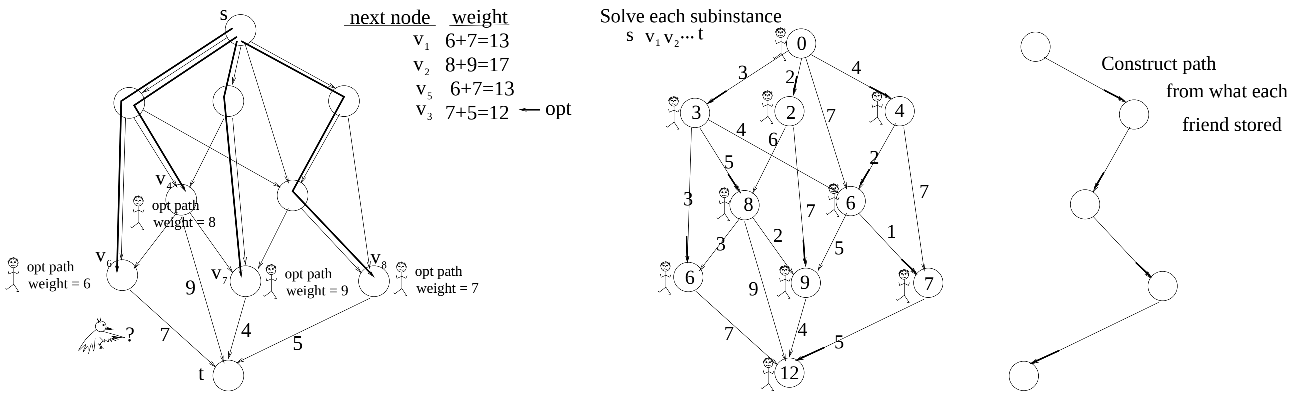


Figure 16.3: a) The reversed recursive backtracking algorithm. b) The forwarded moving dynamic programming algorithm: The value within the circle of node v_i gives the weight of a minimum path from s to v_i. The little arrow out of node v_i indicates the last edge a best path from s to v_i.

The Little Bird Asks About The Last Object: In the original algorithm, we asked the little bird which edge to take first in an optimal solution path from s to t. Instead, we now ask which edge is taken last within this optimal path.

The Recursive Back-Tracking Algorithm: Then we try all of the answers that the bird might give. When trying the possible last edge $<v_6, t>$, we construct the subinstance $\langle G, s, v_6 \rangle$ and in response our friend gives us the best path from s to node v_6. We add the edge $<v_6, t>$ to his solution to get the best path from s to t consistent with this little bird's answer. We repeat this process for the possible last edges, $<v_4, t>$, $<v_7, t>$, and $<v_8, t>$. Then we take the best of these best answers. See figure 16.3.a.

Determining the Set of Subinstances Called: Starting with the instance $\langle G, s, t \rangle$, the complete set of subinstances called will be $\{\langle G, s, v_i \rangle \mid \text{for each node } v_i \text{ below } s\}$, namely, the task of finding the best path from the original source node s to some new destination node v_i.

Base Cases and The Original Instance: The subinstance $\langle G, s, v_0 \rangle$ is a trivial base case $\langle G, s, s \rangle$ and the subinstance $\langle G, s, v_n \rangle$ is the original instance $\langle G, s, t \rangle$.

The Order in which to Fill the Table: A valid order to fill the table is s, v_1, v_2, v_3, $\ldots$, t. We have accomplished our goal of turning the algorithm around.

16.2.6 A Slow Dynamic Algorithm

The algorithm iterates filling in the entries of the table in the chosen order. The loop invariant when working on a particular subinstance is that all "smaller" subinstances that will be needed have been solved. Each iteration maintains the loop invariant while making progress by solving this next subinstance. This is done by completing the work of one stack frame of the recursive backtracking algorithm. However, instead of recursing or asking a friend to solve the subinstances, the dynamic programming algorithm simply looks up the subinstance's optimal solution and cost in the table.

Leveled Graph:

algorithm $LeveledGraph\,(G, s, t)$

$\langle \boldsymbol{pre-cond} \rangle$**:** G is a weighted directed layered graph and s and t are nodes.

$\langle \boldsymbol{post-cond} \rangle$**:** $optSol$ is a path with minimum total weight from s to t and $optCost$ is its weight.

```
begin
    table[0..n] optSol, optCost

    % Base Case.
    optSol[0] = ∅
    optCost[0] = 0
    % Loop over subinstances in the table.
    for i = 1 to n
        % Solve instance ⟨G, s, v_i⟩ and fill in table entry ⟨i⟩
        % Loop over possible bird answers.
        for each of the d edges ⟨v_k, v_i⟩
            % Get help from friend
            optSol_k = ⟨optSol[k], v_i⟩
            optCost_k = optCost[k] + w_⟨v_k,v_i⟩
        end for
        % Take the best bird answer.
        k_min = "a k that minimizes cost_k"
        optSol[i] = optSol_{k_min}
        optCost[i] = optCost_{k_min}
    end for
    return ⟨optSol[n], optCost[n]⟩
end algorithm
```

16.2.7 Decreasing the Time and Space

We will now consider a technique for decreasing time and space used by the dynamic-programming algorithm developed above.

Recap of a Dynamical Programming Algorithm: A dynamic programming algorithm has two nested loops (or sets of loops). The first iterates through all the subinstances represented in the table finding an optimal solution for each. When finding an optimal solution for the current subinstance, the second loop iterates through the K possible answers to the little bird's question, trying each of them. Within this inner loop, the algorithm must find a best solution for the current subinstance from amongst those consistent with the current bird's answer. This step seems to require only a constant amount of work. It involves looking up in the table an optimal solution for a sub-subinstance of the current subinstance and using this to construct a solution for the current subinstance.

Running Time?: From the recap of the algorithm, the running time is clearly the number of subinstances in the table times the number K of answers to the bird's question times what appears (falsely) to be constant time.

Leveled Graph: The running time of this algorithm is now polynomial. There are only $n = 10$ friends. Each constructs one path for each edge going into his destination node v_i. There are at most d of these. Constructing one of these paths appears to be a constant amount of work. He only asks a friend for a path and then tacks the edge being tried onto its end. This would give that the over all running time is only $\mathcal{O}(n \cdot d \cdot 1)$. However, it is more than this.

Friend to Friend Information Transfer: In both a recursive backtracking and a dynamic programming algorithm, information is transfered from sub-friend to friend. In a recursive backtracking, this information is transfered by returning it from a subroutine call. In a dynamic programming algorithm, this information is transfered by having the sub-friend store the information in the table entry associated with his subinstance and having the friend looking this information up from the table. The information transfered is an optimal solution and its cost. The cost, being only an integer, is not a big deal. However, an optimal solution generally requires $\Theta(n)$ characters to write down. Hence, transferring this information requires this much time.

Leveled Graph: The executing line of code is "$optSol_k = \langle optSol[k], v_i \rangle$." The optimal solution being transfered consists of a path. $Friend_i$ asks $Friend_k$ for his best path. This path may contain n nodes. Hence, it could take $Friend_i$ $\mathcal{O}(n)$ time steps simply to transfer the answer from $Friend_k$.

Time and Space Bottleneck: Being within these two nested loops, this information transfer is the bottleneck on the running time of the algorithm. In a dynamic programming algorithm, this information for each subinstance is stored in the table for the duration of the algorithm. Hence, this bottleneck on the memory space requirements of the algorithm.

Leveled Graph: The total time is $\mathcal{O}(n \cdot d \cdot n)$. The total space is $\Theta(n \cdot n)$ space, $\mathcal{O}(n)$ for each of the n table entries. These can be improved to $\Theta(nd)$ time and $\Theta(n)$ space.

A Faster Dynamic Programming Algorithm: We will now modify the dynamic programming algorithm to decrease its time and space requirements. The key idea is to reduce the amount of information transfered.

Cost from Sub-Cost: The sub-friends do not need to provide an optimal sub-solution in order to find the cost of an optimal solution to the current subinstance. The sub-friends need only provide the cost of this optimal sub-solution. Transferring only the costs, speeds up the algorithm.

Leveled Graph: In order for $Friend_i$ to find the *cost* of a best path from s to v_i, he need receive only the best *sub-cost* from his friends. (See the numbers within the circles in Figure 16.3.b.) For each of the edges $\langle v_k, v_i \rangle$ into his destination node v_i, he learns from $Friend_k$ the cost of a best path from s to v_k. He adds the cost of the edge $\langle v_k, v_i \rangle$ to this to determine the cost of a best path from s to v_i from amongst those that take this edge. Then he determines the cost of an overall best path from s to v_i by taking the best of these best costs.

Note that this algorithm requires $\mathcal{O}(n \cdot d)$ not $\mathcal{O}(n \cdot d \cdot n)$ time because this best cost can be transfered from $Friend_k$ to $Friend_i$ in constant time. However, this algorithm finds the cost of the best path, but does not find a best path.

The Little Bird's Advice:

Definition of Advice: A friend trying to find an optimal solution to his subinstance asks the little bird a question about this optimal solution. The answer, usually denoted k, to this question classifies the solutions. If this friend had an all powerful little bird, than she could advice him which class of solutions to search in to find an optimal solution. Given that he does not have such a bird, he must simply try all K of the possible answers and determine himself which answer is best. Either way we will refer to this best answer as *the little bird's advice.*

Leveled Graph: The birds advice to a $Friend_i$ who is trying to find a best path from s to v_i is which edge is taken last within this optimal path before reaching v_i. This edge for each friend is indicated by the little arrows in Figure 16.3.b.)

Advise from Cost: Consider again the algorithm that transfers only the cost of an optimal solution. Within this algorithm, each friend is able to determines the little bird's advice to him.

Leveled Graph: $Friend_i$ determines for each of the edges $\langle v_k, v_i\rangle$ into his node the cost of a best path from s to v_i from amongst those that take this edge and then determines which of these is best. Hence, though this $Friend_i$ never learns a best path from s to v_i in its entirety, he does learn which edge is taken last. In other words, he does determine, even without help from the little bird, what the little bird's advice would be.

Transferring the Bird's Advice: The little bird's advice does not need to be transfered from $Friend_k$ to $Friend_i$ because $Friend_i$ does not need it. However, $Friend_k$ will store this advice in the table so that it can be used at the end of the algorithm. This advice can usually be stored in constant space. Hence, it can be stored along with the best cost in constant time without slowing down the algorithm.

Leveled Graph: The advice indicates a single edge. Theoretically, taking $\mathcal{O}(\log n)$ bits, this takes more than constant space, however, practically it can be stored using two integers.

Information Stored in Table: The key change in order to make the dynamic programming algorithm faster is that the information stored in the table will no longer be an optimal solution and its cost. Instead, only the cost of an optimal solution and the little bird's advice k are stored.

Leveled Graph: The above code for *LeveledGraph* remains unchanged except for two changes. The first change is that the line "$optSol_k = \langle optSol[k], v_i\rangle$" within the inter loop is deleted, because the solution is no longer constructed. Because this line is the bottleneck, removing it speeds up the running time. The second change is that the line "$optSol[i] = optSol_{k_{min}}$" within the outer loop, which stores the optimal solution, is replaced with the line "$birdAdvice[i] = k_{min}$". This new line stores the bird's advice.

Time and Space Requirements: The running time of the algorithm computing the costs and the bird's advice is:

$$\begin{aligned}\text{Time} = &\ \text{“the number of subinstances indexing your table”} \\ &\times \text{“the number of different answers } K \text{ to the bird's question”}\end{aligned}$$

The space requirement is:

$$\text{Space} = \text{“the number of subinstances indexing your table”}$$

Constructing an Optimal Solution: With the above modifications, the algorithm no longer constructs an optimal solution. (Note an optimal solution for the original instance is required, but not for the other subinstance.) We construct an optimal solution for the instance using a separate algorithm that is run after the above faster dynamic programming algorithm fills the table in with costs and bird's advice. This new algorithm starts over from the beginning, solving the optimization problem. However, now we know what answer the little bird would give for every subinstance considered. Hence we can simply run the bird-friend algorithm.

A Recursive Bird-Friend Algorithm: The second run of the algorithm will be identical to the recursive algorithm, except now we only need to follow one path down the recursion tree. Each stack frame, instead of branching for each of the K answers that the bird might give, recurses only on the single answer given by the bird. This algorithm runs very quickly. Its running time is proportional to the number of fields needed to represent the optimal solution.

Leveled Graph: A best path from $s = v_0$ to $t = v_n$ is found as follows. Each friend knows which edge is the last edge taken to get to his node. What remains is to put these pieces together by walking backwards through the graph following the indicated directions. See the right side of Figure 16.3.b. $Friend_t$ knows that the last edge is $\langle v_8, t\rangle$. $Friend_8$ knows that the previous is $\langle v_5, v_8\rangle$. $Friend_5$ knows edge $\langle v_3, v_5\rangle$. Finally $Friend_3$ knows edge $\langle s, v_3\rangle$. This completes the path.

algorithm $LeveledGraphWithAdvice\left(\langle G, s, v_i\rangle, birdAdvice\right)$
$\langle$***pre & post−cond***$\rangle$**:** Same as *LeveledGraph* except with advice.

begin
    if($s = v_i$) then return($\emptyset$)
    $k_{min} = birdAdvice[i]$
    $optSubSol = LeveledGraphWithAdvice\left(\langle G, s, v_{k_{min}}\rangle,\ birdAdvice\right)$
    $optSol = \langle optSubSol, v_i\rangle$
    return $optSol$
end algorithm

Greedy/Iterative Solution: Because this last algorithm only recurses once per stack frame, it is easy to turn it into an iterative algorithm. The algorithm is a lot like a greedy algorithm found in Chapter 10 because the algorithm always knows which greedy choice to make. I leave this, however, for you to do as an exercise.

