CSE321 – Introduction to Algorithm Design

Homework 5 – Report

Yasir Nacak – 161044042

Q1.

In this part, I created a 2 by N+1 (number of months + 1) DP table. The two rows in this table represents the paths starting with New York and San Francisco, respectively. I initialize my first column of my DP table as 0. Then, I created a single for-loop to fill my table. In this loop, I start with 1st index, and for each element of the DP table, I check each of my rows and set the current DP table variable as the following:

1. Set DP[NY][i] as NY_COST[i - 1] + minimum of DP[NY][i - 1] and DP[SF][1 - 1] + M
2. Set DP[SF][i] as SF_COST[i - 1] + minimum of DP[SF][i - 1] and DP[NY][1 - 1] + M

After filling the both rows of the DP table, I return the minimum of the last columns of the table. The filling step basically looks for the two possible options. One being the continuing with the same path as before and the other checks for the cost of the moving and the other side's operation cost so far. And it chooses the minimum all the time, so the solution is optimized all the time. After filling the entire table, the last column gives us the final costs after all the optimizations in the both paths and the minimum of those is the optimum of the two possible paths. Since this algorithm fills up the entire table in one run of N times, worst-case running time of this algorithm is O(N).

Q2.

We have discussed this problem in one of our Greedy Algorithms lectures as the Interval Scheduling Problem. The solution that has the mathematical proof and gives always the optimal solution is to always select the session that finishes first among the all sessions. With doing this, we also need to eliminate any sessions that overlap with the current selected session. In my algorithm, I first sorted the sessions with respect to their finish time. After that, I initialized my list of selected sessions with the first session in the sorted list. From that point on, I iterated through all the sessions and checked if they overlap with the current selected session. If so, I continued iterating without including that session. If not, I picked that session and added it to my list of selected sessions and updated my current session. Since I sorted my list beforehand, this solution guaranteed me that whenever I came across a session that I can select, it is the one with that finishes first. Worst-case running time of my algorithm would be O(N) if the session list was already sorted but since the question does not give that as a guarantee, we need to sort it and sorting takes O(N * Log(N)) time so that is our worst-case.

Q3.

For this question, instead of a bottom-up approach that is generally used in the subset-sum problem, I utilized memoization which we have seen in the calculation of Fibonacci numbers. This method is used whenever there are multiples of the same operations that needs to be calculated repeatedly. In those situations, saving previously calculated variables in a table comes in handy. This problem of finding a subset with a specific sum can be represented as the 0-1 Knapsack Problem. We are still going to construct a recursive structure where we either include a number or not which creates two branches to the call tree. But adding an operation that saves the element and the sum with its inclusion to a table, we don't need to calculate it repeatedly and can just check the memo table for a quick look-up. The worst-case running time of this algorithm for any S = sum value is O(N * S).

Q4.

For this question, I created a 2D DP table with width being the length of the Sequence A + 1 and the height being the length of the Sequence B + 1. I first initialized the first row with the values Gap Score * Column Index. I then initialized the first column with the values Gap Score * Row Index. The movement from the starting cell with another cell can occur in three ways. They are, arriving from the top cell, arriving from the left cell and arriving from the top left cell. The first two cases represent the situations where we need to put a gap to Sequence A and Sequence B, respectively. The third case is the case where we don't put a gap and there occurs a match or a mismatch. The first two case explain the initial values of the DP table. To fill the rest of the table, we start from the top-left uninitialized element and go all the way down to bottom right element. At each step of the iteration, we check for the minimum of three values. They are:

1. Put a gap to Sequence A and add Gap Score to the Current Score.
2. Put a gap to Sequence B and add Gap Score to the Current Score.
3. Don't put any gaps and depending on the current Ith and Jth values of Sequence A and Sequence B, add Match or Mismatch score to the Current Score.

From those options, we choose the one that adds the most to the Current Score and set it as the DP[i][j] value of the table. After filling the entire table, we return the DP[height][width] element as the total maximum score. Note that this algorithm counts the gaps after one of the sequences finishes. So for the example given in the homework document, after the match with the word "SLIME" ends, we still need to put two gaps to finish the both sequences and thus, my algorithm gives the answer 2 to that example.

Let's call the length of Sequence A, N. And let's call the length of Sequence B, M. Since the DP table we constructed has the dimensions N and M and we do one iteration through the table to fill all the elements, the worst-case running time of this algorithm is O(N * M).

Q5.

For this question, I designed an algorithm that always chooses the smallest number currently available because it gives the least operation cost. To achieve that, I sorted the array in the ascending order first. I then summed up the first two elements of the list and added this to the result. After that, I added rest of the list and the summed value to a new list. I then sorted this new list and did this repeatedly until there are only two elements remain. I then returned the sum of the current result and sum of the first two elements. An example:

S = [14, 2, 11, 5, 7]

I first sort this list to get:

S = [2, 5, 7, 11, 14]

After that, I include 2 and 5 in my sum as:

2 + 5 = **7 (Initial Operation Cost)**

From that point on, iterate the rest of my list and add the numbers one by one while adding the newly created sum to my operation cost.

For i = 2 (First Iteration):

S[i] = 7, Current Operation Cost = 7

7 + 7 = **14 (Add this to the operation cost)**

For i = 3:

S[i] = 11, Current Operation Cost = 21

11 + 14 = **25 (Add this to the operation cost)**

For i = 4:

S[i] = 14, Current Operation Cost = 46

14 + 25 = **39 (Add this to the operation cost)**

Total operation cost = 85

The worst-case running time of this algorithm also works the same way as the Question 2. I sort the array at every iteration and iterate through it. The running time of the iteration is O(N) and the running time of the sort operation is O(N * Log(N)). The worse of those two is generally O(N * N * Log(N)) so the worst-case running time of this greedy algorithm is $O(N^2Log(N))$.