CSE Algorithms – Assignment 2

Bhanu Verma – 903151012
Collaborator – Shruti Bhati

Question 1 – Greedy, Great Ice Cream Scale

We have various flavors $s_1, s_2, s_3 \dots \dots \dots, s_n$ and each of these flavors have a remaining amount given by $d_1, d_2, d_3 \dots \dots \dots, d_n$, our task is to maximize the Likeability score defined as L, it is calculated in accordance with few rules which are listed below:

- We are given a bucket of one gallon.
- L is calculated as sum of each individual ice cream's score, if we are able to take the full quantity of the remaining flavor, we get full score s_i for that flavor, but if take only a fraction of a particular flavor L will calculated $s_i * (c/d_i)$, where c is the remaining capacity of my one gallon bucket.

Our Greedy Claim – If we choose flavors in the increasing order of d/s, we can optimize our likeability score L.

Proof – We will use proof by contradiction to show that our greedy algorithm is the optimal algorithm for maximizing L.

Let's say we have three flavors to choose from, their amounts are d_1, d_2, d_3 and their respective scores are s_1, s_2, s_3. Let's say for the given case we have,

$\frac{d_3}{s_3} < \frac{d_2}{s_2} < \frac{d_1}{s_1}$, ------ 1

$d_1 + d_2 + d_3 > 1\ gallon$ ------ 2

$d_1, d_2, d_3 < 1$ (this means that each amount is less than 1 gallon)

As per our greedy claim let's say, we pick the **third flavor** and get full score for that flavor. At this stage,

$L = s_3$

After picking flavor 1, let's say we have amount r remaining in our bucket. Then, using equation 2 we can say that:

$d_1 + d_2 > r$ or $d_1 + d_2 = r + e$ where e is the extra amount ----- 3

Now for the next step, our greedy algorithm will pick **second flavor** and then first flavor and let's say there exists an optimal algorithm and it picks the **first flavor** and then second flavor. Now let's look at the total likeability score for both algorithms.

Our Algorithm

$L = s_3 + s_2 + s_1 * (d_1 - e)/d_1$

$L = s_3 + s_2 + s_1 - s_1 * (\frac{e}{d_1})$

Optimal Algorithm

$$L = s_3 + s_1 + s_2 * (d_2 - e)/d_2$$

$$L = s_3 + s_1 + s_2 - s_2 * (\frac{e}{d_2})$$

Now, L for optimal algorithm should be more than or equal to L for our algorithm. Let's put this thought into a mathematical equation:

$$s_3 + s_2 + s_1 - s_1 * (\frac{e}{d_1}) \leq s_3 + s_1 + s_2 - s_2 * (\frac{e}{d_2})$$

$$-s_1 * (\frac{e}{d_1}) \leq -s_2 * (\frac{e}{d_2})$$

$$\frac{s_1}{d_1} \geq \frac{s_2}{d_2}$$

$$\frac{d_2}{s_2} \geq \frac{d_1}{s_1}$$

But this contradicts Equation 1. Hence, it is proved that our greedy algorithm is as good as any other optimal algorithm and hence is the optimal solution for this problem. Please note that time complexity for this algorithm is $O(n * \log n)$ as we are using sorting and not performing any step which requires significant computation.

Question 2 – Dynamic Programming – George Learns Algorithms

Part 1 – Designing an optimal substructure for the problem

Let's say we have a sequence S $(s_1, s_2, \dots\dots\dots, s_n)$ which contains the sequence of scores that George received. Now, let L $(l_1, l_2, \dots\dots\dots, l_n)$ be any optimal solution defined such that each index i in L contains the length of the longest sequence ending with s_i.

Now what we need to do is iterate for all elements smaller than n and check for all values where $s_n > s_j$ and $n > j > 0$ and get the maximum values out of all those values and add one for getting the final answer for l_i . This will ensure that at each index i in L we will have the length of longest sequence ending with s_i.

This gives us our general algorithm idea which we will use for building up our recurrence relation.