Exercise 16.2.1 *Design This iterative algorithm.*

16.2.8 Using Closure to Determine the Complete Set of Subinstances Called

When using the memoization technique to mechanically convert a recursive algorithm into an iterative algorithm, the most difficult step is determining for each input instance the complete set of subinstances that will get called by the recursive algorithm, starting with this instance.

Can Be Difficult: Sometimes determining this set of subinstances can be difficult.

Leveled Graph: When considering the recursive structure of the Leveled Graph problem we speculated that a subinstance might specify two nodes v_i to node v_j and require that a best path between them is found. By tracing the recursive algorithm, we see that these subinstances are not all needed, because the subinstances called always search for a path starting from the same fixed source node s. Hence, it is sufficient to consider only the subinstance of finding a best path from this s to node v_i for some node v_i. How do we know that only these subinstances are required?

A Set Being Closed under an Operation: The following mathematical concepts will help you. We say that the set of even integers is *closed under* addition and multiplication because the sum and the product of any two even numbers is even. In general, we say a set is closed under an operation if applying the operation to any elements in the set results in an element that is also in the set.

The Construction Game: Consider the following game: I give you the integer 2. You are allowed to construct new objects by taking objects you already have and either adding them or multiplying them. What is the complete set of number that you are able to construct?

Guess a Set: You might guess that you are able to construct the set of even integers. How do you know that this set is big enough and not too big?

Big Enough: Because the set of positive even integers is closed under addition and multiplication, we claim you will never construct an object that is not an even number.

Proof: We prove by induction on $t \geq 0$ that after t steps you only have even numbers. This is true for $t = 0$, because initially you only have the even integer 2. If it is true for t, the object constructed in step $t + 1$ is either the sum or the product of previously constructed objects, which are all even integers. Because the set of even integers is closed under these operations, the resulting object must also be even. This completes the inductive step.

Not Too Big: Every positive even integer can be generated by this game.

Proof: Consider some even number $i = 2j$. Initially, we have only 2. We construct i by adding $2 + 2 + 2 + \cdots + 2$ a total of j times.

Conclusion: The set of positive even integers accurately characterizes which numbers can be generated by this game, no less and no more.

Lemma: The set $\mathcal{S}$ will be the complete set of subinstances called starting from our initial instance I_{start} iff

1. $I_{start} \in \mathcal{S}$.
2. $\mathcal{S}$ is closed under the "sub"-operator. ($\mathcal{S}$ is big enough.)

   The sub-operator is defined as follows: Given a particular instance to the problem, applying the sub-operator produces all the subinstances constructed from it by a single stack frame of the recursive algorithm.
3. Every subinstance $I \in S$ can be generated from I_{start} using the sub-operator. ($\mathcal{S}$ is not too big.)

   This ensures that there are not any instances in $\mathcal{S}$ that are not needed. The dynamic-programming algorithm will work fine if your set of subinstances contains subinstances that are not called. However, you do not want the set too much larger than necessary, because the running time depends on its size.

Guess and Check: First try to trace out the recursive algorithm on a small example and guess what the set of subinstances will be. Then check it using the lemma.

Leveled Graph:

Guess a Set: The guessed set is $\{\langle G, s, v_i\rangle \mid \text{for each node } v_i \text{ below } s\}$.

Closed: Consider an arbitrary subinstance $\langle G, s, v_i\rangle$ from this set. The sub-operator considers some edge $\langle v_k, v_i\rangle$ and forms the subinstance $\langle G, s, v_k\rangle$. This is contained in the stated set of subinstances.

Generating: Consider an arbitrary subinstance $\langle G, s, v_i\rangle$. It will be called by the recursive algorithm if and only if the original destination node t can be reached from v_i. Suppose there is a path $\langle v_i, v_{k_1}, v_{k_2}, v_{k_3}, \ldots, v_{k_r}, t\rangle$. Then we demonstrate that the instance $\langle G, s, v_i\rangle$ is called by the recursive algorithm as follows: The initial stack frame on instance $\langle G, s, t\rangle$, among other things, recurses on $\langle G, s, v_{k_r}\rangle$, which recurses on $\left\langle G, s, v_{k_{r-1}}\right\rangle$, ..., which recurses on $\langle G, s, v_{k_1}\rangle$, which recurses on $\langle G, s, v_i\rangle$.

If the destination node t cannot be reached from node v_i, then the subinstance $\langle G, s, v_i\rangle$ will never be called by the recursive algorithm. Despite this, we will include it in our dynamic program, because this is not known about v_i until after the algorithm has run.

The Choose Example: The following is another example of determining the complete set of subinstance called by a recursive algorithm.

The Recursive Algorithm:

algorithm $Choose\,(m, n)$

$\langle \boldsymbol{pre-cond}\rangle$**:** m and n are positive integers.

$\langle \boldsymbol{post-cond}\rangle$**:** The output is $Choose(m, n) = \binom{m}{n} = \frac{m!}{n!(m-n)!}$.

```
begin
    if(n = 0 or n = m) then return 1
    else return Choose(m − 1, n) + Choose(m − 1, n − 1)
end algorithm
```

It happens that this evaluates to $Choose(m, n) = \binom{m}{n} = \frac{m!}{n!(m-n)!}$. Its running time is exponential.

Draw a Picture: The best way to visualize the set of subinstances $\langle i, j\rangle$ called starting from $\langle m, n\rangle = \langle 9, 4\rangle$ is to consider Figure 16.4. Such a figure will also help when determining the order to fill in the table by indicating the dependencies between the instances.

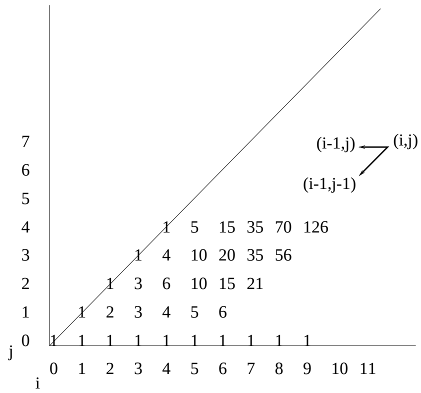


Figure 16.4: The table produced by the iterative Choose algorithm

16.3 Examples of Dynamic Programs

This completes the presentation of the general techniques and the theory behind dynamic programming algorithms. We will now develop algorithms for other optimization problems.

16.3.1 Printing Neatly Problem

Consider the problem of printing a paragraph neatly on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. Each printer line can hold a maximum of M characters. Our criterion for "neatness" is for there to be as few spaces on the ends of the lines as possible.

Printing Neatly:

Instances: An instance $\langle M; l_1, \ldots, l_n \rangle$ consists of the line and the word lengths. Generally, M will be thought of as a constant, so we will leave it out when it is clear from the context.

Solutions: A solution for an instances is a list giving the number of words for each line, $\langle k_1, \ldots, k_r \rangle$.

Cost of Solution: Given the number of words in each line, the cost of this solution is the sum of the cubes of the number of blanks on the end of each line, (including for now the last line).

Goal: Given the line and word lengths, the goal is to split the text into lines in a way that minimizes the cost.

Example: Suppose that one way of breaking the text into lines gives 10 blanks on the end of one of the lines, while another way gives 5 blanks on the end of one line and 5 on the end of another. Our sense of esthetics dictates that the second way is "neater". Our cost heavily penalizes having a large number of blanks on a single line by cubing the number. The cost of the first solution is $10^3 = 1,000$ while the cost of the second is only $5^3 + 5^3 = 250$.

Local vs Global Considerations: We are tempted to follow a greedy algorithm and put as many words on the first line as possible. However, a local *sacrifice* of putting few words on this line may lead globally to a better overall solution.

The Question For the Little Bird: We will ask for the number of words k to put on the *last* line. For each of the possible answers, the best solution is found that is consistent with this answer and then the best of these best solutions is returned.

Your Instance Reduced to a Subinstance: If the bird told you that an optimal number of words to put on the last line is k, then an optimal printing of the words is an optimal printing of first $n-k$ words followed by the remaining k words on a line by themselves. Your friend can find an optimal way of printing these first words by solving the subinstance $\langle M; l_1, \ldots, l_{n-k} \rangle$. All you need to do then is to add the last k words, namely $optSol = \left\langle optSol^{sub}, k \right\rangle$.

The Cost: The total cost of an optimal solution for the given instance $\langle M; l_1, \ldots, l_n \rangle$ is the cost of the optimal solution for the subinstance $\langle M; l_1, \ldots, l_{n-k} \rangle$, plus the cube of the number of blanks on the end of the line that contains the last k words, namely $optCost = optSubCost + (M - k + 1 - \sum_{j=n-k+1}^{n} l_j)^3$.

The Set of Subinstances: By tracing the recursive algorithm, we see that the set of subinstance used consists only of prefixes of the words, namely $\{\langle M; l_1, \ldots, l_i \rangle \mid i \in [0, n]\}$.

Closed: We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub-operator. Consider an arbitrary subinstance $\langle M; l_1, \ldots, l_i \rangle$ from this set. Applying the sub-operator constructs the subinstances $\langle M; l_1, \ldots, l_{i-k} \rangle$ for $1 \leq k \leq i$, which are contained in the stated set of subinstances.

Generating: Consider the arbitrary subinstance $\langle M; l_1, \ldots, l_i \rangle$. We demonstrate that it is called by the recursive algorithm as follows: The initial stack frame on instance $\langle M; l_1, \ldots, l_n \rangle$, among other things, sets k to be 1 and recurses on $\langle M; l_1, \ldots, l_{n-1} \rangle$. This stack frame also sets k to be 1 and recurses on $\langle M; l_1, \ldots, l_{n-2} \rangle$. This continues $n - i$ times, until the desired $\langle M; l_1, \ldots, l_i \rangle$ is called.

Constructing a Table Indexed by Subinstances: We now construct a table having one entry for each subinstance. The single parameter used to specify a particular subinstance is i. Hence, suitable tables would be $birdAdvice[0..n]$ and $cost[0..n]$.

The Order in which to Fill the Table: The "size" of subinstance $\langle M; l_1, \ldots, l_i \rangle$ is simply the number of words i. Hence, the table is filled in by looping with i from 0 to n.

Code:

algorithm $PrintingNeatly\,(\langle M; l_1, \ldots, l_n \rangle)$

$\langle$***pre−cond***$\rangle$: $\langle l_1, \ldots, l_n \rangle$ are the lengths of the words and M is the length of each line.

$\langle$***post−cond***$\rangle$: $optSol$ splits the text into lines in an optimal way and $optCost$ is its cost.

```
begin
    table[0..n] birdAdvice, cost
    % Base Case.
    birdAdvice[0] = ∅
    cost[0] = 0
    % Loop over subinstances in the table.
    for i = 1 to n
        % Solve instance ⟨M; l_1, ..., l_i⟩ and fill in table entry ⟨i⟩
        K = "maximum number k such that the words of length l_{i-k+1}, ..., l_i fit on a single line."
        % Loop over possible bird answers.
        for k = 1 to K
            % Get help from friend
            cost_k = cost[i − k] + (M − k + 1 − Σ_{j=i-k+1}^{i} l_j)^3
        end for
        % Take the best bird answer.
        k_min = "a k that minimizes cost_k"
        birdAdvice[i] = k_min
        cost[i] = cost_{k_min}
    end for
    optSol = PrintingNeatlyWithAdvice (⟨M; l_1, ..., l_n⟩, birdAdvice)
    return ⟨optSol, cost[n]⟩
end algorithm
```

Constructing an Optimal Solution: The friend with advice algorithm is the same as the above bird and friend algorithm, except the table and not the bird provides the information $k = 3$.

algorithm $PrintingNeatlyWithAdvice\left(\langle M; l_1, \ldots, l_i\rangle, birdAdvice\right)$
$\langle$***pre & post−cond***$\rangle$**:** Same as *PrintingNeatly* except with advice.

```
begin
    if(i = 0) then
        optSol = ∅
        return optSol
    else
        k_min = birdAdvice[i]
        optSol^sub = PrintingNeatlyWithAdvice(⟨M; l_1, ..., l_{i−k_min}⟩, birdAdvice)
        optSol = ⟨optSol^sub, k_min⟩
        return optSol
    end if
end algorithm
```

Time and Space Requirements: The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. There are $\Theta(n)$ subinstances in the table and the number of possible bird answers is $\Theta(n)$ because she has the option of telling you pretty well any number of words to put on the last line. Hence, The total running time is $\Theta(n)\cdot\Theta(n) = \Theta(n^2)$ and the space requirements are $\Theta(n)$.

Reusing the Table: Sometimes you can solve many related instances of the same problem using the same table.

The New Problem: When actually printing text neatly, it does not matter how many spaces are on the end of the very last line. Hence, the cube of this number should not be included in the cost.

Algorithm: For $k = 1, 2, 3, \ldots$, we find the best solution with k words on the last line and then take the best of these best. How to print all but the the last k words is an instance of the original Printing Neatly Problem because we charge for all of the remaining lines of text. Each of these takes $\mathcal{O}(n^2)$ time so the total time will be $\mathcal{O}(n \cdot n^2)$.

Reusing the Table: Time can be saved by filling in the table only once. One can get the costs for these different instances off this single table. After determining which is best, call *PrintingNeatlyWithAdvice* once to construct the solution for this instance. The total time is reduced to only $\mathcal{O}(n^2)$.

Exercise 16.3.1 *(See solution in Section 20) Trace out both the PrintingNeatly and the PrintingNeatlyWithAdvice routines on the text "Love life man while there as we be" on a card that is only 11 characters wide. (Yes, I choose the word lengths before I came up with the words.)*

16.3.2 Longest Common Subsequence

With so much money in things like genetics, there is a big demand for algorithms that find patterns in strings. The following optimization problem is called the *longest common subsequence*.

