Homework 7 Solutions

For each of the questions below, design a dynamic programming based algorithm by answering the following sub-parts:

a) Define in plain English the **subproblems** that your DP algorithm solves. (4 points)
b) Define a recurrence relation that expresses the value of each subproblem in terms of the smaller subproblems. (6 points)
c) Using the recurrence formula in part b, write pseudocode to find the solution.
Make sure you specify all the base cases and their values, as well as the final answer to the problem. (6 points)
d) What is the time complexity of the solution? Further, determine whether it is an efficient solution or not. (2+2 points)

1. Suppose we are 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 you burst balloon *i*, you will get *nums[left] * nums[i] * nums[right]* coins. Here, *left* and *right* are balloons adjacent to *i* **at the time of bursting**. After bursting any balloon, **its left and right then become adjacent** (thus, there aren't any "gaps" at any point, see example below). Assume *nums[−1] = nums[n] = 1* when determining the coins obtained by bursting the corner balloons, (-1 and n are not indices of real balloons, therefore you can not burst them). Design a dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons. (20 points)

   Here is an example. Suppose you have the *nums* arrays [3, 1, 5, 8]. The optimal solution would yield 167, where you burst balloons in the order of 1, 5, 3 and 8. The balloons left after each step are: [3, **1**, 5, 8] → [3, **5**, 8] → [**3**, 8] → [**8**] → []
   And the coins you get in each step are:
   (3 * 1 * 5) + (3 * 5 * 8) + (1 * 3 * 8) + (1 * 8 * 1) = 15+120+24+8 = 167

Solution:

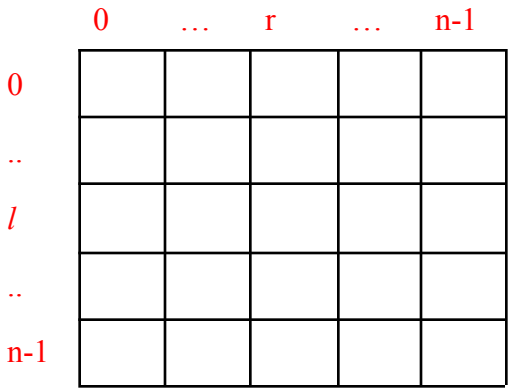
a) Let *OPT*(*l, r*) be the maximum number of coins you can obtain from balloons *l, l+1, . . . , r−1, r* ("while the balloons l-1 and r+1 are intact" to be more precise).

b) The key observation is that to obtain the optimal number of coins for a balloon from *l* to *r*, we choose 'which balloon is the last one to burst' from that interval - this ensures that for any subproblem being solved, the balloons at 'l-1' and 'r+1' are intact and allow to recursively define the subsequent subproblems correctly.

Assume that balloon k is the last one you burst, then you must first burst all balloons from l to k-1 as well as 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) Note that when k = l or r (end of the interval), the new subproblem(s) would be an empty interval - so, we take care of this with the base cases $OPT(l, r) = 0$ if $r < l$ (r=l-1 in particular).
To determine the computation order, imagine the subproblems arranged in a grid, with rows indexed by l, columns by r, and top left corner being (0,0).



Note that a subproblem (l,r) requires that subproblems to the left in the same row (l,k-1) and the ones below in the same column (k+1,r) should already be computed - hence, we must compute the subproblems in the order of increasing r & decreasing l.
Thus, our pseudocode implementation is as follows:

```
For l = 1 to n-1:
        Opt[l][l-1] = 0          // Base cases

for l = n-1 to 0:
  for r = l to n-1:
    bestk = -∞
    for k = l to r:        //Compute for each k while updating the best k
        bestk = max (bestk, OPT[l][k-1] + OPT[k+1][r] + nums[k]*nums[l-1]*nums[r+1]
    OPT(l,r) = bestk

Return Opt[0][n-1]
```

d) Running time analysis: we have $\Theta(n^2)$ subproblems (two outer loops) and computation of each takes $\Theta(n)$ time (innermost loop). Therefore, the total time is $\Theta(n^3)$. Thus, this is a poly-time (efficient) solution.

