GREEDY ALGORITHMS (Lectures 3-4)

Build a solution in small steps, choosing a decision at each step to optimize some criterion

→ There is a local decision rule to construct optimal solutions.

Two analysis tools:

1. "Greedy stays ahead": measure progress step-by-step
2. "Exchange argument": consider any possible solution to the problem and gradually transform it to greedy algorithms solution (so greedy is at least as good).

1. Interval scheduling.

n requests $\{(s_1, f_1), \dots, (s_n, f_n)\}$, choose largest possible subset of mutually compatible subsets.

- Design a rule for picking the first interval, then discard any uncompatible intervals

→ Choose interval which finishes first.

Proof idea: • Want solution to be same size as optimal solution
• Compare partial solutions to "segments" of optimal solution. and show greedy does at least as well.

"$\forall\ r \leq k,\ f(\cancel{A}i_r) \leq f(j_k)$)

$\uparrow$ A $\qquad$ $\uparrow$ OPT

☆ often by contradiction: contradicting a basic property.

USE Contradiction: as a proof strategy, also helps to disprove

(2) MST

Refresher: <u>trees</u> connected, undirected, acyclic graph.

Let G be an undirected graph on n nodes. Then any two of the following imply the third:

(1) G is connected

(2) G is acyclic

(3) G has n-1 edges

MST: a spanning tree w/ minimum total cost

(1) A lightest edge in any cut always belongs to an MST

(2) The heaviest edge in a cycle never belongs to an MST (unless all edges in the cycle are the same cost)

<u>Kruskal</u>.

$X = \{\}$

For each edge e in increasing order of cost:

if e adjoins two distinct components in

$X = X \cup \{e\}$

"exchange": ~~The output~~ Any minimum cost spanning tree can have been output by Kruskal's

why? Our choice of edges respect properties (1), (2) .. ~~and we~~

DIVIDE-AND-CONQUER (Lectures 5-7)

Design Principle

1. Break the problem into subproblems that are smaller instances of the same problem
2. Recursively solve the subproblems
3. combine the answers to the smaller subproblems to solve the larger problem

Work is done in

A • dividing

b • solving at the base of the recursion

c • combining

Sample-d-a-c(A):
if base case
Divide A into A_1, A_2
$a_1 \leftarrow$ sample-d-a-c(A_1)
$a_2 \leftarrow$ sample-d-a-c(A_2)
$a \leftarrow$ combine (a_1

Sample-DAC(A):

    If (A is the base of the recursion):
        [b] return outright computation for A
    else:
        [A] Divide A into subproblems $A_1, \ldots, A_k$
            Obtain solutions to Sample-DAC(A_1), ..., Sample-DAC(A_k)
        [c] return answer to A by combining answers

I Divide and Conquer Principles

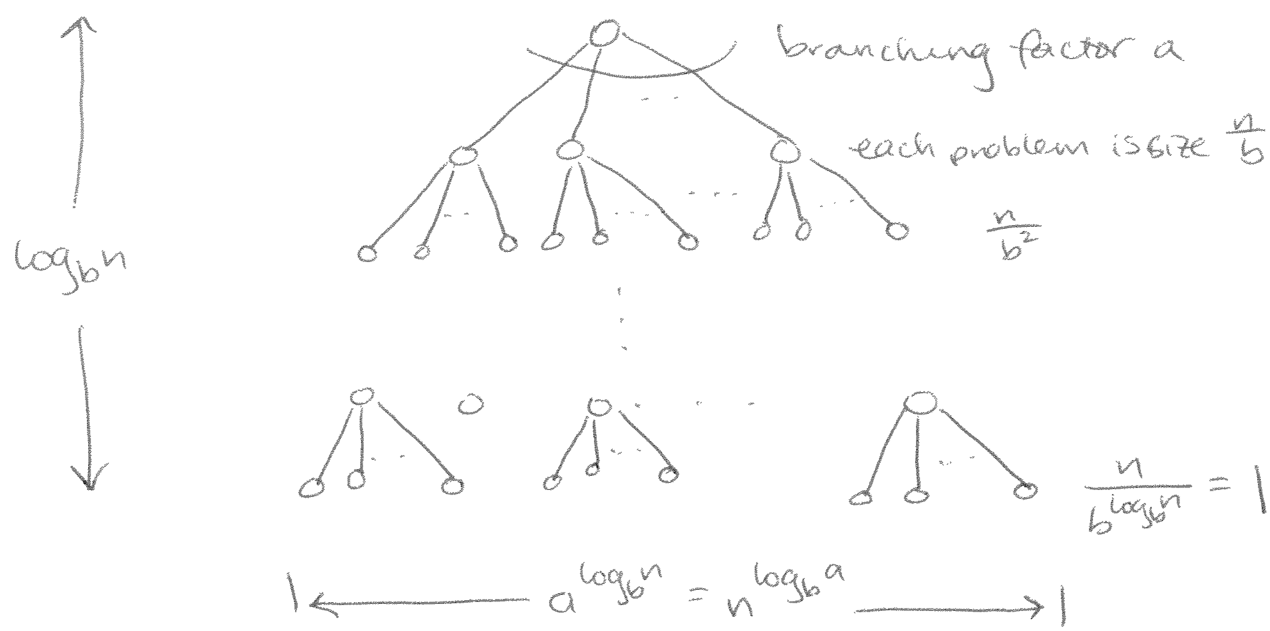
Idea: Divide a problem into smaller subproblems, recursively solve the smaller problems to solve the big problem.