Longest Common Subsequence:

Instances: An instance consists of two sequences: $X = \langle A, B, C, B, D, A, B\rangle$ and $Y = \langle B, D, C, A, B, A\rangle$.

Solutions: A subsequence of a sequence is a subset of the elements taken in the same order. For example, $Z = \langle B, C, A\rangle$ is a subsequence of $X = \langle A, \underline{B}, \underline{C}, B, D, \underline{A}, B\rangle$. A solution is a subsequence Z that is common to both X and Y. For example, $Z = \langle B, C, A\rangle$ is a solution because it is a subsequence common to both X and Y ($Y = \langle \underline{B}, D, \underline{C}, \underline{A}, B, A\rangle$).

Cost of Solution: The cost of a solution is the length of the common subsequence, e.g., $|Z| = 3$.

Goal: Given two sequences X and Y, the goal is to find the longest common subsequence. For the example given above, $Z = \langle B, C, B, A \rangle$ would be the longer common subsequence (LCS).

Possible Little Bird Answers: Typically, the question asked of the little bird is for some detail about the end of an optimal solution.

Case $x_n \neq z_l$: Suppose that the bird assures us that the last character of X is not the last character of at least one longest common subsequence Z of X and Y. We could simply ignore this last character of X. We could ask a friend to give us a longest common subsequence of $X' = \langle x_1, \ldots, x_{n-1} \rangle$ and Y and this would be a longest common subsequence of X and Y.

Case $y_m \neq z_l$: Similarly, if we are told that the last character of Y is not used, than we could ignore it.

Case $x_n = y_m = z_l$: Suppose that we observed that the last characters of X and Y are the same and the little bird tells us that this character is the last character of an optimal Z. We could simply ignore this last character of both X and Y. We could ask a friend to give us a longest common subsequence of $X' = \langle x_1, \ldots, x_{n-1} \rangle$ and $Y' = \langle y_1, \ldots, y_{m-1} \rangle$. A longest common subsequence of X and Y would be the same, except with the character $x_n = y_m$ tacked on to the end, i.e. $Z = Z'x_n$.

Case $z_l =?$: Even more extreme, suppose that the little bird goes as far as to tell us the last character of a longest common subsequence Z. We could then delete the last characters of X and Y up to and including the last occurrence of this character. A friend could give us a longest common subsequence of the remaining X and Y and then we could add on the known character to give us Z.

Case $x_n = z_l \neq y_m$: Suppose that the bird assures us that the last character of X is the last character of an optimal Z. This would tell us the last character of Z and hence the last case would apply.

The Question For the Little Bird: Above we gave a number of different answers that the little bird might give, each of which would help us find an optimal Z. We could add even more possible answers to the list. However, the larger the number K of possible answers is, the more work our algorithm will have to do. Hence, we want to narrow this list of possibilities down as far as possible. It turns out that it is always the case that at least one of the first three cases given is true. To narrow the possible answers even further, note that we know on our own whether or not $x_n = y_m$. If $x_n = y_m$, then we only need to consider the third case, i.e. as in a greedy algorithm, we know the answer even before asking the question. On the other hand, if $x_n \neq y_m$, then we only need to consider the first two cases. For each of these $K = 2$ possible answers, the best solution is found that is consistent with this answer and then the best of these best solutions is returned.

Exercise 16.3.2 *Prove that if $x_n = y_m$, then we only need to consider the third case and if $x_n \neq y_m$, then we only need to consider the first two cases.*

The Set of Subinstances: We guess that the set of subinstances of the instance $\langle \langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_m \rangle \rangle$ is $\{ \langle \langle x_1, \ldots, x_i \rangle, \langle y_1, \ldots, y_j \rangle \rangle \mid i \leq n, j \leq m \}$.

Closed: We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub-operator. Consider an arbitrary subinstance, $\langle \langle x_1, \ldots, x_i \rangle, \langle y_1, \ldots, y_j \rangle \rangle$. Applying the sub-operator constructs the subinstances $\langle \langle x_1, \ldots, x_{i-1} \rangle, \langle y_1, \ldots, y_{j-1} \rangle \rangle$, $\langle \langle x_1, \ldots, x_{i-1} \rangle, \langle y_1, \ldots, y_j \rangle \rangle$, and $\langle \langle x_1, \ldots, x_i \rangle, \langle y_1, \ldots, y_{j-1} \rangle \rangle$, which are all contained in the stated set of subinstances.

Generating: We know that the specified set of subinstances does not contain subinstances not called by the recursive program because we can construct any arbitrary subinstance from the set with the sub-operator. Consider an arbitrary subinstance $\langle \langle x_1, \ldots, x_i \rangle, \langle y_1, \ldots, y_j \rangle \rangle$. The recursive program on instance $\langle\langle x_1,\ldots,x_n\rangle,\langle y_1,\ldots,y_m\rangle\rangle$ can recurse repeatedly on the second option $n-i$ times and then repeatedly on the third option $m-j$ times. This results in the subinstance $\langle\langle x_1,\ldots,x_i\rangle,\langle y_1,\ldots,y_j\rangle\rangle$.

Constructing a Table Indexed by Subinstances: We now construct a table having one entry for each subinstance. It will have a dimension for each of the parameters i and j used to specify a particular subinstance. The tables will be $cost[0..n,0..m]$ and $birdAdvice[0..n,0..m]$.

Order in which to Fill the Table: The official order in which to fill the table with subinstances is from "smaller" to "larger". The "size" of the subinstance $\langle\langle x_1,\ldots,x_i\rangle,\langle y_1,\ldots,y_j\rangle\rangle$ is $i+j$. Thus, you would fill in the table along the diagonals. However, the obvious order of looping for $i=0$ to n and $j=0$ to m also respects the dependencies between the instances and thus could be used instead.

Code:

algorithm $LCS\,(\langle\langle x_1,\ldots,x_n\rangle,\langle y_1,\ldots,y_m\rangle\rangle)$

$\langle$***pre−cond***$\rangle$**:** An instance consists of two sequences

$\langle$***post−cond***$\rangle$**:** *optSol* is a longest common subsequence and *optCost* is its length.

```
begin
    table[0..n, 0..m] birdAdvice, cost
    % Base Cases.
    for i = 0 to n
        birdAdvice[0, i] = ∅
        cost[0, i] = 0
    end for
    for i = 0 to m
        birdAdvice[i, 0] = ∅
        cost[i, 0] = 0
    end for
    % Loop over subinstances in the table.
    for i = 1 to n
        for j = 1 to m
            % Fill in entry ⟨i, j⟩
            if x_i = y_j then
                birdAdvice[i, j] = 1
                cost[i, j] = cost[i − 1, j − 1] + 1
            else
                % Try possible bird answers.
                % cases k = 2, 3
                % Get help from friend
                    cost_2 = cost[i − 1, j]
                    cost_3 = cost[i, j − 1]
                % end cases
                % Take the best bird answer.
                k_max = "a k ∈ [2, 3] that maximizes cost_k"
                birdAdvice[i, j] = k_max
                cost[i, j] = cost_{k_max}
            end if
        end for
    end for
    optSol = LCSWithAdvice(⟨⟨x_1, ..., x_n⟩, ⟨y_1, ..., y_m⟩⟩, birdAdvice)
    return ⟨optSol, cost[n, m]⟩
end algorithm
```

Constructing an Optimal Solution:

algorithm $LCSWithAdvice\left(\langle\langle x_1,\ldots,x_i\rangle,\langle y_1,\ldots,y_j\rangle\rangle, birdAdvice\right)$

$\langle$***pre & post−cond***$\rangle$**:** Same as LCS except with advice.

```
begin
    if (i = 0 or j = 0) then
        optSol = ∅
        return optSol
    end if
    k_max = birdAdvice[i, j]
    if k_max = 1 then
        optSol^sub = LCSWithAdvice(⟨⟨x_1, ..., x_{i−1}⟩, ⟨y_1, ..., y_{j−1}⟩⟩, birdAdvice)
        optSol = ⟨optSol^sub, x_i⟩
    else if k_max = 2 then
        optSol^sub = LCSWithAdvice(⟨⟨x_1, ..., x_{i−1}⟩, ⟨y_1, ..., y_j⟩⟩, birdAdvice)
        optSol = optSol^sub
    else if k_max = 3 then
        optSol^sub = LCSWithAdvice(⟨⟨x_1, ..., x_i⟩, ⟨y_1, ..., y_{j−1}⟩⟩, birdAdvice)
        optSol = optSol^sub
    end if
    return optSol
end algorithm
```

Time and Space Requirements: The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. The number of subinstances is $\Theta(n^2)$ and the bird has K=three possible answers for you. Hence, the time and space requirements are both $\Theta(n^2)$.

Example:

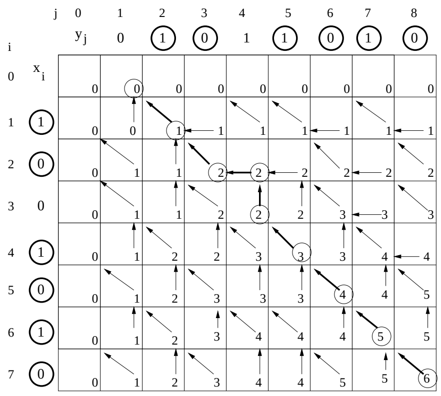


Figure 16.5: Consider the instance $X = 1001010$ and $Y = 01011010$. The tables generated are given. Each number is $cost[i, j]$, which is the length of the longest common subsequence of the first i characters of X and the first j characters of Y. The arrow indicates whether the bird's advice is to include $x_i = y_j$, to exclude x_i, or to exclude y_j. The circled digits of X and Y give an optimal solution.

16.3.3 A Greedy Dynamic Program: The Weighted Job/Activity Scheduling Problem

The Weighted Event Scheduling Problem: Suppose that many events want to use your conference room. Some of these events are given a higher priority than others. Your goal is to schedule the room in the optimal way. A greedy algorithm was given for the unweighted version in Section 10.2.1.

Instances: An instance is $\langle\langle s_1, f_1, w_1\rangle, \langle s_2, f_2, w_2\rangle, \ldots, \langle s_n, f_n, w_n\rangle\rangle$, where $0 \le s_i \le f_i$ are the starting and finishing times for n events and w_i are weights for each event.

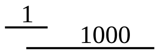
Solutions: A solution for an instance is a schedule S. This consists of a subset $S \subseteq [1..n]$ of the events that don't conflict by overlapping in time.

Cost of Solution: The cost $C(S)$ of a solution S is the sum of the weights of the events scheduled, i.e., $\sum_{i \in S} w_i$.

Goal: The goal of the algorithm is to find the *optimal solution*, i.e., the one that maximizes the total scheduled weight.

Failed Algorithms:

Greedy Earliest Finishing Time: The greedy algorithm used in Section 10.2.1 for the unweighted version greedily selects the event with the earliest finishing time f_i. This algorithm fails, when the events have weights. The following is a counter example.



The specified algorithm schedules the top event for a total weight of 1. The optimal schedule schedules the bottom event for a total weight of 1000.

Greedy Largest Weight: Another greedy algorithm selects the first event using the criteria of the largest weight w_i. The following is a counter example for this.

The top event has weight 2 and the bottom ones each have weight 1. The specified algorithm schedules the top event for a total weight of 2. The optimal schedule schedules the bottom events for a total weight of 9.

Unsorted Dynamic Programming: Now consider the dynamic programming algorithm in which the little bird tells you whether or not to schedule event J_n. Though this algorithm works, it has exponential running time. The following is an instance that has an exponential number of subinstance. The events in the instance are paired so that for $i \in [1..\frac{n}{2}]$, job J_i conflicts with job $J_{\frac{n}{2}+i}$, but jobs between pairs do not conflict. After the little bird tells you whether or not to schedule jobs $J_{\frac{n}{2}+i}$ for $i \in [1..\frac{n}{2}]$, job J_i will remain in the subinstance if and only if job $J_{\frac{n}{2}+i}$ was not scheduled. This results in at least $2^{n/2}$ different paths down the tree of stack frames in the recursive backtracking algorithm, each leading to a different subinstance.

The Greedy Dynamic Programming: First sort the events according to their finishing times f_i (a greedy thing to do). Then run a dynamic programming algorithm in which the little bird tells you whether or not to schedule event J_n.

Bird & Friend Algorithm: Consider an instance $J = \langle\langle s_1, f_1, w_1\rangle, \langle s_2, f_2, w_2\rangle, \ldots, \langle s_n, f_n, w_n\rangle\rangle$. The little bird considers an optimal schedule. We ask the little bird whether or not to schedule event J_n. If she says yes, then the remaining possible events to schedule are those in J excluding event J_n and excluding all events that conflict with event J_n. We ask a friend to schedule these. Our schedule is his with event J_n added. If instead the bird tells us not to schedule event J_n, then the remaining possible events to schedule are those in J excluding event J_n.

The Set of Subinstances: By tracing the recursive algorithm, we see that the set of subinstance used is $\{\langle\langle s_1, f_1, w_1\rangle, \langle s_2, f_2, w_2\rangle, \ldots, \langle s_i, f_i, w_i\rangle\rangle \mid i \in [0..n]\}$.

