Fundamental Algorithms

Homework 4

Due: June 28th, 5:30 PM EST

Instructions

Please answer each **Problem** on a separate page. Submissions must be uploaded to your account on Gradescope by the due date and time above.

Problems to submit

Problem 1 (25 points)

Consider the array $A[1 \dots n]$ consisting of n non-negative integers. There is a frog on the last index of the array, i.e. the nth index of the array. In each step, if the frog is positioned on the i^{th} index, then it can make a jump of size at most $A[i]$ towards the beginning of the array. In other words, it can hop to any of the indices $i, \dots, i - A[i]$.
The goal is to develop a DP-based algorithm to determine whether the frog can reach the 1st index of the array.
For $i = 1, 2, \dots, n$, define the subproblem $\textsc{CanReach}(i)$ to denote true or false depending on whether the frog can reach the 1st index of the array when it is currently standing on the ith index (so $\textsc{CanReach}(i) =$ true, if it can reach the 1st index, and $\textsc{CanReach}(i) =$ false, if it cannot reach the 1st index).

(a) Find the recursion that $\textsc{CanReach}(n)$ satisfies. In other words, you should write $\textsc{CanReach}(n)$ in terms of some of $\{\textsc{CanReach}(n-1), \textsc{CanReach}(n-2), \dots, \textsc{CanReach}(1)\}$. Fully Justify your answer.

Hint: What options does the frog have for the first jump? It can jump to any of the following indices: $n, \dots, n - A[n]$.

(b) Identify the base case for your recursion in part (a) and find its corresponding value. Justify your answer.

(c) Write the pseudo-code for the bottom-up DP algorithm to compute $\textsc{CanReach}(n)$. Justify that the run-time of your algorithm is $O(n^2)$ in the worst-case.

Problem 2 (25 points)

Suppose we want to find the number of ways to make change for n cents with the use of dimes (10 cents), nickels (5 cents), and pennies (1 cent). Note that the ordering of coins in the change matters! (see the examples below)
For $i = 1, \dots, n$, define the subproblem $\textsc{MakeChange}(i)$ to denote the number of ways to make change for i cents with the use of dimes (10 cents), nickels (5 cents), and pennies (1 cent).
Below, you can find the number of ways to make change for i cents for $i = 1, \dots, 7$:

- $i = 1, \dots, 4$: There is only one way to do that. We have to change only using pennies.
- $i = 5$: There are two ways: $1+1+1+1+1$ (use 5 pennies), or 5 (use one nickel).
- $i = 6$: There are three ways: $1+1+1+1+1+1$ (use 6 pennies), or $1+5$ (one penny and one nickel), or $5+1$ (one nickel and one penny).

- $i = 7$: There are four ways: $1+1+1+1+1+1+1$ (use 7 pennies), or $1+1+5$ (two pennies and one nickel), or 1+5+1 (one penny, one nickel, and one penny), or 5+1+1 (one nickel and two pennies).

(a) Find the values of MakeChange(8) and MakeChange(9).

(b) Find the recursion that MakeChange(n) satisfies for $n \geq 10$. In other words, you should write MakeChange(n) in terms of some of {MakeChange($n-1$), MakeChange($n-2$), ..., MakeChange(1)}. Fully justify your answer.

Hint: What options do we have for the first coin of the change?

(c) Write the pseudo-code for the bottom-up DP algorithm to compute MakeChange(n). Find and justify the time complexity of your algorithm in the form of $\Theta(.)$.

Problem 3 (25 points)

Given two strings $S[1 \dots n]$ and $T[1 \dots m]$, let LCS(n, m) denote the length of the longest common substring of $S[1 \dots n]$ and $T[1 \dots m]$. Note that unlike a subsequence, a substring is required to occupy consecutive positions within the original strings.

(a) Find the recursion that LCS(n, m) satisfies. Fully justify your answer.

(b) Identify the base cases for your recursion in part (a) and find their corresponding values. Justify your answer.

(c) Write the pseudo-code for the bottom-up DP algorithm to compute LCS(n, m). Find and justify the time complexity of your algorithm in the form of $\Theta(.)$.

Problem 4 (25 points)

Alice and Bob want to play the following game by alternating turns: They have access to a row of n coins of values $v_1, \dots, v_n$, where n is even. In each turn, a player selects either the first or the last coin from the row, removes it from the row, and receives the value of the coin. Alice starts the game.
Develop a dynamic programming algorithm to determine the maximum possible amount of money Alice can definitely win (assume that Bob will play in such a way to maximize the amount he gets). Find and justify the time complexity of your algorithm in the form of $\Theta(.)$.

Practice Problems

You do NOT need to submit the following problems. You can work on them for further practice.

Problem 1

Given a positive integer n, we can easily compute 2^n by using $\Theta(n)$ multiplications. Follow the following steps to develop an algorithm that computes 2^n using $\Theta(\log n)$ multiplications.

(a) Consider the case where n is a power of 2, i.e., $n = 2^k$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k} = \left(2^{2^{k-1}}\right)^2$.

(b) Consider the case where $n = 2^k + 2^{k-1}$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k+2^{k-1}} = 2^{2^k} \times 2^{2^{k-1}}$.

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

Bonus problems

The following problems are optional. They will NOT appear on any of the exams. Work on them only if you are interested. They will help you to develop problem solving skills for your future endeavors.

Bonus Problem 1

Develop an algorithm to compute the nth Fibonacci number in $O(\log n)$ time.

Bonus Problem 2

(a) Can you develop a bottom-up DP algorithm for Problem 1 with $O(n)$ space that runs in $O(n \log n)$ time in the worst case?
If so, write the pseudo-code of your algorithm.

(b) Can you improve the run-time of your algorithm in part (a) to $O(n)$?
If so, write the pseudo-code of your algorithm.