Part 2 – Recurrence relation for this problem

$l_i = max(l_{j-1}) + 1$, where $s_j < s_i$ and $0 < j < i$

Base case is $l_0 = 0$

Part 3 – Bottom up DP

General idea here is to initialize L with 1 and then move from left to right in S, for each element in S i.e. s_i , we compare every element on the left of S say s_j with the s_i and if $s_i > s_j$, we replace l_i with $l_j + 1$. We keep on updating l_i for every element on the left of s_i if $s_i > s_j$. For this approach, while moving right, we will always have what we need on the left hand side. Hence, it is a valid Bottom up DP algorithm.

Time complexity – For each element in S, we can have maximum of $n - 1$ comparisons and there are n elements in S. Hence our time complexity becomes,

Time Complexity - $O(n^2)$

Space complexity – We need a new array L of size n. Hence our space complexity is

Space Complexity - $O(n)$

This proves to George that out algorithm works better than an exhaustive search approach.

Question 3 – Dynamic Programming, most mysterious mansion

Part 1 – Proving optimal substructures

If we observe this problem carefully, this looks like a knapsack problem but with a minor variation that when both the players chose to evade fighting the monster then instead of no accumulation of weight (stress for this variation), stress for that monster gets evenly distributed. Now, let's try to look at optimal substructure for the given mansion problem.

In order to achieve an optimal solution, there can be following cases for each occurrence of monster:

- If the accumulated stress for both the players becomes more than the maximum allowed stress when you add half the amount of stress for that monster to either player, then we trace back and look at the last monster with no change in accumulated stress.
- If the accumulated stress for any of the players becomes more than maximum allowed stress on fighting a monster, then we trace back and again look at the last monster.
- If the accumulated stress for both the players is less than or equal to the maximum allowed stress when you add half the amount of stress for that monster to either player, then both players evade that monster (i.e. add the half of stress for that monster to each player's accumulated stress and reward for that monster to total earned reward) and trace back to the last monster.
- If the accumulated stress for player 1 is less than or equal to the maximum allowed stress when you add the stress for that monster to player 1, then player 1 fights that monster and accumulates the stress from that monster and earns the reward from that monster for whole team.
- If the accumulated stress for player 2 is less than or equal to the maximum allowed stress when you add the stress for that monster to player 2, then player 2 fights that monster and accumulates the stress from that monster and earns the reward from that monster for whole team.

Please note that the order of these rules matter for algorithm to work correctly for sub-problems. These rules should be used in the same order as they are mentioned. These rules will help us to build our recurrence solution.

Part 2 – Building up recurrence relation

Here is the recurrence relation built from using above rules:

Case 1 – None can't fight, Can't evade

OPT(i, s1,s2) = OPT(i-1, s1, s2) where s1 + si > S **or** s2 + si > S || s1 + si/2 > S **and** s2 + si/2 > S

Case 2 – Evasive Action

OPT(i, s1,s2) = OPT(i-1, s1+si/2, s2+si/2) where s1+si/2 < S **and** s2+si/2 < S

Case 3 – A fights

OPT(i,s1,s2) = OPT(i-1, s1+si, s2) + ri where s1+si < S and s2+si < S

Case 4 – B fights

OPT(i,s1,s2) = OPT(i-1,s1,s2+si) where s1+si < S and s2+si < S

For case 1, we get the maximum reward for last monster and return that as answer.
For case2, case3 and case 4, we take the maximum values all three cases and return that as answer.

Part 3 – Top down algorithm with memoization

```
def optimizeReward(i, s1, s2, m, r, dict, S):
    if i < 0:
      return 0

    # solution exists, MEMOIZATION
    if dict.get((i, s1, s2)) != None:
      return dict.get((i, s1, s2))

    if (s1 + m[i] > S or s2 + m[i] > S) or (s1 + m[i]/2 > S and s2 + m[i]/2 > S):
    # cannot fight, go to last monster
      dict[(i, s1, s2)] = optimizeReward(i − 1, s1, s2, m, r, dict, S)

    if s1 + m[i]/2 < S and s2 + m[i]/2 < S:
    # evasion case
       evasion = optimizeReward(i − 1, s1 + m[i]/2, s2 + m[i]/2, m, r, dict, S)

    if s1 + m[i] < S and s2 + m[i] < S:
    # Any of the player fights, we take maximum of either values
     anyfights = max(optimizeReward(i − 1, s1 + m[i], s2, m, r, dict, S) + r[i],
                     optimizeReward(i − 1, s1, s2 + m[i], m, r, dict, S) + r[i])
```