Closed: We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub-operator. Consider an arbitrary subinstance $\langle\langle s_1, f_1, w_1\rangle, \langle s_2, f_2, w_2\rangle, \ldots, \langle s_i, f_i, w_i\rangle\rangle$ in the set. If we delete from this event J_i and all events that conflict with it, we must show that this new subinstance is again in our set. Let $i' \in [0..i-1]$ be the largest index such that $f_{i'} \leq s_i$. Because the events have been sorted by their finishing time, we know that all events J_k in $\langle\langle s_1, f_1, w_1\rangle, \langle s_2, f_2, w_2\rangle, \ldots, \langle s_{i'}, f_{i'}, w_{i'}\rangle\rangle$ also have $f_k \leq s_i$ and hence do not conflict with J_i. All events J_k in $\langle\langle s_{i'+1}, f_{i'+1}, w_{i'+1}\rangle, \ldots, \langle s_i, f_i, w_i\rangle\rangle$ have $s_i < f_k \leq f_i$ and hence conflict with J_i. If follows that the resulting subinstance is $\langle\langle s_1, f_1, w_1\rangle, \langle s_2, f_2, w_2\rangle, \ldots, \langle s_{i'}, f_{i'}, w_{i'}\rangle\rangle$, which is our set of subinstances. If only the other hand, only event J_i is deleted, then the resulting subinstance is $\langle\langle s_1, f_1, w_1\rangle, \ldots, \langle s_{i-1}, f_{i-1}, w_{i-1}\rangle\rangle$, which is obviously in our set of subinstances.

Generating: Consider the arbitrary subinstance $\langle\langle s_1, f_1, w_1\rangle, \langle s_2, f_2, w_2\rangle, \ldots, \langle s_i, f_i, w_i\rangle\rangle$. It is generated by the recursive algorithm when the little bird states that none of the later events are included in the solution.

The Table: The dynamic programming table is a one dimensional array indexed by $i \in [0..n]$. The order to fill it in is with increasing i. As in the greedy algorithm, the events are being considered ordered by earliest finishing time first. The i entry is filled in by trying each of the two answers the bird might give.

Time and Space Requirements: The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. This gives $T = \Theta(n \times 2)$. The time of the entire algorithm is then dominated by the time to initially sort the events by their finishing time f_i.

Exercise 16.3.3 *Write out the pseudo code for this algorithm.*

16.3.4 Exponential Time? The Integer-Knapsack Problem

Another example of an optimization problem is the *integer-knapsack problem*. For the problem in general, no polynomial algorithm is known. However, if the volume of the knapsack is a small integer, then dynamic programming provides a fast algorithm.

Integer-Knapsack Problem:

Instances: An instance consists of $\langle V, \langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\rangle$. Here, V is the total volume of the knapsack. There are n objects in a store. The volume of the i^{th} object is v_i, and its price is p_i.

Solutions: A solution is a subset $S \subseteq [1..n]$ of the objects that fit into the knapsack, i.e., $\sum_{i \in S} v_i \leq V$.

Cost of Solution: The cost of a solution S is the total value of what is put in the knapsack, i.e., $\sum_{i \in S} p_i$.

Goal: Given a set of objects and the size of the knapsack, the goal is fill the knapsack with the greatest possible total price.

The Question to Ask the Little Bird: Consider a particular instance, $\langle V, \langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\rangle$ to the knapsack problem. The little bird might tell us whether or not an optimal solution for this instance contains the n^{th} item from the store. For each of these $K = 2$ possible answers, the best solution is found that is consistent with this answer and then the best of these best solutions is returned.

Reduced to Subinstance: Either way, our search for an optimal packing is simplified. If an optimal solution does *not* contain the n^{th} item, then we simply delete this last item from consideration. This leaves us with the smaller instance, $\langle V, \langle v_1, p_1\rangle, \ldots, \langle v_{n-1}, p_{n-1}\rangle\rangle$. On the other hand, if an optimal solution *does* contain the n^{th} item, then we can take this last item and put it into the knapsack first. This leaves a volume of $V - v_n$ in the knapsack. We determine how best to fill the rest of the knapsack with the remaining items by looking at the smaller instance $\langle V - v_n, \langle v_1, p_1\rangle, \ldots, \langle v_{n-1}, p_{n-1}\rangle\rangle$.

The Set of Subinstances: By tracing the recursive algorithm, we see that the set of subinstance used is $\{\langle V', \langle v_1, p_1\rangle, \ldots, \langle v_i, p_i\rangle\rangle \mid V' \in [0..V], i \in [0..n]\}$.

Closed: We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub-operator. Applying the sub-operator to an arbitrary subinstance $\langle V', \langle v_1, p_1\rangle, \ldots, \langle v_i, p_i\rangle\rangle$ from this set constructs subinstances $\langle V', \langle v_1, p_1\rangle, \ldots, \langle v_i, p_{i-1}\rangle\rangle$ and $\langle V' - v_i, \langle v_1, p_1\rangle, \ldots, \langle v_i, p_{i-1}\rangle\rangle$, which are contained in the stated set of subinstances.

Generating: For some instances, these subinstances might not get called in the recursive program for every possible value of V'. However, as an exercise you could construct instances for which each such subinstance was called.

Exercise 16.3.4 *Construct an instance for which each of its subinstances are called.*

Constructing a Table Indexed by Subinstances: The table indexed by the above set of subinstances will have a dimension for each of the parameters i and V' used to specify a particular subinstance. The tables will be $cost[0..n, 0..V]$ and $birdAdvice[0..V, 0..n]$.

Code:

algorithm $Knapsack\left(\langle V, \langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\rangle\right)$

$\langle$***pre−cond***$\rangle$**:** V is the volume of the knapsack. v_i and p_i are the volume and the price of the i^{th} objects in a store.

$\langle$***post−cond***$\rangle$**:** *optSol* is a way to fill the knapsack with the greatest possible total price. *optCost* is its price.

```
begin
    table[0..V, 0..n] birdAdvice, cost
    % Base cases are when # of items is zero
    loop V' = 0..V
        cost[V', 0] = 0
        birdAdvice[V', 0] = ∅
    end loop
    % Loop over subinstances in the table.
    loop i = 1 to n
        loop V' = 0 to V
            % Fill in entry ⟨V', i⟩
            % Try possible bird answers.
            % cases k = 1, 2 where 1=exclude 2=include
                % Get help from friend
                cost_1 = cost[V', i − 1]
                if(V' − v_i ≥ 0) then
                    cost_2 = cost[V' − v_i, i − 1] + p_i
                else
                    cost_2 = −∞
                end if
            % end cases
            % Take the best bird answer.
            k_max = "a k that minimizes cost_k"
            birdAdvice[V', i] = k_max
            cost[V', i] = cost_{k_max}
        end for
    end for
    optSol = KnapsackWithAdvice (⟨V, ⟨v_1, p_1⟩, ..., ⟨v_n, p_n⟩⟩, birdAdvice)
    return ⟨optSol, cost[V, n]⟩
end algorithm
```

Constructing an Optimal Solution:

algorithm $KnapsackWithAdvice\,(\langle V', \langle v_1, p_1\rangle, \ldots, \langle v_i, p_i\rangle\rangle, birdAdvice)$

$\langle$***pre & post−cond***$\rangle$: Same as $Knapsack$ except with advice.

```
begin
    if (i = 0) then
        optSol = ∅
        return optSol
    end if
    k_max = birdAdvice[V', i]
    if k_max = 1 then
        optSol^sub = KnapsackWithAdvice(⟨V', ⟨v_1, p_1⟩, ..., ⟨v_{i-1}, p_{i-1}⟩⟩, birdAdvice)
        optSol = optSol^sub
    else
        optSol^sub = KnapsackWithAdvice(⟨V' − v_i, ⟨v_1, p_1⟩, ..., ⟨v_{i-1}, p_{i-1}⟩⟩, birdAdvice)
        optSol =< optSol^sub ∪ i
    end if
    return optSol
end algorithm
```

Time and Space Requirements: The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. The number of subinstances is $\Theta(V \cdot n)$ and the bird chooses between two options: to include or not to include the object. Hence, the running and the space requirements are both $\Theta(V \cdot n)$.

You should express the running time of an algorithm as a function of input size. The number of bits needed to represent the instance $\langle V', \langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\rangle$ is $N = |V| + n \cdot (|v| + |p|)$, where (1) $|V|$ is the number of bits in integer V and (2) $|v|$ and $|p|$ are the maximum number of bits needed to represent the volumes and prices of the individual items. Expressed in these terms, the running time is $T(|\text{instance}|) = \Theta(nV) = \Theta(n2^{|V|})$. This is quicker than the brute-force algorithm because running time is polynomial in the number of items n. In the worst case, however, V is large and the time can be exponential in the number of bits N. I.e., if $|V| = \Theta(N)$, then $T = \Theta(2^N)$. In fact, the knapsack problem is one of the classic NP complete problems. NP completeness, which indicates that the problem is hard, will be covered briefly in this course and at more length in future courses.

Exercise 16.3.5 *The fractional-knapsack problem is the same as the integer-knapsack problem except that there is a given amount of each object and any fractional amount of each it can be put into the knapsack. Develop a quick greedy algorithm for this problem.*

16.3.5 The Solution Viewed as a Tree: Chains of Matrix Multiplications

The next example will be our first example in which the fields of information specifying a solution are organized into a tree instead of into a sequence. See Section 15.1.4. The algorithm asks the little bird to tell it the field at the root of one of the instance's optimal solutions and then a separate friend will be asked for each of the solution's subtrees.

The optimization problem determines how to optimally multiplying together a chain of matrices. Multiplying an $a_1 \times a_2$ matrix by a $a_2 \times a_3$ matrix requires $a_1 \cdot a_2 \cdot a_3$ scalar multiplications. Matrix multiplication is commutative, meaning that $(M_1 \cdot M_2) \cdot M_3 = M_1 \cdot (M_2 \cdot M_3)$. Sometimes different bracketing of a sequence of matrix multiplications can lead to the total number of scalar multiplications being very different. For example,

$$\big(\,\langle 5 \times 1,000\rangle \cdot \langle 1,000 \times 2\rangle\,\big) \cdot \langle 2 \times 2,000\rangle = \langle 5 \times 2\rangle \cdot \langle 2 \times 2,000\rangle = \langle 5 \times 2,000\rangle$$

requires $5 \times 1,000 \times 2 + 5 \times 2 \times 2,000 = 10,000 + 20,000 = 30,000$ scalar multiplications. However,

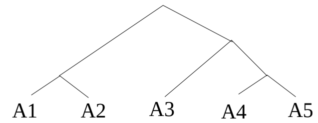
$$\langle 5 \times 1,000\rangle \cdot \big(\,\langle 1,000 \times 2\rangle \cdot \langle 2 \times 2,000\rangle\,\big) = \langle 5 \times 1,000\rangle \cdot \langle 1,000 \times 2,000\rangle = \langle 5 \times 2,000\rangle$$

requires $1,000 \times 2 \times 2,000 + 5 \times 1,000 \times 2,000 = 4,000,000 + 10,000,000 = 14,000,000$. The problem considered here is to find how to bracket a sequence of matrix multiplications in order to minimize the number of scalar multiplications.

Chains of Matrix Multiplications:

Instances: An instance is a sequence of n matrices $\langle A_1, A_2, \ldots, A_n \rangle$. (A precondition is that for each $k \in [1..n-1]$ $width(A_k) = height(A_{k+1})$.)

Solutions: A solution is a way of bracketing the matrices, e.g., $((A_1 A_2)(A_3(A_4 A_5)))$. A solution can equivalently be viewed as a binary tree with the matrices $A_1, \ldots, A_n$ at the leaves. The binary tree would give the order in which to multiply the matrices.



Cost of Solution: The cost of a solution is the number of scalar multiplications needed to multiply the matrices according to the bracketing.

Goal: Given a sequence of matrices, the goal is to find a bracketing that requires the fewest multiplications.

A Failed Greedy Algorithm: An obvious greedy algorithm selects where the last multiplication will occur according the criteria of which is cheapest. We can prove that any such simple greedy algorithm will fail, even when the instance contains only three matrices. Let the matrices A_1, A_2, and A_3 have height and width $\langle a_0, a_1 \rangle$, $\langle a_1, a_2 \rangle$, and $\langle a_2, a_3 \rangle$. There are two orders in which these can be multiplied. Their costs are as follows.

$cost((A_1 \cdot A_2) \cdot A_3) = a_0 a_1 a_2 + a_0 a_2 a_3$
$cost(A_1 \cdot (A_2 \cdot A_3)) = a_1 a_2 a_3 + a_0 a_1 a_3$

Consider the algorithm that uses the method whose last multiplication is the cheapest. Let us assume that the algorithm uses the first method. This gives that

a) $a_0 a_2 a_3 < a_0 a_1 a_3$

However, we want the algorithm to give the wrong answer. Hence, we want the second method to be the cheapest. This gives that

b) $a_0 a_1 a_2 + a_0 a_2 a_3 > a_1 a_2 a_3 + a_0 a_1 a_3$

Simplifying line (a) gives $a_2 < a_1$. Plugging line (a) into line (b) gives $a_0 a_1 a_2 >> a_1 a_2 a_3$. Simplifying this gives $a_0 >> a_3$. Let us now assign simple values meeting $a_2 < a_1$ and $a_0 >> a_3$. Say $a_0 = 1000$, $a_1 = 2$, $a_2 = 1$, and $a_3 = 1$. Plugging these in gives

$cost((A_1 \cdot A_2) \cdot A_3) = 1000 \cdot 2 \cdot 1 + 1000 \cdot 1 \cdot 1 = 2000 + 1000 = 3000$
$cost(A_1 \cdot (A_2 \cdot A_3)) = 2 \cdot 1 \cdot 1 + 1000 \cdot 2 \cdot 1 = 2 + 2000 = 2002$

This is an instance in which the algorithm gives the wrong answer. Because $1000 < 2000$ it uses the first method. However, the second method is cheaper.

Exercise 16.3.6 *Give the steps in the technique above to find a counter example for the greedy algorithm that multiplies the cheapest pair together first.*

A Failed Dynamic Programming Algorithm: An obvious question to ask the little bird would be which pair of consecutive matrices to multiply together first. Though this algorithm works, it has exponential running time. The problem is that if you trace the execution of the recursive algorithm, there is an exponential number of different subinstances. Consider paths down the tree of stack frames in which for each pair A_{2i} and A_{2i+1}, the bird either gets us to multiply them together or does not. This results in $2^{n/2}$ different paths down the tree of stack frames in the recursive backtracking algorithm, each leading to a different subinstance.

