GA HW 03

Piyush Hinduja

October 2023

Question 1

a)

We are given a set of 'n' people who have 'm' different skills in total and there are 'k' (k ≤ n) people in the optimal solution who have all the 'm' skills.
We need to prove that after running algorithm for 4k times, the number of uncovered skills will be less than 90% of the total skills.
We will take the following claim for greedy algorithm and move forward.
Claim: We know that greedy algorithms don't guarantee an optimal solution but they claim that if a problem has an optimal solution of 'k' units then greedy approach will not be worse than 'k*log n' units where n is the total number of instances and k are the optimal instances.
Hence in our problem, if u_t is the set of uncovered skills at 't' iteration then,

$$u_{t+1} \leq u_t - \frac{u_t}{k}$$

$$u_{t+1} \leq u_t * (1 - \frac{1}{k})$$

$$\leq u_{t-1} * (1 - \frac{1}{k})^2$$

$$\leq u_{t-2} * (1 - \frac{1}{k})^3$$

......

$$\leq u_0 * (1 - \frac{1}{k})^{t+1} \leq m * (1 - \frac{1}{k})^{t+1}$$

Substituting t+1 = 4k,

$$u_{4k} \leq m * (1 - \frac{1}{k})^{4k}$$

We know that,

$$(1 - \frac{1}{k})^k \approx \frac{1}{e}$$

Hence,

$$u_{4k} \leq m * (\frac{1}{e})^4$$

$$e \approx 2.718$$
$$u_{4k} \leq m * (\frac{1}{2.718})^4$$
$$u_{4k} \leq 0.018 * m$$

Therefore for after 4k iterations, uncovered skill set will be less than or equal to 1.8% of total skill set 'm' i.e. we would have covered more than 90% of the skills.

b)

For the street surveillance problem, we have to select the minimum number of nodes to put cameras so that all the edges or streets can be monitored.
Hence, in the previous set cover problem, nodes resemble people whereas edges resemble their respective skills.

c)

We know that each edge has 2 nodes and to monitor each edge, we should place camera on at least one of it's edges.
But we can see that whenever our algorithm detects an unmonitored edge, it adds both of its nodes to the S whereas the optimal algorithm could add 1 or 2 edges depending on the solution.
This could be written as,

$$k \leq \#ofedges$$

Also for our lazy algorithm,

$$|S| \leq 2 * (\#ofedges)$$

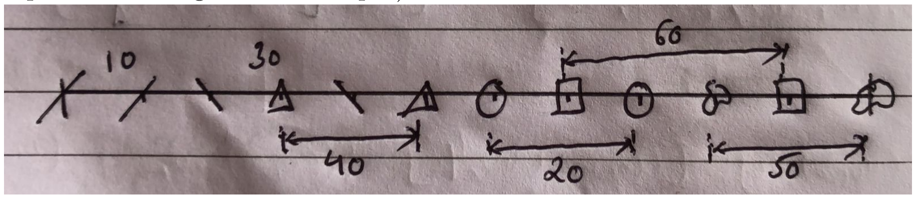
Hence, we can deduce that,

$$|S| \leq 2k$$

Question 2

a)

Let's take the below example of line having different intervals (each interval is represented using different shapes).



For simplicity, we assume the length of each interval to be 1 so the price to length ratio of each interval becomes equal to its price.
Given greedy algorithm works in the following manner,
We are given the set of intervals in the decreasing order of their ratios,
Hence, we will first add interval having ratio 60 then 40 and then finally 10, which will give us a total profit of 60+40+10=110.
But we can see that the optimal solution would be picking intervals having ratios 50, 40, 20 and 10 giving total profit of 50+40+20+10=120.

b)

The dynamic programming based algorithm for this solution could be:

```
n = Number of elements in main list (elements to be processed)
S = Sum of ratios of all the elements
Initialize dp[n+1][S+1] to -1;
main[n]; // Main array having ratios of all the intervals
Initialize processed[]; // Empty array which will store the list of intervals to be returned as answer at the end
// Initially i=0, Cost=Sum of all the ratios

function interval_dp(i, Cost):
    if(i ≥ main.length)
        return Cost;
    if(main[i] is overlapping with any interval in processed):
        return dp[i+1][Cost];
    else:
        if(dp[i][Cost] != -1):
            processed.append(i);
            return dp[i][Cost];
        else:
            processed.append(i);
            a1 = dp[i+1][Cost-main[i]]
            processed.remove(i);
            a2 = dp[i+1][Cost]
            if(a1 > a2):
                dp[i][Cost] = a2;
            else:
                dp[i][Cost] = a1;
                processed.append(i);
            return dp[i][Cost];
```

Correctness:
Base Case:
The algorithm works correctly when main.length = 1 and returns the ratio of that single element.
Inductive case:
When started with i=0, at each level, the algorithm chooses between 2 values (whichever has the lowest Cost or maximum Profit), hence we can imagine a binary tree where each node has exactly 2 children (whether to include that node or not).
The main first function call checks all the values from i=0 to i=main.length-1 and makes a choice between taking that particular element of main[i] or not.
Hence, the algorithm ensures that it checks for each and every case possible and hence gives the optimal solution.

Running Time:
The algorithm will have a running time complexity of $\mathbf{O}(n\log_2 n)$ since for each element it makes 2 function calls with the same array sizes of length 1 less than their parent.

Bonus:

In this case, the overall idea of algorithm will remain the same with just once change that if the current interval is overlapping with some other processes interval then instead of skipping it and moving forward, we will interpret it too by subtracting the Cost parameter by penalty.

