CSCI 270: Homework 6

1. From the lecture, you know how to use dynamic programming to solve the 0-1 knapsack problem where each item is unique and only one of each kind is available. Now let us consider the knapsack problem where you have infinitely many items of each kind. Namely, there are n different types of items. All the items of the same type i have equal size wi and value vi. You are offered with infinitely many items of each type. Design a dynamic programming algorithm to compute the optimal value you can get from a knapsack with capacity W.

   Solution:

   a. Define (in plain English) subproblems to be solved. (4 pts):

      Let $OPT(k, w)$ be the maximum value achievable using a knapsack of capacity $0 \leq w \leq W$ and with k types of items $1 \leq k \leq n$.

   b. Write a recurrence relation for the subproblems (6 pts):

      Since we have infinitely many items of each type, we choose between the following two cases:
      - We include another item of type k and solve the sub-problem $OPT(k, w - w_k)$.
      - We do not include any item of type k and move to consider the next type of item thus solving the sub-problem $OPT(k - 1, w)$.

      Therefore, we have:

      $$OPT(k, w) = max\{OPT(k - 1, w), OPT(k, w - w_k) + v_k\}$$

   c. Make sure you specify
      a. base cases and their values (2 pts)

         $$OPT(0, 0) = 0$$

      b. where the final answer can be found (1 pt):

         $$OPT(n, W)$$

   d. What is the complexity of your solution? (2 pts)

      $$O(n \, x \, W)$$

2. Solve Kleinberg and Tardos, Chapter 6, Exercise 12.

   Solution:

   a. Define (in plain English) subproblems to be solved. (4 pts)

      $OPT(j)$ be the minimum cost of a solution on servers 1 through *j*, given that we place a copy of the file at server *j*.

   b. Write a recurrence relation for the subproblems (6 pts)

      We want to search over the possible places to put the highest copy of the file before j; say in the optimal solution this is at position *i*. Then the cost for all servers up to i is *OPT(i)* (since we behave optimally up to *i*), and the cost for servers i + 1,...,j is the sum of the access costs for i + 1 through j, which is:

      $$0 + 1 + .. + j - i - 1 = \sum_{x=0}^{j-i-1} x$$

      $$OPT(j) = c_j + \text{minimum over } 0 \leq i < j \text{ of : } \left(OPT(i) + \sum_{x=0}^{j-i-1} x \right)$$

   c. Make sure you specify

      i. base cases and their values (2 pts)

         $$OPT(0) = 0$$

      ii. where the final answer can be found (1 pt):

         $OPT(n)$ as we always have a copy at S_n

   d. What is the complexity of your solution? (2 pts)

      The complexity is $O(n^2)$. As for each time we calculate OPT(j), we have to iterate through 0 to *j*-1 to find the optimal placements till *j*. We don't need to calculate the summation equation in every iteration as we can keep track of the traversal costs in the previous iteration and just add j-i-1 to it.

3. Given n balloons, indexed from 0 to $n - 1$. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get $nums[left] * nums[i] * nums[right]$ coins. Here $left$ and $right$ are adjacent indices of i. After bursting the balloon, the $left$ and $right$ then

becomes adjacent. You may assume $nums[-1] = nums[n] = 1$ and they are not real therefore you can not burst them. Design a dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely. Analyze the running time of your algorithm.

Here is an example. If you have the nums arrays equals $[3, 1, 5, 8]$. The optimal solution would be 167, where you burst balloons in the order of 1, 5, 3 and 8. The left balloons after each step is:

$$[3, 1, 5, 8] \rightarrow [3, 5, 8] \rightarrow [3, 8] \rightarrow [8] \rightarrow []$$

And the coins you get equals:

$$(3 * 1 * 5) + (3 * 5 * 8) + (1 * 3 * 8) + (1 * 8 * 1) = 167$$

Solution:

a. Define (in plain English) subproblems to be solved. (4 pts)

Let $OPT(l, r)$ be the maximum number of coins you can obtain from balloons $l, l+1, \ldots, r-1, r$

b. Write a recurrence relation for the subproblems (6 pts)

The key observation is that to obtain the optimal number of coins for balloon from l to r, we choose which balloon is the last one to burst. Assume that balloon k is the last one you burst, then you must first burst all balloons from l to $k-1$ and all the balloons from $k+1$ to r which are two sub problems. Therefore, we have the following recurrence relation:

$$OPT(l, r) = max_{l \le k \le r}\{OPT(l, k-1) + OPT(k+1, r) + nums[k] * nums[l-1] * nums[r+1]\}$$

c. Make sure you specify
   i. base cases and their values (2 pts)

   $$OPT(l, r) = 0 \quad if\ r < l$$

   ii. where the final answer can be found (1 pt):

   $$OPT(0, n-1)$$

d. What is the complexity of your solution? (2 pts)

For running time analysis, we in total have $O(n^2)$ and computation of each state takes $O(n)$ time. Therefore, the total time is $O(n^3)$.