The Question to Ask the Little Bird: A better question is to ask the little bird to give us the splitting k so that the last multiplication multiplies the product of $\langle A_1, A_2, \ldots, A_k \rangle$ and of $\langle A_{k+1}, \ldots, A_n \rangle$. This is equivalent to asking for the root of the binary tree. For each of the possible answers, the best solution is found that is consistent with this answer and then the best of these best solutions is returned.

Reduced to SubInstance: With this advice, our search for an optimal bracketing is simplified. We need only solve two subinstances: finding an optimal bracketing of $\langle A_1, A_2, \ldots, A_k\rangle$ and of $\langle A_{k+1}, \ldots, A_n\rangle$.

Recursive Structure: An optimal bracketing of the matrices $\langle A_1, A_2, \ldots, A_n\rangle$ multiplies the sequence $\langle A_1, \ldots, A_k\rangle$ with its optimal bracketing, multiplies $\langle A_{k+1}, \ldots, A_n\rangle$ with its optimal bracketing, and then multiplies these two resulting matrices together, i.e., $optSol = \Big(\ optLeft\ \Big)\Big(\ optRight\ \Big)$.

The Cost of the Optimal Solution Derived from the Cost for Subinstances: The total number of scalar multiplications used in this optimal bracketing is the number used to multiply $\langle A_1, \ldots, A_k\rangle$, plus the number for $\langle A_{k+1}, \ldots, A_n\rangle$, plus the number to multiply the final two matrices. $\langle A_1, \ldots, A_k\rangle$ evaluates to a matrix whose height is the same as that of A_1 and whose width is that of A_k. Similarly, $\langle A_{k+1}, \ldots, A_n\rangle$ becomes a $height(A_{k+1}) \times width(A_n)$ matrix. (Note that $width(A_k) = height(A_{k+1})$.) Multiplying these requires $height(A_1) \times width(A_k) \times width(A_n)$ number of scalar multiplications. Hence, in total, $cost = costLeft + costRight + height(A_1) \times width(A_k) \times width(A_n)$.

The Set of Subinstances Called: The set of subinstances of the instance $\langle A_1, A_2, \ldots, A_n\rangle$ is $\langle A_i, A_{i+1}, \ldots, A_j\rangle$ for every choice of end points $1 \leq i \leq j \leq n$. This set of subinstances contains all the subinstances called, because it is closed under the sub-operator. Applying the sub-operator to an arbitrary subinstance $\langle A_i, A_{i+1}, \ldots, A_j\rangle$ from this set constructs subinstances $\langle A_i, \ldots, A_k\rangle$ and $\langle A_{k+1}, \ldots, A_j\rangle$ for $i \leq k < j$, which are contained in the stated set of subinstances. Similarly, the set does not contain subinstances *not* called by the recursive program, because we easily can construct any arbitrary subinstance in the set with the sub-operator. For example, $\langle A_1, \ldots, A_n\rangle$ sets $k = j$ and calls $\langle A_1, \ldots, A_j\rangle$, which sets $k = i + 1$ and calls $\langle A_i, \ldots, A_j\rangle$.

Constructing a Table Indexed by Subinstances: The table indexed by the above set of subinstances will have a dimension for each of the parameters i and j used to specify a particular subinstance. The tables will be $cost[1..n, 1..n]$ and $birdAdvice[1..n, 1..n]$. See Figure 16.6.

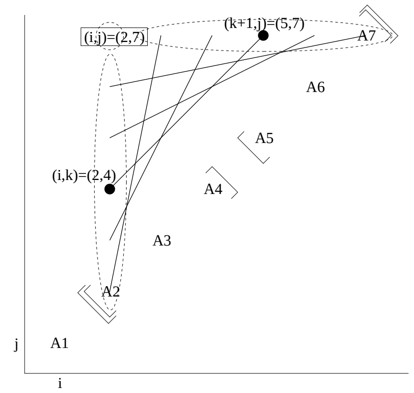


Figure 16.6: The table produced by the dynamic-programming solution for Choose. When searching for the optimal bracketing of $A_2, \ldots, A_7$, one of the methods to consider is $[A_2, \ldots, A_4][A_5, \ldots, A_7]$.

Order in which to Fill the Table: The size of a subinstance is the number of matrices in it. We will fill the table in this order.

Exercise 16.3.7 *(See solution in Section 20) Use a picture to make sure that when $\langle i, j\rangle$ is filled, $\langle i, k\rangle$ and $\langle k + 1, j\rangle$ are already filled for all $i \leq k < j$. Give two other orders that work.*

Code:

algorithm $MatrixMultiplication\left(\langle A_1, A_2, \ldots, A_n\rangle\right)$

$\langle$***pre–cond***$\rangle$**:** An instance is a sequence of n matrices.

$\langle$***post–cond***$\rangle$**:** *optSol* is a bracketing that requires the fewest multiplications and *optCost* is this number.

```
begin
    table[1..n, 1..n] birdAdvice, cost
    % Subinstances of size one.
    for i = 1 to n
        birdAdvice[i, i] = ∅
        cost[i, i] = 0
    end for
    % Loop over subinstances in the table.
    for size = 2 to n
        for i = 1 to n − size + 1
            j= i+size-1
            % Fill in entry ⟨i, j⟩
            % Loop over possible bird answers.
            for k = i to j − 1
                % Get help from friend
                cost_k = cost[i, k] + cost[k + 1, j] + height(A_i) × width(A_k) × width(A_j)
            end for
            % Take the best bird answer.
            k_min = "a k that minimizes cost_k"
            birdAdvice[i, j] = k_min
            cost[i, j] = cost_{k_min}
        end for
    end for
    optSol = MatrixMultiplicationWithAdvice (⟨A_1, A_2, ..., A_n⟩, birdAdvice)
    return ⟨optSol, cost[1, n]⟩
end algorithm
```

Constructing an Optimal Solution:

algorithm $MatrixMultiplicationWithAdvice\left(\langle A_i, A_2, \ldots, A_j\rangle, birdAdvice\right)$

$\langle$***pre & post–cond***$\rangle$**:** Same as $MatrixMultiplication$ except with advice.

```
begin
    if (i = j) then
        optSol = ∅
        return optSol
    end if
    k_min = birdAdvice[i, j]
    optLeft = MatrixMultiplicationWithAdvice (⟨A_1, ..., A_{k_min}⟩, birdAdvice)
    optRight = MatrixMultiplicationWithAdvice (⟨A_{k_min+1}, ..., A_j⟩, birdAdvice)
    optSol = ( optLeft ) ( optRight )
    return optSol
end algorithm
```

Time and Space Requirements: The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. The number of subinstances is $\Theta(n^2)$ and the bird chooses one of $\Theta(n)$ places to split the sequence of matrices. Hence, the running time is $\Theta(n^3)$ and the space requirements are $\Theta(n^2)$.

16.3.6 Generalizing the Problem Solved: Best AVL Tree

As discussed in Section 11.1.3, it is sometimes useful to generalizing the problem solved so that you can either give or receive more information from your friend in a recursive algorithm. This was demonstrated in Section 12 with a recursive algorithm for determining whether or not a tree is an AVL tree. This same idea is useful for dynamic programming. We will now demonstrate this by giving an algorithm for finding the best AVL tree. To begin, we will develop the algorithm for the *Best Binary Search Tree* Problem introduced at the beginning of this chapter.

The Best Binary Search Tree Problem:

Instances: An instance consists of n probabilities $p_1, \ldots, p_n$ to be associated with the n keys $a_1 < a_2 < \ldots < a_n$. The values of the keys themselves do not matter. Hence, can assume that $a_i = i$.

Solutions: A solution for an instance is a binary search tree containing the keys. A binary search tree is a binary tree such that the nodes are labeled with the keys and for each node all the keys in its left subtree are smaller and all those in the right are larger.

Cost of Solution: The cost of a solution is the expected depth of a key when choosing a key according to the given probabilities, namely $\sum_{i\in[1..n]} [p_i \cdot$ depth of a_i in tree$]$.

Goal: Given the keys and the probabilities, the goal is to find a binary search tree with minimum expected depth.

Expected Depth: The time required to searching for a key is proportional to the depth of the key in the binary search tree. Finding the root is fast. Finding a deep leaf takes much longer. The goal is to design the search tree so that the keys that are searched for often are closer to the root. The probabilities $p_1, \ldots, p_n$ given as part of the input specify the frequency with which each key is searched for, eg. $p_3 = \frac{1}{8}$ means that key a_3 is search for on average one out of every eight times.

One minimizes the depth of a binary search tree by having it be completely balanced. Having it balanced, however, dictates the location of each key. Although having the tree partially unbalanced increases its overall height, it may allow for the keys that are searched for often to be placed closer to the top.

We will manage to put some of the nodes close to the root and others we will not. The standard mathematical way of measuring the overall successes of putting more likely keys closer to the top is the expected depth of a key when the key is chosen randomly according to the given probability distribution. It is calculated by $\sum_{i\in[1..n]} p_i \cdot d_i$, where d_i is the depth of a_i in the search tree.

One way to understand this is to suppose that we needed to search for a billion keys. If $p_3 = \frac{1}{8}$, a_3 is searched for on average one out of ever eight times. Because we are searching for so many keys, it is almost certain that the number of times we search for this key is very close to $\frac{1}{8}$ billion. In general, the number of times we search for a_i is $p_i\times$ billion. To compute the average depth of these billion searches, we sum up the depth of each them and divide by a billion, namely $\frac{1}{\text{billion}} \sum_{k\in[1..\text{billion}]} \left[\text{depth of } k^{th} \text{ search}\right] = \frac{1}{\text{billion}} \sum_{i\in[1..n]} (p_i \times \text{billion}) \cdot d_i = \sum_{i\in[1..n]} p_i \cdot d_i$.

Bird and Friend Algorithm: I am given an instance consisting of n probabilities $p_1, \ldots, p_n$. I ask the bird which key to put at the root. She answer a_k. I ask one friend for the best binary search tree for the keys $a_1, \ldots, a_{k-1}$ and its expected depth. I ask another friend for the best tree of the specified height for $a_{k+1}, \ldots, a_n$ and its expected depth. I build the tree with a_k at the root and these as the left and right subtrees.

Generalizing the Problem Solved: A set of probabilities $p_1, \ldots, p_n$ defining a probability distribution should have the property that $\sum_{i\in[1..n]} p_i = 1$. However, we will generalize the problem by removing this restriction. This will allow us to ask our friend to solve subinstances that are not officially legal. Note that the probabilities given to the friends in the above algorithm do not sum to 1.

The Cost of an Optimal Solution Derived from the Costs for the Subinstances: The expected depth of my tree is computed from that given by my friend as follows.

$$
\begin{aligned}
Cost &= \sum_{i\in[1..n]} [p_i \cdot \text{depth of } a_i \text{ in tree}] & (16.1)\\
&= \sum_{i\in[1..k-1]} [p_i \cdot \text{ (depth of } a_i \text{ in left subtree) } +1\ ] + [p_k \cdot 1] & (16.2)\\
&\quad +\sum_{i\in[k+1..n]} [p_i \cdot \text{ (depth of } a_i \text{ in right subtree) } + 1] & (16.3)\\
&= Cost_{left} + \left[\sum_{i\in[1..k-1]} p_i\right] + p_k + Cost_{right} + \left[\sum_{i\in[k+1..n]} p_i\right] & (16.4)\\
&= Cost_{left} + \left[\sum_{i\in[1..n]} p_i\right] + Cost_{right} & (16.5)\\
&= Cost_{left} + Cost_{right} + 1 & (16.6)
\end{aligned}
$$

The Complete Set of Subinstances that Will Get Called: The complete set of subinstances is $S = \{\langle a_i,\ldots,a_j; p_i,\ldots,p_j\rangle \mid 1 \le i \le j \le n\}$. The table is 2-dimensional $\Theta(n \times n)$.

Running Time: The table has size $\Theta(n \times n)$. The bird can give n different answers. Hence, the time is $\Theta(n^3)$.

We now change the above problem so that it is looking for the best AVL Search Tree.

The Best AVL Tree Problem:

Instances: An instance consists of n probabilities $p_1,\ldots,p_n$ to be associated with the n keys $a_1 < a_2 < \ldots < a_n$.

Solutions: A solution for an instance is an AVL tree containing the keys. An AVL tree is a binary search tree with the property that every node has a balance factor of -1, 0, or 1, where its balance factor is the difference between the height of its left and its right subtrees.

Cost of Solution: The cost of a solution is the expected depth of a key $\sum_{i\in[1..n]} [p_i \cdot \text{ depth of } a_i \text{ in } T]$.

Goal: Given the keys and the probabilities, the goal is to find an AVL tree with minimum expected depth.

Generalizing the Problem Solved: Recall Section 12 we wrote a recursive program to determine whether a tree is an AVL tree. We needed to know the height of the left and the right subtree. Therefore, we generalized the problem so that it also returned the height. We will do a similar thing here.

Insuring the Balance Between Heights of Left and Right SubTrees: Let us begin by trying the algorithm for the general binary search tree. We ask the bird which key to put at the root. She answer a_k. We ask friends to build the left and the right sub-AVL trees. They could even tell us how high they are. What do we do, however, if the difference in their heights is greater than one?

We would like to ask the friends to build their AVL trees so that the difference in their heights is at most one. This, however, requires the friends to coordinate. This would be hard.

Another option is to ask the bird what height the left and right subtree should be. The bird will give you heights that are within one of each other. Then we could separately ask each friend to give the best AVL tree of the given height.

The New Problem: An instance consists of the keys, the probabilities, and a required height. The goal is to find the best AVL tree with the given height.

