CS 331: Algorithms and Complexity (Fall 2016)
Unique numbers: 51420, 51425, 51430, 51435

Assignment 5

Due on Wednesday, October 19, by 11.59pm

Matt Gmitro 51430 MTG759

Problem 1

(10 points) You work for a contractor who builds toll roads. The government is offering to give you a portion of a road which is up to n miles long that must be contiguous. You are given a profit prediction model in the form of an array of length n which has profit values for each mile at each index.

1. (8 points) Using dynamic programming, write the pseudo code of an $O(n)$ iterative algorithm that returns the most profitable contiguous portion of road (it should output the indices range at which the profit is maximized).

   Solution:
   Let $N = n$ (the longest possible road) and $P[]$ be the profit prediction array
   Let M be an array of size N and initialize its values to 0.
   Let $mx = 0$ be a variable.
   Let $a = 0$,$b = 0$, and $temp_a = 0$ be start/end index variables.
   $Road(P[], N)$:
   $-> for(i = 1, ..., N)$
   $---> M[i] = max(P[i] + M[i-1], P[i])$
   $---> mx = max(mx, M[i])$
   $---> if(M[i] == mx)$
   $-------> b = i.$
   $-------> if(a\ != temp_a)$
   $-----------> a = temp_a.$
   $---> if(M[i] == P[i])$
   $-------> temp_a = i.$
   $-> return(a, b, mx)$

2. (2 points) Prove the correctness of your algorithm.

   Solution: 1.)The recurrence in this case is OPT(i) = max(P[i]+OPT(i-1), P[i]) where P is the profit . The sum at OPT(0) is zero. If P[i] increases the solution at i=1 then we add it and get a better solution because we are trying to maximize.

This proves the base case. For the inductive case, we assume OPT(k) is the optimal solution and want to prove this implies OPT(k+1) is optimal. If we add P[k+1] to OPT(k) we have two cases- either it is greater than or less than P[k+1]. If it is less than P[k+1], then the optimal solution at k+1 must be P[k+1] because this will be the maximum possible profit at point k+1 (since OPT(k) is already optimal for k and the subarray must be contiguous). If $P[k+1] + OPT(k) > P[k+1]$, then we know P[k+1] + OPT(k) is the maximum possible profit and we can include P[k+1] in the same contiguous array as OPT(k).
2.) The algorithm implements the recurrence nearly verbatim as we generate the optimal solution in each element of M by setting it to the max(P[i] + M[i-1], P[i]) and then checking whether this is the max of all possible iterations by setting it in mx and checking mx on each iteration. Furthermore we store the indices for each subarray and update them if a new subarray has a higher max.

As an example, if the array given to you was $A = [1, 2, -3, 4, -1, 2, -6]$, the most profitable contiguous segment of this road is $A[3, 4, 5] = [4, -1, 2]$ which yields a profit of 5.

Problem 2

(10 points) Brave Brian of the Boy scouts has an obstacle course that he has to traverse. The obstacle course has a single start location but multiple end locations. Brave Brian can choose to end at any of the available end locations. On observing the map of the obstacle course, Brian notices that the obstacle course is in fact a tree with the root being the start location, and the leaves being the end locations. At every node except the start node, Mean Mendes might have arranged for a bully to punch Brian a certain number of times. The map contains the locations of all such nodes, and the number of punches Brian would receive at each such node. Brian can either hop from a node to a child of the node or long-jump from a node, to a grandchild of the node. Brian is bamboozled by this problem. He wants to makes sure he gets punched as little as possible, but he is just not smart enough to figure out how to do that. So he asks his best friend, Smart Samantha, for help. Samantha suggests a Dynamic Programming Solution for the problem. What is a solution that Smart Samantha could have come up with?

1. (8 points) Using Dynamic programming, describe an iterative algorithm to solve this problem. There could be multiple solutions to this problem. Don't bother trying to find the same solution that Samantha would have come up with. Just come up a solution that she *could* have come with, i.e., a solution that works. Show the **time complexity** of your algorithm.