When we divide evenly,
Divide a problem of size n into $\underline{a}$ subproblems of size $\frac{n}{\underline{b}}$, using $\underline{O(n^d)}$ work to solve each subproblem.

Recursion ~ Recurrence relation.

Total work: $T(n) = aT(\frac{n}{b}) + O(n^d)$.

A. Recurrences



To analyze runtime, compare # of levels (depth of recursion) and amount of work done at each level.

B. Master Theorem:

If $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$:

1.) $d < \log_b a$	$b^d < a$	$\Rightarrow T(n) = O(n^{\log_b a})$ "Too many leaves": leaves outweigh the cost of combining, so width dominates.
2.) $d = \log_b a$	$b^d = a$	$\Rightarrow T(n) = O(n^{\log_b a} \log n) = O(n^d \log n)$ "equal work per level": Running time is simpl cost per level $(n^{\log_b a})$ x # levels $(\log n)$
3.) $d > \log_b a$	$b^d > a$	$\Rightarrow T(n) = O(n^d) = \Omega(n^{\log_b a + \varepsilon})$ "Too expensive a root": when combining, costs grow quickly in n, so combining at the root dominates.

ex 1. Solve $T(n) = 2T\left(\frac{n}{3}\right) + 1$

A. expanding:

$$T(n) = 2T\left(\frac{n}{3}\right) + O(1)$$

$$= 2\left[2T\left(\frac{n}{3^2}\right) + O(1)\right] + O(1) = 2^2 T\left(\frac{n}{3^2}\right) + O(2^1 + 1)$$

$$= 2^2\left[2T\left(\frac{n}{3^3}\right) + O(1)\right] + O(2+1) = 2^3 T\left(\frac{n}{3^k}\right) + O(2^2 + 2^1 + 1)$$

$$= \cdots = 2^k T\left(\frac{n}{3^k}\right) + O(2^{k-1} + \cdots + 2^3 + 2^2 + 2 + 1)$$

$$= 2^k T\left(\frac{n}{3^k}\right) + O(2^k)$$

$\uparrow$

$T(1)$ $\rightsquigarrow$ take $3^k = n \Rightarrow k = \log_3 n$

$$= 2^{\log_3 n} T(1) + O(2^{\log_3 n})$$

$$= n^{\log_3 2} + O(n^{\log_3 2}) = O(n^{\log_3 2})$$

B. Master THM.

$T(n) = 2T(\frac{n}{3}) + O(1)$

$a = 2$, $b = 3$, $d = 0$

$0 < \log_3 2 \Rightarrow T(n) = O(n^{\log_3 2})$

ex2. $T(n) = 7T(\frac{n}{7}) + O(n)$

A. expanding:

$T(n) = 7T(\frac{n}{7}) + O(n)$

$= 7[7T(\frac{n}{7^2}) + \frac{cn}{7}] + cn = 7^2 T(\frac{n}{7^2}) + 2cn$

$= 7^2[7T(\frac{n}{7^3}) + \frac{cn}{7^2}] + 2cn = 7^3 T(\frac{n}{7^3}) + 3cn$

$= \cdots 7^k T(\frac{n}{7^k}) + kcn$

take $k = \log_7 n$

$= 7^{\log_7 n} T(1) + n \log_7 n$

$= n + n\log n = O(n \log n)$

B. Master THM.

$a = 7$ $b = 7$ $d = 1$

$1 = \log_7 7 \Rightarrow T(n) = O(n \log n)$.

DYNAMIC PROGRAMMING (Lectures 7–10)

☆ Iterate over subproblems rather than solving recursively.

Main guiding principle: ① A recurrence equation that expresses the optimal solution (or its value) in terms of the optimal solutions to smaller problems.
② Use memoization

Refresher example: String reconstruction.

Design Principles

① Divide the problem into subproblems

- polynomial # subproblems
- solutions to original problem can be easily computed from solutions to the subproblems
- There is a natural ordering on subproblems and an easy-to-compute recurrence that allows solution of one subproblem from solution of smaller subproblems.