Bird and Friend Algorithm: An AVL tree of height h has left and right subtrees of heights either $\langle h-2, h-1\rangle$, $\langle h-1, h-1\rangle$, or $\langle h-1, h-2\rangle$. The bird tells me which of these three is the case and which value a_k will be at the root. I can then ask the friends for the best left and right subtrees of the specified height.

The Complete Set of Subinstances that Will Get Called: Recall in Section 12, we proved that the minimum height of an AVL tree with n nodes is $h = \log_2 n$ and that its maximum height is $h = 1.455 \log_2 n$. Hence, the complete set of subinstances is $S = \{\langle h; a_i, \ldots, a_j; p_i, \ldots, p_j \rangle \mid 1 \le i \le j \le n,\ h \in [\log_2(j-i+1)..1.455\log_2(j-i+1)]\}$. The table is a 3-dimensional $\Theta(n \times n \times \log n)$ box.

Running Time: The table has size $\Theta(n \times n \times \log n)$. The bird can give $3 \cdot n$ different answers. Hence, the time is $\Theta(n^3 \log n)$.

Solving the Original Problem: In the original problem, the height was not fixed. To solve this problem, we could simply run the previous algorithm for each h and take the best of these AVL trees. There is a faster way.

Reusing the Table: We do not need to run the previous algorithm more than once. After running it once, the table already contains the cost of the best AVL for each of the possible heights h. To find the best overall AVL tree, we need only compare these listed in the table.

16.3.7 Another View of a Solution: All Pairs Shortest Paths with Negative Cycles

The algorithm given here has three interesting features: 1) The question asked of the little bird is interesting; 2) Instead of organizing each solution into a sequence of answers or into a binary tree of answers, something in between is done; 3) It stores something other than the bird's advice in the table. This algorithm is due to Floyd-Warshall and Johnson.

All Pairs Shortest Weighted Paths with Possible Negative Cycles: Like the problem solved by Dijkstra's algorithm in Section 8.3 and by dynamic programming for leveled graphs in Section 15.1.3, the problem is to find a shortest-weighted path between two nodes. Now, however, we consider the possibility of cycles with negative weights and hence the possibility of optimal solutions that are infinitely long paths with infinitely negative weight. Also the problem is extended to ask for the shortest path between every pair of nodes.

Instances: An instance (input) consists a weighted graph G (either directed or undirected). Each edge $\langle u, v\rangle$ is allocated with a possibly negative weight $w_{\langle u,v\rangle} \in [-\infty, \infty]$.

Solutions: A solution consists of a data structure, $\langle \pi, cycleNode\rangle$, that specifies a path from v_i to v_j for every pair of nodes.

Cost of Solution: The cost of a path is the sum of the weights of the edges within the path.

Goal: Given a graph G, the goal is to find, for each pair of nodes, a path with minimum total weight between them.

Types of Paths: Here we discuss the types of paths that arise within this algorithm and how they are stored.

Negative Cycles: As an example, find a minimum weighted path from v_i to v_j in the graph Figure 16.7. The path $\langle v_i, v_c, v_d, v_e, v_j\rangle$ has weight $1+2+2+1=6$. The path $\langle v_i, v_a, v_b, v_j\rangle$ has weight $1+2+4=7$. This is not as good as the previous. However, we can take a detour in this path by going around the cycle, giving the path $\langle v_i, v_a, v_b,\ v_m, v_a, v_b,\ v_j\rangle$. Because the cycle has negative weight, the new weight is better, $1+2+(3+(-6)+2)+4=6$. That working well, why not go around the cycle again, bring the weight down to 5. The more we go around, the better the path. In fact, there is no reason that we cannot go around an infinite number of times, producing a path with weight $-\infty$. We have no requirement that the paths have a finite number of edges, so this is a legal path. There may or may not be other paths with $-\infty$ weight, but none can beat this. Hence this must be a path with minimum weight.

Definition of Simple Paths: A path is said to be *simple* if no node is repeated.

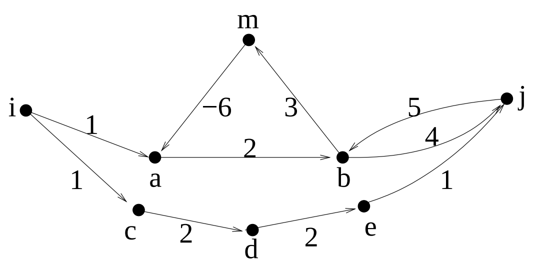


Figure 16.7: An infinite path around a negative cycle.

The π All Paths Data Structure: The problem asks for an optimal path between each pair of nodes v_i and v_j. Section 8.2 introduces a good data structure, $\pi[j]$, for storing an optimal path from one fixed node s to all nodes v_j. It can be modified to store an optimal path between every pair of nodes. Here $\pi[i,j]$ stores the second last node in the path from v_i to v_j. A problem, however, with this data structure is that it is only able to store simple paths.

Simple Paths: Consider the simple path in Figure 16.8. The entire path is defined recursively to be the path from v_i to the node given as $\pi[i,j]$, followed by the last edge $\langle\pi[i,j],v_j\rangle$ to v_j. More concretely, the nodes in the path walking backwards from v_j to v_i are v_j, $\pi[i,j]$, $\pi[i,\pi[i,j]]$, $\pi[i,\pi[i,\pi[i,j]]]$, ..., until the node v_i is reached.

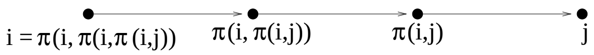


Figure 16.8: The π all paths data structure for simple paths.

Code for Walking Backwards: The following code, traces the path backwards from v_j to v_i.

algorithm *PrintSimplePath* (i,j,π)
$\langle$***pre−cond***$\rangle$**:** π specifies a path between every pair of nodes.
$\langle$***post−cond***$\rangle$**:** Prints the path from v_i to v_j backwards.
begin
  $walk = j$
  print v_j
  while($walk \neq i$)
    $walk = \pi[i, walk]$
    print v_{walk}
  end while
end algorithm

Cannot Store Paths with Cycles: The data structure π is unable to store paths with cycles for the following reason. Suppose that the path from v_i to v_j contains the node v_m more than once. π gives how to back up from v_j along this path. After reaching v_m, π gives how to back up further until v_m is reached a second time. Backing up further, π will follow the same cycle back to v_m over and over again. π is unable to describe how to deviate off this cycle in order continue on to v_i.

Storing Optimal Paths: For each pair of nodes v_i and v_j, an optimal path is stored.

Finite Paths: If the minimum weight of a path from v_i to v_j is not $-\infty$, then we know that there is an optimal path that is simple, because if a cycle has a positive weight it should not be taken and if it has a negative weight it should be taken again. Such a simple path is traced backwards by π.

Infinite Paths: If the minimum weight of a path from v_i to v_j is $-\infty$, then π cannot trace this path backwards, because it contains a negative weighted cycle. Instead, an infinite optimal path from v_i to v_j is stored as follows.

Node on Cycle $cycleNode[i,j]$: The data structure $cycleNode[i,j]$ will store one node v_m on one such negative cycle along the path.

The Path in Three Segments: The path from v_i to v_j that will be stored by the algorithm follows a simple path from v_i to v_m, followed an infinite number of times by a negative weighted simple path from v_m back to v_m, ending with a simple path from v_m to v_j. For the example in Figure 16.8, these paths are $\langle v_i, v_a, v_b, v_m, \rangle$, $\langle v_m, v_a, v_b, v_m\rangle$, and $\langle v_m, v_a, v_b, v_j\rangle$. Each of these three simple paths is stored by being traced backwards by π. More concretely, the infinite path from v_i to v_j is traced backwards as follows.

$$v_j,\ \pi[m,j],\ \pi[m,\pi[m,j]],\ \pi[m,\pi[m,\pi[m,j]]],\ \ldots,\ v_m$$

infinite cycle begins

$$\pi[m,m],\ \pi[m,\pi[m,m]],\ \pi[m,\pi[m,\pi[m,m]]],\ \ldots,\ v_m$$

infinite cycle ends

$$\pi[i,m],\ \pi[i,\pi[i,m]],\ \pi[i,\pi[i,\pi[i,m]]],\ \ldots,\ v_i$$

The following code traces this path.

algorithm $PrintPath\,(i,j,\pi,cost,cycleNode)$

$\langle$***pre−cond***$\rangle$**:** π, $cost$, and $cycleNode$ are as given by the $AllPairsShortestPaths$ algorithm. v_i and v_j are two nodes.

$\langle$***post−cond***$\rangle$**:** Prints the path from v_i to v_j backwards.

```
begin
    if( cost[i,j] ≠ −∞ ) then
        PrintSimplePath (i, j, π)
    else
        m = cycleNode[i,j]
        PrintSimplePath (i, m, π)
        print "infinite cycle begins"
        PrintSimplePath (m, π[m,m], π)
        print "infinite cycle ends"
        PrintSimplePath (i, π[i,m], π)
    end if
end algorithm
```

Additional Simple Paths: The infinite path from v_i to v_j required three simple paths, from v_i to v_m, v_m to v_m, and v_m to v_j. These will not be optimal paths from between these pairs of nodes. Hence, the algorithm must store these separately.

Complications in Find Simple Paths: Having negative cycles around, greatly complicates the algorithm's task of ensuring that the paths traced back by π are in fact simple paths. For example, if one were looking for a minimal weighted simple path from v_i to v_m, one would be tempted to include the negative cycle from v_m back to v_m. If it was included, however, the path would not be simple and π would not be able to trace backwards from v_m to v_i.

Non-Optimal Simple Paths: Luckily, there is no requirement on these three simple paths being optimal in any way. After all, the combined path, having weight $-\infty$, must be a path with minimum total weight. Therefore, to simplify the algorithm, the algorithm's only goal is to store some simple path in each of these three cases.

A Simple Path from v_i to v_j: Just as the infinite path from v_i to v_j required three simple paths, v_i to v_m, v_m to v_m, and v_m to v_j, other pairs of nodes may require a simple path from v_i to v_j. Just in case it is needed, the algorithm will ensure that π traces some non-optimal simple path backwards from v_j to v_i. In order to avoid the negative cycle, this path from will avoid the node v_m. For the example in Figure 16.8, the path stored will be $\langle v_i, v_p, v_q, v_r, v_j\rangle$.

This completes the discuss if the types of paths that arise within this algorithm and how they are stored.

The Question For the Little Bird:

The Last Edge: The algorithm in Section 15.1.3 asked the little bird for the last edge in an optimal path from v_i to v_j (equivalently, we could ask for the second last node $\pi[i,j]$) and then deleted this edge from consideration. One problem with this approach is that the optimal path may be infinitely long and deleting one edge does not make it significantly shorter. Hence, no "progress" is made. Another problem is that this edge may appear many times within the optimal solution. Hence, we cannot stop considering it so quickly.

Number of Times Last Node Appears: Section 15.1.5 suggests that if the instance is a sequence of objects and a solution is a subset of these object, than a good question may be whether the last object of the instance is included in an optimal solution? Here an instance is a set of nodes, but we can order them in some arbitrary way. Then we can ask how many times the node v_n is included internally within an optimal path from v_i to v_j. There are only $K = 3$ possible answers: $k = 0$, 1, and ∞. Note that v_n will not be included in the path only twice, because if this cycle from v_n back to v_n has a positive weight, why take the cycle, and if the cycle has a negative weight, why not go around again. For each of these $K = 3$ possible answers that the little bird may give, the best solution is found that is consistent with this answer and then the best of these best solutions is returned.

Recursive Structure: An optimal path from v_i to v_j that contains the node v_n only once consists of an optimal path from v_i to v_n that includes node v_n only at the end followed by an optimal one from v_n to v_j that includes node v_n only at the beginning.

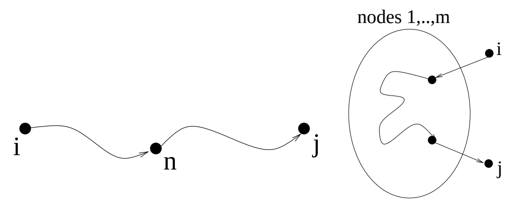


Figure 16.9: Generalizing the problem

Generalizing the Problem: This recursive structure motivates wanting to search for a shortest path between two specified nodes that excludes a specified set of nodes except possibly at the beginning and at the end of the path. By tracing the recursive algorithm, we will see that the set of subinstance used is $\{\langle G_m, i, j\rangle \mid i, j \in [1..n] \text{ and } m \in [0..n]\}$, where the goal of the instance $\langle G_m, i, j\rangle$ is to find a path with minimum total weight from v_i to v_j that includes only the nodes $v_1, v_2, \ldots, v_m$ except possibly at its beginning or end. We allow the ends of the paths to be outside of this range of nodes because the specified nodes v_i and v_j might be outside of this range. To simplify the problem, the algorithm must return only $cost_m[i,j]$ and $\pi_m[i,j]$ which are the total weight of the path found and the second last node in it.

Base Cases and The Original Instance: Note that the new instance $\langle G_n, s, t\rangle$ is our original instance asking for an optimal path from v_i to v_j including any node in the middle. Also note that the new instance $\langle G_0, i, j\rangle$ asks for an optimal path from v_i to v_j but does not allow any nodes to be included in the middle. Hence, the only valid solution is the single edge from v_i to v_j. If there is such an edge then $cost_0[i,j]$ is the weight of this edge and $\pi_0[i,j] = i$. Otherwise, $cost_0[i,j] = \infty$ and $\pi_0[i,j]$ is nil.

Your Instance Reduced to a Subinstance: Given an instance $\langle G_m, i, j\rangle$, the last node in the graph of consideration is v_m not v_n. Hence, we will ask the little bird how many times the node v_m is included internally within an optimal path from v_i to v_j. Recall that there are only $K = 3$ possible answers, $k = 0$, 1, or ∞.

v_m is Included Internally $k = 0$ Times: Suppose that the bird assures us that the node v_m does not appear internally within at least one of the shortest weighted paths from v_i to v_j.