Hence, the algorithm becomes:

```
function interval_dp(i, Cost):
    if(i ≥ main.length)
        return Cost;
    if(main[i] is overlapping with any interval in processed):
        Cost = Cost - penalty;
    if(dp[i][Cost] != -1):
        processed.append(i);
        return dp[i][Cost];
    else:
        processed.append(i);
```

```
a1 = dp[i+1][Cost-main[i]]
processed.remove(i);
a2 = dp[i+1][Cost]
if(a1 > a2):
    dp[i][Cost] = a2;
else:
    dp[i][Cost] = a1;
    processed.append(i);
return dp[i][Cost];
```

Running Time:
The running time of this algorithm also remains $\mathbf{O}(n \log_2 n)$ since for each element we are still checking 2 cases of same size.

Question 3

a)

Algorithm:

```
Initialize empty list 'pairs' which will store all the successful pairs;
function successful_pairs(arr, T, pairs):
    n = arr.length
    if(n ≤ 1):
        return pairs
    if(arr[0]+arr[n-1] ≥ T):
        pairs.append([arr[0], arr[n-1]])
        arr.remove(n-1)
        arr.remove(0)
    else:
        arr.remove(0)
    return successful_pairs(arr, T, pairs)
```

Running time:
Here, in the worst case, there will be no successful pairs and the algorithm will keep reducing the array by removing the first element at each function call and at most it can remove 'n' elements.
Hence, the running time of the algorithm will be O(n).

b)

Yes, this algorithm returns the largest number of possible successful pairs and it's correctness can be proved as follows:
Here, our **claim** is that the algorithm will return the maximum possible number of successful pairs (without repeating the elements).
Base Cases:
When we will have n=1, the algorithm simply return the empty 'pairs' list (since it will not satisfy the first IF condition itself) whereas when we have n=2, it will check sum of both the elements and if it is greater than or equal to T, it will add both the elements as a pair to the 'pairs' list else not.
Other Case:
i) At each step, the size of our array is decreasing by 1 or 2 elements which ensures that the base condition of $n \leq 1$ will be satisfied at some point and the program will terminate.
Lets say we have n=k at some iteration or step, then at next step the array size will be (k-1) or (k-2) based on the decision whether we have got a successful pair at current step or not.
The same process repeats at each step until finally the array size (or n) becomes 0 or 1.
ii) Also, our algorithm does the sum of the largest and the smallest element to

compare it with the threshold, which ensures that largest possible pairs compared to an approach where we check random elements for pairs.
iii) And since we are removing first or last element after each step this also ensures that each element is traversed at least once.

After noting all the above points we can conclude that we get our claim which we had claimed before.

Question 4

Notations:
$P = p_1, p_2, ..., p_n$
$S = q_1, q_2, ..., q_k$ // Local search solution
$O = x_1, x_2, ..., x_k$ // Optimal solution

a)

We know that, if we have a S of size k then it is the locally optimal solution i.e., we have compared each element of S with each element of P and then we have derived the subset S.
Now we know that O (optimal subset) also has elements from P itself but O and S are disjoint sets i.e. they don't have anything in common.
And from the first statement we know that we have compared all the elements of S with all the elements of P, so we can say that any element from O when replaced with element from S we will get a worse or equal solution than what we have in S.
Therefore for each element,

$$\sum_{i=2}^{k} d(x_1, q_i) \leq \sum_{i=2}^{k} d(q_1, q_i)$$

b)

From the above statements we can derive 2 of the following equations:

$$d(x_1, q_2) + d(x_1, q_3) + ... + d(x_1, q_k) \leq d(q_1, q_2) + d(q_1, q_3) + ... + d(q_1, q_k)$$

and

$$d(x_2, q_2) + d(x_2, q_3) + ... + d(x_2, q_k) \leq d(q_1, q_2) + d(q_1, q_3) + ... + d(q_1, q_k)$$

We know the distance is commutative property i.e. $d(a, b) = d(b, a)$
Hence,

$$d(q_2, x_2) + d(q_3, x_2) + ... + d(q_k, x_2) \leq d(q_1, q_2) + d(q_1, q_3) + ... + d(q_1, q_k)$$

We also know that $d(a, b) + d(b, c) = d(a, c)$, Hence if we add both the above equations, we get,

$$(k-1) * d(x_1, x_2) \leq 2 * [d(q_1, q_2) + d(q_1, q_3) + ... + d(q_1, q_k)]$$

c)

From the above sub-problem, we get,

$$(k-1) * d(x_1, x_2) \leq 2 * [d(q_1, q_2) + d(q_1, q_3) + ... + d(q_1, q_k)]$$

We can also deduce that,

$$(k-1) * d(x_1, x_3) \leq 2 * [d(q_1, q_2) + d(q_1, q_3) + ... + d(q_1, q_k)]$$

In general we could write as,

$$\sum_{j=1}^{k} (k-1) * d(x_1, x_j) \leq \sum_{j=1}^{k} 2 * [d(q_1, q_j)]$$

In the above equation, i could also range from 1 to k, and when we add them all we will get each right hand side term k times,

$$\sum_{i,j=1}^{k} (k-1) * d(x_i, x_j) \leq k * \sum_{i,j=1}^{k} 2 * [d(q_i, q_j)]$$

For large values of k, we can say that $(k-1) \approx k$, hence cancelling them out,

$$\sum_{i,j=1}^{k} d(x_i, x_j) \leq 2 * \sum_{i,j=1}^{k} [d(q_i, q_j)]$$

Hence Proved.

References

1)Analyzing Correctness of a recursive algorithm - Link

2)Verifying an algorithm - Link