2. Tommy and Bruiny are playing a turn-based game. It involves N marbles placed in a row. Each marble i has a positive value m_i. On each player's turn, they can remove either the leftmost marble or the rightmost marble from the row and receive a score given by the value of that marble. The players' goal is to beat the opponent's score by the highest margin possible (equivalently, minimizing the losing margin in the case of the player with the lesser score).

   Tommy goes first in this game. Devise a Dynamic Programming algorithm to return the maximum difference in score that Tommy can achieve over Bruiny, assuming both players are playing optimally. Your algorithm must run in $O(N^2)$ time.

   (**Ungraded** Follow-up: Suppose the score received when removing a marble is changed to 'the sum of the remaining marbles' values left in the row.' How will you modify your solution **without affecting** the runtime complexity?)

Solution

a) Define $OPT(i, j)$ as the maximum difference in score achievable by the player whose turn it is to play, when the marbles from index i to j (inclusive) remain.

b) $OPT(i,j) = \max\{m_i - OPT(i+1,j)\ ,\ m_j - OPT(i,j-1)\}$

   Explanation: Suppose this is player X's turn, with the next turn being player Y (X and Y can be Tommy and Bruiny or vice versa). The first term of the recurrence corresponds to X making the choice of taking marble i (left end) and receiving m_i points, which leads to Y being the first player with the remaining marbles (i+1,j) - which allows Y to get a difference of OPT(i+1,j) over X from thereon **as per the subproblem definition**. Thus, the difference X can get in the current turn with this choice (marble i) is '$m_i - OPT(i+1,j)$'. The second term similarly captures the difference obtainable by choosing j (right end). X must choose the better of these two choices, which gives us the recurrence.

c) Note that when i = j, the new subproblems in the recurrence are empty intervals - we need to cover these as base cases. Thus, Opt(i,j) = 0 when j < i (j = i-1 in particular) since one can only get zero difference with no marbles left. (Alternately, can also make i=j as the base case instead with $OPT(i,i) = m_i$)

   Similar to the analysis in Q1, the computation order needs to have increasing j and decreasing i. This gives us the following pseudo-code:

```
For i = 1 to n-1:
        Opt[i][i-1] = 0          // Base cases
for i = n - 1 to 0:
    for j = i to n - 1:
```

ifTakei = m_i - OPT[i+1][j]
ifTakej = m_j - OPT[i][j-1]
OPT[i][j] = max(ifTakei, ifTakej) //recurrence terms computed separately for readability

return OPT[0][n-1]

d) The time complexity of this proposed algorithm is $\Theta(n^2)$ as there are $\Theta(n^2)$ subproblems (both loops) and each subproblem costs $\Theta(1)$ to compute.

For ungraded follow-up: We replace the reward in the original question when taking marble i with sum[i+1,..j] instead of m_i and similarly sum[i,..j-1] instead of m_j, thus, we need a way of computing the sum of marbles from any i to j without adding $\Theta(n)$ overhead in each recurrence step, to maintain the runtime complexity.

We first calculate a "prefix sum" for the marbles array. This enables us to find the sum of a continuous range of values in $O(1)$ time. If we have an array like this: [5, 3, 1, 4, 2], then our prefix sum array would be [0, 5, 8, 9, 13, 15]. This prefix sum array is calculated by simply iterating over the array (can be thought of as DP given by prefixsum[i] = prefixsum[i-1] + m_i), thus taking $\Theta(n)$ time, but only once in the beginning. Now, we can write any sum[i,..j] = prefixsum[j] - prefixsum[i-1]. With this, both the sums needed in our recurrence are computed simply in $\Theta(1)$ time using Prefixsum.