One Friend: We could simply ignore the node v_m and ask a friend to give us a shortest weighted path from v_i to v_j that does not include node v_m. This amounts to the subinstance $\langle G_{m-1}, i, j\rangle$. See the left path from v_i to v_j in Figure 16.10.

$cost_m[i,j]$ and $\pi_m[i,j]$: The path, and hence its weight and second last node, have not changed, giving $cost_m[i,j] = cost_{m-1}[i,j]$ and $\pi_m[i,j] = \pi_{m-1}[i,j]$.

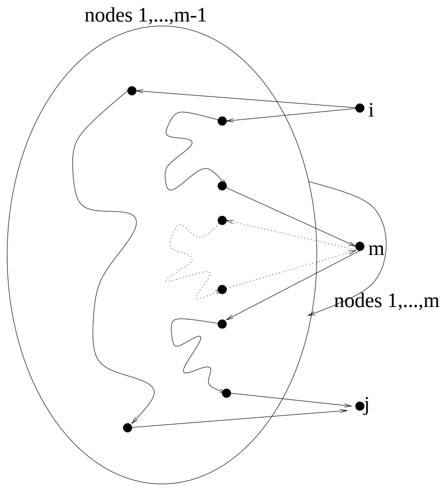


Figure 16.10: Updating the path from v_i to v_j when node v_m is included

v_m is Included Internally $k = 1$ Times: On the other hand, suppose that we are told that node v_m appears exactly once in least one of the shortest weighted paths from v_i to v_j.

Two Friends: We know that there is a shortest paths from v_i to v_j that is composed of the following two parts: a shortest weighted path from v_i to v_m that includes node v_m only at the end and a shortest one from v_m to v_j that includes node v_m only at the beginning. We can ask friends to find each of these two subpaths by giving them the subinstances $\langle G_{m-1}, i, m\rangle$ and $\langle G_{m-1}, m, j\rangle$. We combine their answers to obtain our path. See the right path from v_i to v_j in Figure 16.10 excluding the dotted part.

$cost_m[i,j]$ and $\pi_m[i,j]$: The weight of our combined path will be the sum of the weights given to us by our friends, namely $cost_m[i,j] = cost_{m-1}[i,m] + cost_{m-1}[m,j]$. The second last node in our path from v_i to v_j is the second last node in the path from v_m to v_j given to us by our second friend, namely $\pi_m[i,j] = \pi_{m-1}[m,j]$.

v_m is Included Internally $k = \infty$ Times: Finally, suppose that we are told that v_m appears infinitely often in least one of the shortest weighted paths from v_i to v_j.

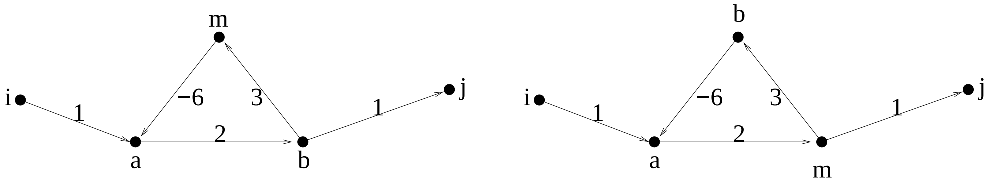
Three Friends: The only reason for an optimal path to return to v_m more than once is because there is an negative weighted cycle including v_m. Though there may be more than one such negative weighted cycle, there is no advantage in an optimal path taking different ones. Hence, there is an optimal path from v_i to v_j with the following form: an optimal path from v_i to v_m that includes node v_m only at the end; an infinitely repeating negatively weighted optimal path from v_m back to v_m that includes node v_m only at the ends; finally an optimal path from v_m to v_j that includes node v_m only at the beginning. We can ask friends to find each of these three subpaths by giving them the subinstances $\langle G_{m-1}, i, m\rangle$, $\langle G_{m-1}, m, m\rangle$, and $\langle G_{m-1}, m, j\rangle$. We combine their answers to obtain our optimal path. See the right most path from v_i to v_j in Figure 16.10 including the dotted part.

$cost_m[i,j]$: Again the weight of our combined path will be the sum of the weights given to us by our friends, namely $cost_m[i,j] = cost_{m-1}[i,m] + \infty \cdot cost_{m-1}[m,m] + cost_{m-1}[m,j]$. For this weight to be $-\infty$ as desired, there are three requirements. The cycle needs to have a negative weight, namely $cost_{m-1}[m,m] < 0$. Also there must actually be a path from v_i to the cycle, namely $cost_{m-1}[i,m] \neq \infty$, and then from the cycle onto v_j, namely $cost_{m-1}[m,j] \neq \infty$. If one of these is missing, then the bird is mistaken to tell us to visit v_m more than once. We strongly discourage this options by setting $cost_m[i,j] = +\infty$.

Node on Cycle $cycleNode[i,j]$: Recall that when there is the optimal path from v_i to v_j is infinitely long, the data structure $cycleNode[i,j]$ is to store one node v_m on one such negative cycle along the path. This is done simply by setting $cycleNode[i,j] = m$.

$\pi_m[i,j]$: In this case, the path traced backwards by π does not need to be an optimal path. The only requirement is that it is an simple path. There are two cases to consider.

The Simple Path Excluding v_m: The $k = 0$ friend, given the subinstance $\langle G_{m-1}, i, j\rangle$, can give us a simple path from v_i to v_j that does not contain v_m. This path would suit our purposes fine, assuming that it exists. We can test this by whether $cost_{m-1}[i,j] \neq \infty$. To take this path, we set $\pi_m[i,j] = \pi_{m-1}[i,j]$. Note that in the left graph below such a path exists, but that in the right one it does not.



The Simple Path Including v_m: Now suppose that a path from v_i to v_j excluding v_m does not exist. We do know that there is one that includes v_m an infinite number of times, hence there is also a truncated version of this path that includes it only once. The $k = 1$ friends, with the subinstances $\langle G_{m-1}, i, m\rangle$ and $\langle G_{m-1}, m, j\rangle$, can give this path. This path would also suit our purposes fine, assuming that it does not contain a cycle. Note that in the left graph above this path does contain a cycle, but that in the right one it does not. We have handled the left example, using the $k = 0$ path. For the right example, we use the $k = 1$ path. To take this path, we set $\pi_m[i,j] = \pi_{m-1}[m,j]$. Closer examination reveals that these are the only two cases.

This completes the steps for solving an individual instance $\langle G_m, i, j\rangle$.

Filling in The Table:

The Set of Subinstances: The claim was that the set of subinstance used is $\{\langle G_m, i, j\rangle \mid i, j \in [1..n] \text{ and } m \in [0..n]\}$.

Closed: We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub-operator. Consider an arbitrary subinstance $\langle G_m, i, j\rangle$ from this set. Applying the sub-operator constructs the subinstances $\langle G_{m-1}, i, j\rangle$, $\langle G_{m-1}, i, m\rangle$, $\langle G_{m-1}, m, m\rangle$, and $\langle G_{m-1}, m, j\rangle$, all of which are contained in the stated set of subinstances.

Generating: Starting for the original instance $\langle G_n, s, t\rangle$, the only subinstances $\langle G_m, i, j\rangle$ that will get called by the recursive algorithm are those for which v_i and v_j are either v_i and v_j or within the range $v_m, v_{m+1}, \ldots, v_n$. However, if we want to solve the all pairs shortest paths problem, then all of these subinstances will be needed.

What is Saved in the Table: As indicated in the introduction for this problem, the algorithm stores something other than the bird's advice in the table $birdAdvice_m[i,j]$. The purpose of storing the bird's advice for each subinstance is so that in the end the optimal solution can be constructed from piecing together the bird's advice for each of these subinstances. The Bird's advice is the number of times $k = 0$, 1, or ∞ that the node v_m appears in an optimal path for the instance. Pieced together, these fields of information are organized into a strange tree structure in which each field has one, two, or three children depending on whether the node v_m breaks the path into one, two, or three subpaths. See Section 15.1.4. Though this would work, it is a strange data structure to store the optimal paths. Instead, the tables store $cost_m[i,j]$ and $\pi_m[i,j]$.

The Order in which to Fill the Table: The "size" of subinstance $\langle G_m, i, j\rangle$ is the number of nodes m being considered in the middle of the paths. The tables will be filled in order of $m = 0, 1, 2, \ldots, n$.

Code:

algorithm *AllPairsShortestPaths* (G)

$\langle$***pre−cond***$\rangle$**:** G is a weighted graph (directed or undirected) with possibly negative weights.

$\langle$***post−cond***$\rangle$**:** π specifies a path with minimum total weight between every pair of nodes and *cost* their weights.

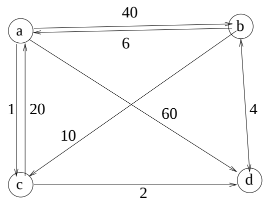
```
begin
    table[0..n, 1..n, 1..n] π, cost
    % Base Cases
    for i = 1 to n
        for j = 1 to n
            if( i = j ) then
                π_0[i,j] = nil
                cost_0[i,j] = 0
            else if( ⟨v_i, v_j⟩ is an edge ) then
                π_0[i,j] = i
                cost_0[i,j] = w_⟨v_i,v_j⟩
            else
                π_0[i,j] = nil
                cost_0[i,j] = ∞
            end if
        end for
    end for
    % Loop over subinstances in the table.
    for m = 1 to n
        for i = 1 to n
            for j = 1 to n
                % Solve the subinstance ⟨G_m, i, j⟩
                % Try possible bird answers.
                % cases k = 0
                    % Get help from one friend
                    cost_{k=0} = cost_{m-1}[i,j]
                    π_{k=0} = π_{m-1}[i,j]
                % cases k = 1
                    % Get help from two friends
                    cost_{k=1} = cost_{m-1}[i,m] + cost_{m-1}[m,j]
                    π_{k=1} = π_{m-1}[m,j]
                % cases k = ∞
                    % Get help from three friends
                    if( cost_{m-1}[i,m] ≠ ∞ and cost_{m-1}[m,m] < 0 and cost_{m-1}[m,j] ≠ ∞ ) then
                        % v_m is on a negative cycle
                        cost_{k=∞} = −∞
                        cycleNode[i,j] = m
                        if( cost_{m-1}[i,j] ≠ ∞)
                            π_{k=∞} = π_{m-1}[i,j]
```

else
$\pi_{k=\infty} = \pi_{m-1}[m, j]$
end if
else
% Little bird made a mistake
$cost_\infty = +\infty$
end if
% end cases
% Take the best bird answer. (Given a tie, take $k = 0$ over $k = 1$ over $k = \infty$.)
k_{min} = "a $k \in [0, 1, \infty]$ that minimizes $cost_k$"
$\pi_m[i, j] = \pi_{k_{min}}$
$cost[i, j] = cost_{k_{min}}$
end for
end for
end for
return $\langle \pi_n, cost_n \rangle$
end algorithm

Time and Space Requirements: The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. The number of subinstances is $\Theta(n^3)$ and the bird has K=three possible answers for you. Hence, the time and space requirements are both $\Theta(n^3)$.

Saving Space: Memory space can be saved by observing that to solve the subinstance $\langle G_m, i, j\rangle$ only the values for the previous m are needed. Hence the same table can be reused, by constructing the current table from the previous table and then coping the current to the previous and repeating. Even more space and coding hassles can be saved by not having one set of tables for the current value of m and another for the previous value, by simply updating the values in place. However, before this is done, more care is needed to make sure that this simplified algorithm still works.

Example:



$cost_0 =$

	a	b	c	d
a	0	40	1	60
b	6	0	10	4
c	20	∞	0	2
d	∞	4	∞	0

$\Pi_0 =$

	a	b	c	d
a		a	a	a
b	b		b	b
c	c			c
d		d		

$cost_a =$

	a	b	c	d
a	0	40	1	60
b	6	0	7	4
c	20	60	0	2
d	∞	4	∞	0

$\Pi_a =$

	a	b	c	d
a		a	a	a
b	b		a	b
c	c	a		c
d		d		

$cost_b =$

	a	b	c	d
a	0	40	1	44
b	6	0	7	4
c	20	60	0	2
d	10	4	11	0

$\Pi_b =$

	a	b	c	d
a		a	a	b
b	b		a	b
c	c	a		c
d	b	d	a	

$cost_c =$

	a	b	c	d
a	0	40	1	3
b	6	0	7	4
c	20	60	0	2
d	10	4	11	0

$\Pi_c =$

	a	b	c	d
a		a	a	c
b	b		a	b
c	c	a		c
d	b	d	a	

$cost_d =$

	a	b	c	d
a	0	7	1	3
b	6	0	7	4
c	12	6	0	2
d	10	4	11	0

$\Pi_d =$

	a	b	c	d
a		d	a	c
b	b		a	b
c	b	d		c
d	b	d	a	

Shorten $PrintPath$ to simply P.
Then $P(a, b) = P(a, d), b = P(a, c), d, b = P(a, a), c, d, b = a, c, d, b$.

Exercise 16.3.8 *Trace the AllPairsShortestPaths and PrintPath on the graph in Figure 16.7. Consider the nodes ordered alphabetically.*

16.3.8 Parsing with Context-Free Grammars

Recall the problem of parsing a string according to a given context-free grammar. Section 14 developed an elegant recursive algorithm which works only for look-ahead-one grammars. We now present a dynamic programming algorithm that works for any context-free grammar.

Given a grammar G and a string s, the first step in parsing is to convert the grammar into one in **Chomsky normal form**, which is defined below. (Although, a dynamic program could be written to work directly for any context-free grammar, it runs much faster if the grammar is converted first.)