   Solution:
   Let n be the number of nodes in the graph.
   Let M[] be an array size n. Let mn=infinity be a variable
   Initialize M[v] for vertices in the first level of the tree to be current+root. Then for each vertex:
   $-- > M[v]$ = min(grandparent + current, parent + current)

$- - -- > if$ (v has no children and $M[v] < mn$):
$- - - - -- > mn = M[v]$
return mn.
Time complexity should be O(n) where n is the number of nodes in the graph because we iteratively traverse each node.

2. (2 points) Prove the correctness of your algorithm.

Solution: 1.) The recurrence is $OPT(i) = min(OPT(grandchild(i)+v_i), OPT(parent(i)+v_i))$. The base of the recurrence is the root node and its children which are initialized for the number of punches they receive. Then assuming OPT(k) and OPT(k-1) are optimal parent/grandparents, we need to show OPT(k+1) is optimal. In order to get to node k+1 we can either come from node k or node k-1 (since the structure is a tree and we can only jump or long jump). Thus we know the number of punches at k+1 will be punches(k+1) + punches(k) OR punches(k+1) + punches(k-1). By taking the minimum of these as in the recurrence we have the minimum number of punches at k+1. And if the node is a leaf (has no children) we check to see if it meets the minimum mn. Brian can then take the path to minimum leaf node using this recurrence.
2.) The recurrence is implemented by initializing the first couple nodes and then going through each node to find the minimum punches possible on that path. By looking at whether adding from a grandparent jump or a parent jump would be ideal.

Problem 3

(15 points) Assume that you have to make change for a value $v(v \geq 0)$, and that you have an infinite supply of coins of denominations $x_1, x_2, ..., x_n (n \geq 1)$ where each x_i is a positive integer between 1 and $v - 1$. We want to compute the *minimum* number of coins required to make the change. For example, if $v = 27$, and the coin denominations are 2,4,7,11,14, then we need at least 3 coins (1 each of 2, 11, and 14) to give the change. However, if $v = 27$ and the coin denominations are 2, 4, 7, 11, 15, then we need at least 4 coins (1 each of 2, 11 and 2 of 7) to give the change.

1. (10 points) Using dynamic programming, write the pseudo code of an iterative algorithm which solves the problem. Show the **time complexity** of your algorithm.

Solution: Let M[] be an array the size of the target value v and initialize M[0] to 0. For all other values of M[i to v] initialize to infinity.
Let C[] be the array of coin values size n for the number of coin values n.
$for(i = 1, ..., v)$
$-- > for(j = 0, ..., n)$
$- - -- > if(C[j] <= i)$
$- - - - -- > M[i] = min(M[i - C[j]] + 1, M[i])$
$returnM[v]$
Time complexity is O(v * n) where v is the target value and n is the number of coin

values.

2. (5 points) Prove the correctness of your algorithm. **Solution:** 1.) The recurrence is OPT(v) = min for all coin values(min(OPT(v - C[j])+1, OPT(v))) where C is the array of coins. The base of this is v = 2 with a coin value 1 and a coin value 2. The OPT(v) for the coin value 1 is two coins while the value for a coin value 2 is one coin. Since we check with both coin values, we know Assuming we have some OPT(k) we must prove OPT(k+1). For the k+1 problem we will either have some OPT(k - c) for a better coin choice or use a OPT(k+1) defined by other coins. We can assume by induction that OPT(k-c) is optimal, and we know OPT(k+1) is initialized to infinity. So we take the OPT(k-c) path and add 1 to it which is clearly less than infinity. Then if another coin value has a better performance we take that minimum, eventually arriving at the best solution. 2.) The algorithm implements this by maintaining an array which is built on the smallest coin values first and then checks each possible coin value as a subtraction when solving the larger subproblem.