4. Suppose you have a rod of length N, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth p_i dollars. Devise a Dynamic Programming algorithm to determine the maximum amount of money you can get by cutting the rod strategically and selling the cut pieces.

   Solution:

   a. Define (in plain English) subproblems to be solved. (4 pts)

      Let $OPT(l)$ be the maximum amount of money you can get from cutting the rod of length l

   b. Write a recurrence relation for the subproblems (6 pts)

      At each remaining length of the rod, we can choose to cut the rod at any point, and obtain points for one of the cut pieces and recursively compute the maximum points we can get for the other piece.

      $$OPT(l) = max_{1 \le i \le l}\{p[i] + OPT[l - i]\}$$

   c. Using the recurrence formula in part b, write pseudocode to find the subarray (containing at least one number) which has the largest sum. (5 pts)

      Let $OPT[0, \ldots, N]$ be a new array
      $OPT[0] = 0$
      for l = 1 to N
          q = -∞
          for i = 1 to l
              q = max(q, $p[i] + OPT[l - i]$)
          $OPT[l]$ = q
      return $OPT[N]$

   d. Make sure you specify
      i. base cases and their values (2 pts)

$OPT(0) = 0$

ii. where the final answer can be found (1 pt):

$OPT(N)$

e. What is the complexity of your solution? (2 pts)
$\Theta(n^2)$

5. Solve Kleinberg and Tardos, Chapter 6, Exercise 10.

Solution:

Part (a) (4 pts):

	Minute 1	Minute 2
A	2	10
B	1	20

The algorithm would choose A for both steps but the optimal solution is to choose B for both steps

Part (b):

a. Define (in plain English) subproblems to be solved. (4 pts)

let $OPT_A(i)$ represent the maximum value of a plan in minute 1 through i that ends on machine A, and define $OPT_B(i)$ analogously for B.

b. Write a recurrence relation for the subproblems (6 pts)

For the time interval from minute 1 to minute i, consider that if you are on machine A in minute i, you either (i) stay on machine A in minute i − 1 or (ii) are in the process of moving from machine B to A in minute i − 1. If case (i) is the best action to make for minute i-1, we have $OPT_A(i) = a_i + OPT_A(i-1)$; otherwise, we have $OPT_A(i) = a_i + OPT_B(i-2)$. In sum, we have:

$$OPT_A(i) = a_i + \max\{OPT_A(i-1), OPT_B(i-2)\}$$

Similarly, we get the recursive relation for OP TB(i):

$$OPT_B(i) = b_i + \max\{OPT_B(i-1), OPT_A(i-2)\}$$

c. Using the recurrence formula in part b, write pseudocode to find the subarray (containing at least one number) which has the largest sum. (5 pts)

$OPT_A(0) = 0$; $OPT_B(0) = 0$;
$OPT_A(1) = a_1$; $OPT_B(1) = b_1$;

For i = 2, .. , n do:

$OPT_A(i) = a_i + \max\{OPT_A(i-1), OPT_B(i-2)\}$
Record the action (either stay or move) in minute i-1 that achieves the maximum.

$OPT_B(i) = b_i + \max\{OPT_B(i-1), OPT_A(i-2)\}$

Record the action (either stay or move) in minute i-1 that achieves the maximum.

End for

Return $\max\{OPT_A(n), OPT_B(n)\}$

Track back through the arrays OPT_A and OPT_B by checking the action records from minute n-1 to minute 1 to recover the optimal solution

d. Make sure you specify
   i. base cases and their values (2 pts)

$OPT_A(0) = 0$; $OPT_B(0) = 0$;
$OPT_A(1) = a_1$; $OPT_B(1) = b_1$;

   ii. where the final answer can be found (1 pt):

The solution can be calculated as: $\max\{OPT_A(n), OPT_B(n)\}$

e. What is the complexity of your solution? (2 pts)

O(n) as all the operations within the for loop are constant time.