The Parsing Problem:

Instance: An instance consists of $\langle G, T_{start}, s\rangle$, where G is a grammar in Chomsky normal form, T_{start} is the non-terminal of G designated as the start symbol, and s is the string $\langle a_1, \ldots, a_n\rangle$ of terminal symbols to be generated. The grammar G consists of a set of non-terminal symbols $V = \langle T_1, \ldots, T_{|V|}\rangle$ and a set of rules $\langle r_1, \ldots, r_m\rangle$. The definition of Chomsky normal form is that each rule r_q has one of the following three forms:

- $A_q \Rightarrow B_q C_q$, where A_q, B_q, and C_q are non-terminal symbols.
- $A_q \Rightarrow b_q$, where b_q is a terminal symbol.
- $T_{start} \Rightarrow \epsilon$, where T_{start} is the start symbol and ϵ is the empty string. This rule can only be used to parse the string $s = \epsilon$. It may not be used within the parsing of a larger string.

Solution: A solution is a partial parsing P, consisting of a tree. Each internal node of the tree is labeled with a non-terminal symbol, the root with the specified symbol T_{start}. Each internal node must correspond to a rule of the grammar G. For example, for rule $A \Rightarrow BC$, the node is labeled A and its two children are labeled B and C. In a complete parsing, each leaf of the tree is labeled with a terminal symbol. (In a partial parsing, some leaves may still be labeled with non-terminals.)

Cost of Solution: A parsing P is said to generate the string s if the leaves of the parsing in order forms s. The cost of P will be zero if it generates the string s and will be infinity otherwise.

Goal: The goal of the problem is, given an instance $\langle G, T_{start}, s\rangle$, to find a parsing P that generates s.

Not Look Ahead One: The grammar G might not be *look ahead one*. For example, in

A $\Rightarrow$ B C
A $\Rightarrow$ D E

you do not know whether to start parsing the string as a B or a D. If you make the wrong choice, you have to back up and repeat the process. However, this problem is a perfect candidate for a dynamic-programming algorithm.

The Parsing Abstract Data Type: We will use the following abstract data type to represent parsings. Suppose that there is a rule $r_q =$ "$A_q \Rightarrow B_q C_q$ that generates B_q and C_q from A_q. Suppose as well that the string $s_1 = \langle a_1, \ldots, a_k\rangle$ is generated starting with the symbol B_q using the parsing P_1 (B_q is the root of P_1) and that $s_2 = \langle a_{k+1}, \ldots, a_n\rangle$ is generated from C_q using P_1. Then we say that the string $s = s_1 \circ s_2 = \langle a_1, \ldots, a_n\rangle$ is generated from A_q using the parsing $P = \langle A_q, P_1, P_2\rangle$.

The Number of Parsings: Usually, the first algorithmic attempts at parsing are some form of brute-force algorithm. The problem is that there are an exponential number of parsings to try. This number can be estimated roughly as follows. When parsing, the string of symbols needs to increase from being of size 1 (consisting only of the start symbols) to being of size n (consisting of s). Applying a rule adds only one more symbol to this string. Hence, rules need to be applied $n-1$ times. Each time you apply a rule, you have to choose which of the m rules to apply. Hence, the total number of choices may be $\Theta(m^n)$.

The Question to Ask the Little Bird: Given an instance $\langle G, T_{start}, s\rangle$, we will ask the little bird a question that contains two sub-question about a parsing P that generates s from T_{start}.

The first sub-question is the index q of the rule $r_q =$ "$T_{start} \Rightarrow B_q C_q$" that is applied first to our start symbol T_{start}. Although this is useful information, I don't see how it alone could lead to a subinstance.

We don't know P, but we do know that P generates $s = \langle a_1, \ldots, a_n \rangle$. It follows that, for some $k \in [1..n]$, after P applies its first rule $r_q =$ "$T_{start} \Rightarrow B_q C_q$", it then generates the string $s_1 = \langle a_1, \ldots, a_k \rangle$ from B_q and the string $s_2 = \langle a_{k+1}, \ldots, a_n \rangle$ from C_q, so that overall it generates $s = s_1 \circ s_2 = \langle a_1, \ldots, a_n \rangle$. Our second sub-question asked of the bird is to tell us this k that splits the string s.

Help From Friend: What we do not know about of the parsing tree P is how B_q generates $s_1 = \langle a_1, \ldots, a_k \rangle$ and how C_q generates $s_2 = \langle a_{k+1}, \ldots, a_n \rangle$. Hence, we ask our friends for optimal parsings for the subinstances $\langle G, B_q, s_1 \rangle$ and $\langle G, C_q, s_2 \rangle$. They respond with the parsings P_1 and P_2. We conclude that $P = \langle T_{start}, P_1, P_2 \rangle$ generates $s = s_1 \circ s_2 = \langle a_1, \ldots, a_n \rangle$ from T_{start}. If either friend gives us a parsing with infinite cost, then we know that no parsing consistent with the information provided by the bird is possible. The cost of our parsings in this case is infinity as well. This can be achieved by setting the cost of the new parsing to be the maximum of that for P_1 and for P_2. The line of code will be $cost_{\langle q,k \rangle} = \max(cost[B_q, 1, k], cost[C_q, k+1, n])$.

The Set of Subinstances: The set of subinstances that get called by the recursive program consisting of you, your friends, and their friends is $\{\langle G, T_h, a_i, \ldots, a_j \rangle \mid h \in V,\ 0 \le i \le j \le n\}$.

Closed: We know that this set contains all subinstances generated by the recursive algorithm because it contains the initial instance and is closed under the sub-operator. Consider an arbitrary subinstance $\langle G, T_h, a_i, \ldots, a_j \rangle$ in the set. Its subinstances are $\langle G, B_q, a_i, \ldots, a_k \rangle$ and $\langle G, C_q, a_{k+1}, \ldots, a_j \rangle$, which are both in the set.

Generating: Some of these subinstances will not be generated. However, most of our instances will.

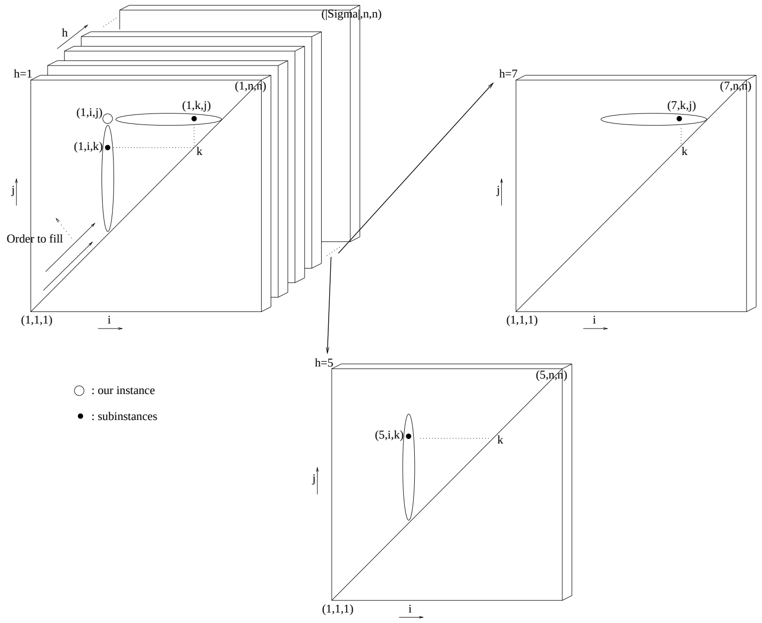


Figure 16.11: The dynamic-programming table for parsing is shown. The table entry corresponding to the instance $\langle G, T_1, a_i, \ldots, a_j \rangle$ is represented by the little circle. Using the rule $T_1 \Rightarrow T_1 T_1$, the subinstances $\langle G, T_1, a_i, \ldots, a_k \rangle$ and $\langle G, T_1, a_{k+1}, \ldots, a_j \rangle$ are formed. Using the rule $T_1 \Rightarrow T_5 T_7$, the subinstances $\langle G, T_5, a_i, \ldots, a_k \rangle$ and $\langle G, T_7, a_{k+1}, \ldots, a_j \rangle$ are formed. The table entries corresponding to these subinstances are represented by the dot within the ovals.

Constructing a Table Indexed by Subinstances: The table would be three dimensional. The solution for subinstance $\langle G, T_h, a_i, \ldots, a_j \rangle$ would be stored in entry $Table[h, i, j]$ for $h \in V$ and $0 \le i \le j \le n$.

The Order in which to Fill the Table: The size of the subinstance $\langle G, T_h, a_i, \ldots, a_j \rangle$ is the length of the string to be generated, i.e. $j - i + 1$. We will consider longer and longer strings.

Base Cases: One base case is the subinstance $\langle G, T_{start}, \epsilon\rangle$. This empty string ϵ is parsed with the rule $T_{start} \Rightarrow \epsilon$, assuming that this is a legal rule. The other base cases are the subinstances $\langle G, A_q, b_q\rangle$. This string consisting of the single character b_q is parsed with the rule $A_q \Rightarrow b_q$, assuming that this is a legal rule.

Code:

algorithm $Parsing\,(\langle G, T_{start}, a_1, \ldots, a_n\rangle)$

$\langle$***pre−cond***$\rangle$**:** G is a Chomsky normal form grammar, T_{start} is a non-terminal, and s is the string $\langle a_1, \ldots, a_n\rangle$ of terminal symbols.

$\langle$***post−cond***$\rangle$**:** P, if possible, is a parsing that generates s starting from T_{start} using G.

```
begin
    table[|V|, n, n] birdAdvice, cost

    % The case s = ε is handled separately
    if n = 0 then
        if T_start ⇒ ε is a rule then
            P = the parsing applies this one rule.
        else
            P = ∅
        end if
        return(P)
    end if

    % The base cases in the table are all subinstances of size one.
    % Solve instance ⟨G, T_h, a_i, ..., a_i⟩ and fill in table entry ⟨h, i, i⟩
    For i = 1 to n
        For each non-terminal T_h
            If there is a rule r_q = "A_q ⇒ b_q", where A_q is T_h and b_q is a_i then
                birdAdvice[h, i, i] = ⟨q, ?⟩
                cost[h, i, i] = 0
            else
                birdAdviceQ[h, i, i] = ⟨?, ?⟩
                cost[h, i, i] = ∞
            end if
        end loop
    end loop

    % Loop over subinstances in the table.
    for size = 2 to n % length of substring ⟨a_i, ..., a_j⟩
        for i = 1 to n − size + 1
            j = i + size - 1
            For each non-terminal T_h, i.e., h ∈ [1..|V|]
                % Solve instance ⟨G, T_h, a_i, ..., a_j⟩ and fill in table entry ⟨h, i, j⟩
                % Loop over possible bird answers.
                for each rule r_q = "A_q ⇒ B_q C_q" for which A_q is T_h
                    for k = i to j − 1
                        % Ask friend if you can generate ⟨a_i, ..., a_k⟩ from B_q
                        % Ask another friend if you can generate ⟨a_k+1, ..., a_j⟩ from C_q
                        cost_⟨q,k⟩ = max(cost[B_q, i, k], cost[C_q, k + 1, j])
                    end for
                end for
                % Take the best bird answer, i.e. one of cost zero if s can be generated.
```

$\langle q_{min}, k_{min}\rangle$ = "a $\langle q, k\rangle$ that minimizes $cost_{\langle q,k\rangle}$"
$birdAdviceQ[h, i, j] = \langle q_{min}, k_{min}\rangle$
$cost[h, i, j] = cost_{\langle q_{min}, k_{min}\rangle}$
end for
end for
end for

% Constructing the solution P
if($cost[1, 1, n] = 0$) then % i.e. if s can be generated from T_{start}
$P = ParsingWithAdvice\left(\langle G, T_{start}, a_1, \ldots, a_n\rangle, birdAdvice\right)$
else
$P = \emptyset$
end if
return(P)
end algorithm

Constructing an Optimal Solution:

algorithm $ParsingWithAdvice\left(\langle G, T_h, a_i, \ldots, a_j\rangle, birdAdvice\right)$

$\langle$***pre & post−cond***$\rangle$**:** Same as $Parsing$ except with advice.

begin
$\langle q, k\rangle = birdAdvice[h, i, j]$
if($i = j$) then
Rule r_q must have the form "$A_q \Rightarrow b_q$", where A_q is T_h and b_q is a_i
Parsing $P = \langle T_h, a_i\rangle$
else
Rule r_q must have the form "$A_q \Rightarrow C_q B_q$", where A_q is T_h
$P_1 = ParsingWithAdvice\left(\langle G, B_q, a_i, \ldots, a_k\rangle,\ birdAdviceQ, birdAdviceK\right)$
$P_2 = ParsingWithAdvice\left(\langle G, C_q, a_{k+1}, \ldots, a_j\rangle,\ birdAdviceQ, birdAdviceK\right)$
Parsing $P = \langle T_h, P_1, P_2\rangle$
end if
return(P)
end algorithm

Time and Space Requirements: The running time is the number of subinstances times the number of possible bird answers and the space is the number of subinstances. The number of subinstances indexing your table is $\Theta(|V|n^2)$, namely $Table[h, i, j]$ for $h \in V$ and $0 \le i \le j \le n$. The number of answers that the bird might give you is at most $\mathcal{O}(mn)$, namely $\langle q, k\rangle$ for each of the m rules r_q and split $k \in [1..n-1]$. This gives $time = \mathcal{O}(|V|n^2 \cdot mn)$. If the grammar G is fixed, then the time is $\Theta(n^3)$.

A tighter analysis would note that the bird would only answer q for rules r_q = "$A_q \Rightarrow B_q C_q$", for which the left-hand side A_q is the non-terminal T_h specified in the instance. Let m_{T_h} be the number of such rules. Then the loop over non-terminals T_h and the loop over rules r_q would not require $|V|m$ time, but $\sum_{T_h \in V} m_{T_h} = m$. This gives a total time of $\Theta(n^3 m)$.