$dict[(i, s1, s2)] = max(evasion, anyfights)$

$return\ dict[(i, s1, s2)]$

Please note that dictionary used in the above code will help us in memoization by storing the values once they have been calculated and returning them if they are needed again by recurring solution. Hence, saving us time for computation.

Time Complexity – Please note that we can have $n*S^2$ possible computations in the worst case where n is the number of monsters and S is the maximum allowed stress. Hence, time complexity for our solution is $O(n*S^2)$.

Space Complexity – As we introduced memoization, we only need to computer $n*S^2$ solutions and as we are using dictionary, at max we can have $n*S^2$ keys in our dictionary. This ensures that our space complexity is also $O(n*S^2)$.

Question 4 – Prove that getting two buckets with same likeability score is NP-complete.

To prove that a problem is NP-complete is, we need to take 3 steps:

Step 1 – We need to prove that our problem Y is in NP. This can be proved by showing that a potential solution or witness or certificate can be verified in polynomial time for the given problem. This translates into following for our problem:

- Given two set of flavors, $s_1, s_2, s_3 \dots \dots \dots, s_p$ & $s_1, s_2, s_3 \dots \dots \dots, s_q$ and corresponding set of remaining amounts for two buckets $d_1, d_2, d_3 \dots \dots \dots, d_p$ & $d_1, d_2, d_3 \dots \dots \dots, d_q$, we need to check if we can calculate L_1 and L_2 from them in polynomial time.
- This can actually be achieved in polynomial time. All we need to do is calculate the L for both the sets using the algorithm explained in question 1. Note that all we are doing is multiplying numbers for each set in increasing order of d_i/s_i (this step requires $O(n * \log n)$).

Hence, we can actually verify a potential witness in polynomial time. Hence, proved that our problem is in NP.

Step 2 – Choose another NP complete problem X. We will use subset problem (given hint in the question), which is already known to be NP-complete.

Step 3 – Final step is to show that X is poly-time reducible to Y.

- First step for step3 is to prove that we can convert inputs of X to inputs of Y in polynomial time. So what we need to do is convert an input of X i.e. subset of a set which contains integers (for the sake of simplicity) to inputs of refined ice cream problem i.e. two sets of numbers.

- **TRANSFORMATION** - Let's split the input (set of numbers) of X in such a way that one set of input sums up to k' and other half sums to $(sum - k')$, now if we can add two numbers, one to each set that was split and those two numbers will $3k - k'$ and $2k + k'$ (we already now k from subset problem definition).

- Now sum for both the sets will be $3k$ and $2k + sum$. Till this point we have actually converted the input of subset problem to our refined ice cream problem.

- Please note that all we have done is added few numbers and split up the set, all of these operations can be done in polynomial time. So, we have transformed our input for X to input for Y in **polynomial time**.

- **EQUIVALENCE (if and only if)** - Now let's say that input that we used for problem X is the actual solution i.e. $sum = k$, then we need to prove that our transformed input is also a solution for problem Y. If we put in the value of sum in $2k + sum$, then sum for both of our sets become $3k$, which is actually a solution for problem Y, refined ice-cream problem.

- Final step to prove is that if the transformed input that we got for problem Y is a solution then the original input that we chose for problem X is also a solution for problem X. This means that

$$3k = 2k + sum$$
$$sum = k$$

- sum here is nothing but the sum of all elements in set which was chosen as input for problem X and we just proved that it is equal to k, hence our input for problem X is also a solution for this problem.

- We just proved two equivalences which proves that refined ice cream problem is also NP-complete.