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:


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.

⇒ A requests \( (s_1, f_1), \ldots, (s_n, f_n) \), choose largest possible subset of mutually compatible subsets.

⇒ Design a rule for picking the first interval, then discard any incompatible 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.

\[
\forall r \in \mathbb{R}, f(A \cup [r, \infty)) < f(B) \\
\text{⇒ } A \text{ is } \text{OPT}
\]

⇒ Often by contradiction: contradicting a basic property.

USE CONTRADICTION: as a proof strategy, also helps to disprove
MST

Refresher: trees 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 with 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)

Kruskal:

\[ X = \emptyset \]

For each edge \( e \) in increasing order of cost:
if \( e \) adjoins two distinct components in
\[ X = X \cup e \]

"Exchange": Theorem: Any minimum cost spanning tree can have
been output by Kruskal's

Why? Our choice of edges respect properties 1, 2, and...
**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 DAC (A):

If (A is the base of the recursion):
- return outright computation for A

else:
- Divide A into subproblems A₁, ..., A_k
- Obtain solutions to Sample_DAC(A₁), ..., Sample_DAC(A_k)
- return answer to A by combining answers
Divide and Conquer Principles

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

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

Recursion vs Recurrence relation.

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

A. Recurrences

To analyze runtime, compare the number of levels (depth of recursion) and amount of work done at each level.
B. Master Theorem:

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

1.) \( d < \log_b a \) \( \quad \) \( b^d < a \) \( \Rightarrow \) \( T(n) = O(n \log_b a) \)

"Too many leaves": leaves outweigh the cost of combining, so width dominates.

2.) \( d = \log_b a \) \( \quad \) \( b^d = a \) \( \Rightarrow \) \( T(n) = O(n \log_b a \log n) = O(n^d \log n) \)

"Equal work per level": Running time is simply cost per level \( (n \log_b a) \times \) levels \( (\log n) \).

3.) \( d > \log_b a \) \( \quad \) \( b^d > a \) \( \Rightarrow \) \( T(n) = O(n^d) = \Omega (n \log_b a + \varepsilon) \)

"Too expensive a root": when combining, costs grow quickly in \( n \), so combining at the root dominates.

Exh. Solve \( T(n) = 2T(\frac{n}{3}) + 1 \)

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(1) \cdot 2^1 + O(1) \\
= 2^2 \left[ 2T\left(\frac{n}{3^3}\right) + O(1) \right] + O(1) \cdot 2^3 T\left(\frac{n}{3^4}\right) + O(1) \cdot 2^2 + 2^1 + O(1) \\
= \cdots = 2^k T\left(\frac{n}{3^k}\right) + O(1) \cdot 2^k + O(1) \cdot 2^{k-1} + \cdots + 2^3 \cdot 2^2 + 2^1 \cdot 2^0 + 1 \\
\uparrow \\
= 2^k + T\left(\frac{n}{3^k}\right) + O(2^k) \\
T(1) \sim \text{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.

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

\[ a = 2, \ b = 3, \ c = 0 \]

\[ 0 < \log_3 2 \Rightarrow T(n) = O(n \log_3 n) \]

\[ 2^{\log_3 n} = T\left(\frac{n}{3}\right) + O(n) \]

A. Expanding:

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

\[ = 2\left[2T\left(\frac{n}{9}\right) + \frac{cn}{3}\right] + cn = 2^2 T\left(\frac{n}{9}\right) + 2cn \]

\[ = 2^2 \left[2T\left(\frac{n}{27}\right) + \frac{2n}{9}\right] + 12cn = 2^3 T\left(\frac{n}{27}\right) + 3cn \]

\[ = \ldots 2^{\log_3 n} T\left(\frac{n}{3^{\log_3 n}}\right) + \log_3 n \cr \text{take } b = \log_3 n \]

\[ = 2^{\log_3 n} T(1) + n \log_3 n \]

\[ = n + n \log n = O(n \log n) \]

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: 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

(i) 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 \( x_1, \ldots, x_n \) and a subproblem is \( y_{i+1} \ldots x_i \):

\[
\begin{array}{cccc}
\, & x_1 & x_2 & \ldots & x_i & x_{i+1} & \ldots & x_n \\
\end{array}
\]

