LECTURE 5: DYNAMIC PROGRAMMING

ADVANCED ALGORITHMS
ANNOUNCEMENTS

▸ Homework 1 due on Monday Sep 10, 11:59 PM

▸ Instructor office hours Thursday -> Friday (1 - 2 pm)

▸ Comment on notes – JeffE’s material

▸ Solutions to HW0 on course page

▸ Pecking order problem of HW1

![Diagram of vertices and birds, potentially indicating cycles or relationships between them. The text on the diagram suggests considering cycles as the pecking order.]
LAST CLASS

- Divide and conquer
  - Break instance into smaller pieces, solve pieces separately, combine solutions (*)
  - Merge-sort, multiplying $n$ digit numbers (time $<< n^2$)
  - Median (and $k$'th smallest element) in $O(n)$ time (no sorting!)
  - **Key idea:** approximate median to break array into “nearly equal” pieces
  - All analyses by writing recursion, solving (tree, plug-and-chug, ...)

Master theorem
Akra-Bazzi theorem

(Notes on homepage; Jeff E’s notes)
OVERLAP IN RECURSIVE CALLS

- Divide & conquer: usually sub-problems don’t interact “much”

- What if they do? Can run into wastefulness – solving same sub-problem many times...

- Motivation for *dynamic programming*
TODAY’S PLAN

▸ Continue example – subset sum
▸ Knapsack problem
▸ Traveling salesman
RECAP: SUBSET SUM

**Problem:** given a set of non-negative integers $A[0], A[1], \ldots, A[n-1]$, and an integer $S$, find if there is a subset of $A[i]$ whose sum is $S$.

- **Trivial algorithm:** $2^n$ time – check all subsets
- **Sometimes better:** divide into two halves – check all possibilities for how sum is divided

\[
T(n) = \frac{S}{2} \cdot T(n/2) \implies T(n) = S^{\log_2 n} = n^{\log_2 S}
\]

$S = 30 \quad 1, 5, 7, 4, 15, 3, \ldots$

\[
2^{(S+1)} = n^{1 + \log_2 S}
\]
**Problem:** given a set of non-negative integers \( A[0], A[1], \ldots, A[n-1] \), and an integer \( S \), find if there is a subset of \( A[i] \) whose sum is \( S \).

- See if first element is picked or not

Suppose we can write

\[
S = \sum_{i=1}^{k} A[i] = A[i_1] + A[i_2] + \ldots + A[i_k]
\]

- \( A[0] \) is used to write \( S \)
- \( A[0] \) is NOT used to write \( S \)

See if we can write \( S \) using \( A[1], \ldots, A[n-1] \)

\( \text{recursive call} \)

\( T(n) \): runtime for procedure

when we have \( n \) \( A[i] \)'s.

Correctness is immediate—we are checking all possibilities.

See if \( S - A[0] \) can be written as using \( A[i], \ldots, A[n-1] \).
RUNNING TIME

$T(n)$: runtime when we have $m$ $A[i,j]$'s.

\[ T(n) = 2T(n-1) + 1 \]

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

\[ \approx 2 \cdot 2^n \]

(almost as bad as brute-force!)
A[] = {1, 2, 3, 5, 7, 9, 10, 11};  S = 20

Start index = 2
Sum needed = 17
AVOIDING MULTIPLE SOLVES ON “SAME” INSTANCE

- Store answers!
  - How much are we storing?
  - Running time

How many different sub-problems can one have? \((< 2^n)\)

- Subsets of the \(A[i]\)’s
- Only \(n\) possibilities! \((S+1)\) possibilities

\# of distinct instances is \(\leq n(S+1)\)
Modified algorithm:

→ In each recursive step, first check if we have solved this instance before! If so, read off the answer.

→ If not, store the answer after computation!

Answers to computation can be stored in an $n \times (s+1)$ array!

[Start index, sum needed].
IS THIS POLYNOMIAL? (subtle point)

When we say poly time algorithm, we mean polynomial in INPUT SIZE

$$a_0, a_1, \ldots, a_{n-1} \in S$$

$$\lceil \log a_0 \rceil + \lceil \log a_1 \rceil + \ldots + \lceil \log_2 a_{n-1} \rceil + \lceil \log S \rceil$$

Running time = $n \cdot S$

Input size $\approx n^2$

Running time $= n \cdot 2^n$

Example: each $a_i$ is

$\in \left(\frac{2^n}{2}, 2^n\right)$, and say $S \in \left[2^n, n \cdot 2^n\right]$
Problem: suppose we are given coins of denominations 1c, 5c, 10c, 20c, 25c, 50c. Make change for $1.90 using the fewest # of coins.

Heuristics?

Greedy: Always use the largest coin & that you CAN use.
RECURSIVE ALGORITHM

denominations \{d_0, d_1, \ldots, d_{n-1}\}

target sum = S

don't use \(d_0\) once

each is a recursive call:

\[ \begin{align*}
S & \rightarrow (S, \text{using } \{d_1, \ldots, d_{n-1}\}) \rightarrow \text{ans1} \\
S - d_0 & \rightarrow (S - d_0, \text{using } \{d_1, \ldots, d_{n-1}\}) \rightarrow \text{ans2} \\
S - 2d_0 & \rightarrow (S - 2d_0, \{d_1, \ldots, d_{n-1}\}) \rightarrow \ldots
\end{align*} \]
RUNNING TIME

All recursive calls are of the type:

\[ \exists S, \text{ set of denominations} \]

\[ \exists \text{ suffix of } S_{d_0}, \ldots, d_{n-1} \]

\[ \exists \text{ sum needed} \]

Space needed to store answers = \( O(nS) \)

Time needed = \( O(n \cdot S \cdot \frac{S}{d_0}) \)
GENERAL IDEA OF DYNAMIC PROGRAMMING

▸ Write down recursive algorithm

▸ Observe “common sub-problems” in recursion

▸ **Capture sub-problems “succinctly”**
**Problem:** given an undirected graph with edge weights, find a path that visits every vertex precisely once and returns to start.
TRAVELING SALESMAN PROBLEM

Problem: given an undirected graph with edge weights, find a path that visits every vertex precisely once and returns to start.

▸ Trivial algorithm?
**TRAVELING SALESMAN PROBLEM**

**Problem:** given an undirected graph with edge weights, find a path that visits every vertex precisely once and returns to start.

- Trivial algorithm?
- Many heuristics for geometric cases
RECURSIVE ALGORITHM
RECURSIVE ALGORITHM — CORRECTNESS
RECURSIVE ALGORITHM — CORRECTNESS
SUB-PROBLEMS, RUNNING TIME
CAN WE DO BETTER?

- Likely not – NP hard problem
- We will see that finding “approximately good” solution possible!