Common subproblems:

(i) input is $x_1, \dots, x_n$ and a subproblem is $x_1, \dots, x_i$:

$$\boxed{x_1 \quad x_2 \quad x_3 \quad \cdots \quad x_i} \quad x_{i+1} \quad \cdots \quad x_n$$

LINEAR # subproblems

(ii) input: $x_1, \dots, x_n, y_1, \dots, y_m$. Subproblem: $x_1, \dots, x_i, y_1, \dots, y_j$

$$\boxed{\begin{matrix} x_1 & x_2 & \cdots & x_i \\ y_1 & y_2 & \cdots & y_j \end{matrix}} \begin{matrix} x_{i+1} & \cdots & x_n \\ y_{j+1} & \cdots & y_m \end{matrix}$$

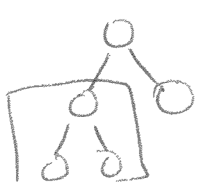
$O(mn)$ subproblems

(iii) input $x_1, \dots, x_n$ subproblem: $x_i, \dots, x_j$

$$x_1 \quad x_2 \quad \cdots \boxed{x_i \quad x_{i+1} \quad \cdots \quad x_j} \; x_{j+1} \quad \cdots \quad x_n$$

$O(n^2)$ subproblems

(iv) input: rooted tree. Subproblem: rooted subtree



Linear # subproblems.

Design Principles

② Define the subtasks recursively (express larger subtasks in terms of smaller ones)

(i) binary choice (member/non member?)
- interval scheduling
- string reconstruction

(ii) multiway choice
- segmented least squares (max over $1 \le i \le j$)
- max score pruning

(iii) adding a variable (2-dimensional array)
- knapsack
- balanced path

(iv) dynamic programming over intervals (use >1 subproblem)
- RNA secondary structure

③ Find the right order for solving subtasks
From ②:

(i) $L \to R$, $R \to L$, ...?

(ii) $L \to R / T \to B$, (i: $1 \to n$, j: $1 \to i$), ...

(iv) leaf $\to$ root, root $\to$ leaves?, ...

④ Proof of correctness
- usually by induction
- recurrence relation from ② could be the inductive argument

⑤ running time analysis
- bear in mind # subtasks!

Steps for
<u>Analysis</u> example: String reconstruction

reminder: Given an array $x[1 \cdots n]$ of characters, and a function dict(w) which returns T if w is a valid word.

Problem: is x a sequence of valid words?

① Define the subproblem

Let $S(k) = \begin{cases} T & \text{if } x[1 \cdots k] \text{ is a valid sequence of words} \\ F & \text{o/w} \end{cases}$

② Express the subproblem recursively

$S(k) = T$ if $\exists\, j < k$ s.t. $S(j) = T$ && $dict(x[j+1, \ldots, k]) = T$

③ Order of computation, base case, final solution

"L to R": $S(1), S(2), \ldots, S(n)$

☆ do not solve <u>recursively</u>, but <u>iteratively</u>

base case: $S(1) = dict(x[1])$

final solution $S(n)$

④ (3a) Value vs. Solution

reconstructing the string:

Define $D(1, \ldots, n)$:

if $S(k) = T$, $D(k)$ = starting position of word that ends at $x[k]$

ANARRAY
TTTFFFT
113ϕϕϕ3

④ Prove correct by <u>induction</u>

Base case

Inductive argument uses ②

⑤ Running time analysis:

#subproblems, time to solve each (here, n and $T(S(k)) = O(k)$)

$1 + 2 + \cdots + n$

$\Rightarrow O(n^2)$

Another example: weighted interval scheduling

Given a set of n interval requests of the form (s_i, f_i), and associated value v
Select a subset $S \subseteq \{1, \dots, n\}$ of mutually compatible intervals which maximizes
$\sum_{i \in S} v_i$.

* sort the intervals by finish time $f_1 \le f_2 \le \dots \le f_n$, and let $p(j)$ be the latest occurring interval i before (s_j, f_j) s.t. $(s_i, f_i) \cap (s_j, f_j) = \emptyset$. (or $p(j) = 0$)

① OPT(j) = value of optimal solution to problem consisting of requests $\{1, \dots, j\}$

② $OPT(j) = \max\{\underbrace{v_j + OPT(p(j))}_{\text{include interval } j}, \underbrace{OPT(j-1)}_{\text{do not include interval } j}\}$

③ OPT(1), OPT(2), ..., OPT(n)

base case: $OPT(1) = v_1$

final solution: OPT(n)

③a Solution + value:

j belongs to $\mathcal{O}$ for $\{1, \dots, j\}$ $\Leftrightarrow$ $v_j + OPT(p(j)) \ge OPT(j-1)$

"if $v_j + M[p(j)] \ge M[j-1]$ add j to solution + find_solution(p(j))

else solution ← find_solution(j-1)"

④ "Request j belongs to an optimal solution on $\{1, \dots, j\}$ $\Leftrightarrow$

$v_j + OPT(p(j)) \ge OPT(j-1)$"

⑤ Assuming sorted: $O(n)$