Linear # subproblems

(ii) Input: \( x_1, \ldots, x_n, y_{i+1} \ldots y_j \). Subproblem: \( x_{i+1} \ldots x_i, y_{i+1} \ldots y_j \):

\[
\begin{array}{cccc}
\, & x_1 & x_2 & \ldots & x_i & x_{i+1} & \ldots & x_n \\
\, & y_1 & y_2 & \ldots & y_j & y_{j+1} & \ldots & y_m \\
\end{array}
\]

\( O(mn) \) subproblems

(iii) Input: \( x_1, \ldots, x_n \) subproblem: \( x_{i+1} \ldots x_j \):

\[
\begin{array}{cccc}
\, & x_1 & x_2 & \ldots & x_i & x_{i+1} & \ldots & x_j & x_{j+1} & \ldots & x_n \\
\end{array}
\]

\( O(n^2) \) subproblems

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

\[
\begin{array}{c}
\text{Linear # subproblems:}
\end{array}
\]
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) multi-way choice
      - segmented least squares (max over $i, e, i, g$)
      - max score pruning
   (iii) adding a variable (2-dimensional array)
      - knapsack
      - balanced path
   (iv) dynamic programming over intervals (use $\geq 1$ subproblem)
      - RNA secondary structure

3. Find the right order for solving subtasks
   from (ii):
   (i) $L \rightarrow R, R \rightarrow L, \ldots$?
   (ii) $L \rightarrow B, T \rightarrow B, (i \rightarrow n, j \rightarrow i), \ldots$
   (iv) least -> root -> leaves?

4. Proof of correctness
   * usually by induction
   * recurrence relation from (ii) could be the inductive argument

5. Running time analysis
   * bear in mind # subtasks!
Steps for example: String reconstruction

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

Problem: is \(x\) a sequence of valid words?

1. Define the subproblem
   
   Let \(S(k) = \begin{cases} 
   T & \text{if } x[1\ldots k] \text{ is a valid sequence of words} \\
   F & \text{otherwise}
   \end{cases} \)

2. Express the subproblem recursively
   
   \(S(k) = T \text{ if } \exists j \leq k \text{ s.t. } S(j) = T \text{ and } \text{dict}(x[j+1\ldots k]) = T\)

3. Order of computation, base case, final solution
   
   "1 to \(n\): \(S(1), S(2), \ldots, S(n)\)
   
   Do not solve recursively, but iteratively

   Base case: \(S(1) = \text{dict}(x[1])\)
   
   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 \(x[k]\)

   \(\begin{array}{l}
   \text{ANARRAY} \\
   \text{T,T,F,F,F,F,F,F,F} \\
   \text{1,3,8,9,9,3}
   \end{array}\)

5. Prove correct by induction
   
   Base case

   Inductive argument uses (2)

6. Running time analysis:
   
   \#subproblems, time to solve each (here, \(n\) and \(T(S(k)) = O(1)\))

   \[1 + 2 + \ldots + n\]

   \[\Rightarrow O(n^2)\]
Another example: weighted interval scheduling

Given a set of $n$ interval requests of the form $(s_i, f_i)$, and associated values $v_i$, select a subset $S \subseteq \{1, \ldots, n\}$ of mutually compatible intervals which maximizes

$$\sum_{i \in S} v_i.$$

Sort the intervals by finish time $f_1 \leq f_2 \leq \cdots \leq f_n$ and let $p(j)$ be the latest occurring interval 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 $\{1, \ldots, j\}$

2. $OPT(j) = \max\{v_j + OPT(p(j)), OPT(j-1)\}$ under $\begin{cases} \text{include interval } j \\ \text{do not include interval } j \end{cases}$

3. $OPT(1), OPT(2), \ldots, OPT(n)$
   - base case: $OPT(1) = v_1$
   - final solution: $OPT(n)$

4. (a) Solution + value:
   
   $j$ belongs to $S$ for $\{1, \ldots, j\} \iff v_j + OPT(p(j)) \geq OPT(j-1)$
   
   "if $v_j + M[p(j)] \geq M[j-1]$, add $j$ to solution + find solution $(p(j))$ else solution = find solution $(j-1)$"

5. "Request $j$ belongs to an optimal solution on $\{1, \ldots, j\} \iff v_j + OPT(p(j)) \geq OPT(j-1)"

5. Assuming sorted: $O(n)$