3. The Trojan Band consisting of n band members hurries to line up in a straight line to start a march. Due to the height differences, the line is looking very messy. The band leader decides to pull out a few band members so that the line *follows a formation* with the remaining band members staying where they are. We say that k members remaining in the line with heights $r_1, r_2, \ldots, r_k$ are *in formation* if $r_1 < r_2 < \ldots < r_i > \ldots > r_k$, for some $1 \leq i \leq k$.

   For example, if the initial sequence of heights in inches is (67, 65, 72, 75, 73, 70, 70, 68),

   then, pulling out member #2 and #6 gives us the formation: (67, 72, **75**, 73, 70, 68).

   Give an algorithm using Dynamic Programming that runs in $O(n^2)$ time to find the minimum number of band members to pull out of the line, so that we are left with a *formation* as described.

Solution

a) Let OPTleft(i) = max length of the increasing subsequence ending at i (including i)

   & OPTright(i) = max length of the decreasing subsequence starting at i (including i)

Note that the two sub-problem are symmetric, in that, we can flip the input array R and compute the values OPTleft on it, which would give us the OPTright values of the original unflipped input

b) The recurrence relations are:

   OPTleft(i) = 1 + max{OPTleft(j)} such that $r_i > r_j \ \forall \ 1 \leq j < i$

   If none of the j's $i < j \leq n$ satisfies $r_i > r_j$, OPTleft(i) = 1

   Similarly, OPTright(i) = 1 + max{OPTright(j)} such that $r_i > r_j \ \forall \ i < j \leq n$

   If none of the j's $(i < j \leq n)$ satisfies $r_i > r_j$, OPTright(i) = 1

c) We do not need explicit base cases, since the condition (2nd line) in each case accounts for subproblems which have no valid subproblems to call. The pseudocode is as follows:

```
for i = 1 to n:
    bestJ = 0          //0 if no valid j is found in the loop below
    for j = 1 to i - 1:          //Compute for each j while updating the best j
        if (r[i] > r[j]): bestJ = max(bestJ, OPTleft[j])
    OPTleft[i] = 1 + bestJ
```

// As noted previously, can repeat the computations above on the flipped input to get the OPTright values, or compute separately as below:

```
for i = n to 1:
        bestJ = 0          //0 if no valid j is found in the loop below
        for j = n to i + 1:          //Compute for each j while updating the best j
                if (r[i] > r[j]): bestJ = max(bestJ, OPTright[j])
        OPTright[i] = 1 + bestJ
// The final answer comes from finding the best combination:
bestComb = -∞
for i = 1 to n:
    bestComb = max(bestComb, OPTleft[i] + OPTright[i] - 1)   // `-1' since 'i' is double-counted otherwise
return (n - bestComb)            //Since the question asks the min. no. of people to remove
```

d) The runtime of this algorithm is dominated by the computation of OPTleft and OPTright which have the nested for loops taking $\Theta(n^2)$ time, Thus, it is polynomial time (efficient).

Ungraded Problems

Note: The solutions to the problems below mention the key components required to expand into the full pseudo-code which is expected for consistency as in the solutions above.

1. You are in Downtown of a city where all the streets are one-way streets. At any point, you may go right one block, down one block, or diagonally down and right one block. However, at each city block (i, j) you have to pay the entrance fees fee(i, j). The fees are shown in a grid below:

	0	1	2	3	⋯	n
0	$fee_{(0,0)}$	$fee_{(0,1)}$	$fee_{(0,2)}$	$fee_{(0,3)}$	⋯	$fee_{(0,n)}$
1	$fee_{(1,0)}$	$fee_{(1,1)}$	$fee_{(1,2)}$	$fee_{(1,3)}$	⋯	$fee_{(1,n)}$
2	$fee_{(2,0)}$	$fee_{(2,1)}$	$fee_{(2,2)}$	$fee_{(2,3)}$	⋯	$fee_{(2,n)}$
3	$fee_{(3,0)}$	$fee_{(3,1)}$	$fee_{(3,2)}$	$fee_{(3,3)}$	⋯	$fee_{(3,n)}$
⋮	⋮	⋮	⋮	⋮	⋱	⋮
n	$fee_{(n,0)}$	$fee_{(n,1)}$	$fee_{(n,2)}$	$fee_{(n,3)}$	⋯	$fee_{(n,n)}$

Your objective is to travel from the starting point at the city's entrance, located at block (0,0), to a specific destination block (n,n) (paying a fee at both these blocks as well).
You would like to get to your destination with the least possible cost.

Formulate the solution to this problem using dynamic programming.

Solution:
Subproblems: Let OPT(i, j) be the minimum cost to get from (0,0) to (i, j).

Recurrence: OPT (i, j) = fee(i, j) + min (OPT (i − 1, j), OPT (i, j − 1), OPT (i − 1, j − 1))
The 'out-of-bounds' terms are ignored during the pseudo-code.

Base cases: OPT(0,0) = fee(0,0)
Computation order: increasing i and j
Final answer: OPT(i,j)

Runtime: $\Theta(n^2)$, poly-time solution (efficient).

2. Suppose we have N workers to be assigned to work at one of M factories. For each of the M factories, they will produce a different profit depending on how many workers are assigned to that factory. We will denote the profits of factory i with j workers by P(i,j). Develop a dynamic programming solution to find the maximum profit possible by assigning workers to factories.

Solution:

Subproblems: B(m,j) denotes the maximum profit possible by assigning j workers to first m factories.

Recurrence: $B(m,j) = \max_{0 \le k \le j} \{B(m-1, j-k) + P(m,k)\}$
Each term in the recurrence considers assigning k workers to the m^{th} factory, varying k from 0 to j

Base case: $B(1,j) = P(1,j)$ for all $0 \le j \le N$
Computation order: Increasing m and j
Final answer: B(M,N)

Runtime: $\Theta(MN^2)$, poly-time (efficient)

3. You have two rooms to rent out for a period of D days. There are n customers interested in renting the rooms. The ith customer wishes to rent one room (either room you have) for d[i] days and is willing to pay bid[i] for the entire stay. Customer requests are non-negotiable in that they would not be willing to rent for a shorter or longer duration, but they are indifferent to which days they get room for out of the D days. Devise a dynamic programming algorithm to determine the maximum profit that you can make from the customers over a period of D days.

a) Define (in plain English) subproblems to be solved. (4 points)
b) Write the recurrence relation for subproblems. (6 points)

Solution:
Subproblem: Let OPT(d1, d2, i) be the maximum profit obtainable with d1 remaining days for room 1 and d2 remaining days for room 2 using the first i customers.

Recurrence: $OPT(d1, d2, i) = \max\{bid[i] + OPT(d1 - d[i], d2, i - 1), bid[i] + OPT(d1, d2 - d[i], i - 1), OPT(d1, d2, i - 1)\}$
The 3 terms correspond to the choices of accommodating customer i in room 1, room 2 or not at all.

Base cases: $OPT(d1,d2,0) = 0$
Pseudo-code checks for whether $d1 < d[i]$ and $d2 < d[i]$ and ignores the corresponding recurrence terms.

Runtime: $\Theta(D^2n)$. Pseudo-polynomial time (Not efficient)

4. Solve Q4 from exam 1 (non-consecutive subsequence) using Dynamic Programming.

Solution:

Subproblems: Longest non-consecutive subsequence from the first i elements

Recurrence: OPT(i) := max{OPT(i-2)+1, OPT(i-1])}
The two terms correspond to choices of including or excluding element i.

Base cases: = Opt[0] = 0, Opt[1] = 1
Computation order: Increasing i from 2 to n
Final answer: Opt[n]

Runtime: O(n), poly-time (efficient)