Fall 2021: CSOR 4231 - Analysis of Algorithms

Aayush Kumar Verma, UNI: av2955

Homework 4 (Due Date: November 12, 2021)

Problem 1:

Given that we have a sorted set of real numbers denoting where each painting is placed and a guard can protect paintings at at-most distance = 1 on both sides, we can see that it is clearly a Greedy approach to place minimum number of guards such that all paintings are protected.

The algorithm works with the following idea - *if we find an unprotected painting (at position i), we place a guard on it's right, i.e. i+1*, so that the guard can protect the painting at positions *i, i+1*, and *i+2*. As we go on scanning from left to right, we maintain another list which will store the position of the guards that we place which we will return at the end. This list will be of Boolean type containing *False* if a guard is not present at that position and *True* if a guard is present on that position. If we need to find the minimum number of guards required, we can simply sum up the occurrences of *True* in this list. The size of this list is equal to the length of *X*.

We begin assuming that a guard is present at negative infinity and when we add a guard, we update the position of the last guard placed. By this argument, what follows is that a painting is unprotected if it is situated at position > (position of last guard + 1).

Algorithm 1 MinimumGuards(X):

```
last_guard = -∞
positions = []
for i = 1 to length(X) do
    positions [i] = False
end for
for i = 1 to length(X) do
    if X[i] > last_guard + 1 then
        position = min(i + 1, length(X))
        positions [position] = True
        last_guard = position
    end if
end for
return positions
```

Time Complexity Analysis and Proof of Correctness:

The above algorithm has two **for** loops running from 1 to N separately, hence giving a total run-time of N, thus a time complexity of *O(N)*. The algorithm gets the desired result correctly because it is greedily placing the guards exactly where they need to be in order to protect the maximum number of paintings. Had we been placing the guards at position *i* instead of *i+1*, we would be needing an extra guard for a setup where paintings are present at locations: *(i - 1), i, (i + 2)* whereas with the above algorithm, we ensure that if a painting is present on the left, then it would already have been covered previously so we need not worry about that. Thus, it ensures that we only look at the current and next paintings giving us

the minimum number of guards.

The algorithm maintains the optimal substructure property of Greedy Algorithms. The first Greedy choice is the first painting and the position of the guard that is 1 place on the right of the painting. This is an optimal choice because by placing a guard at this position, we are capturing the maximum distance without violating any of the constraints mentioned in the question. Hence, we proceed in the same manner for the remaining positions of paintings which monotonically gives the optimal solution to the overall problem. Also, we can have an exchange argument considering another optimal solution where we are not placing a guard to the right of an unprotected painting, we have shown above that a guard can protect maximum paintings by being placed at position *(i + 1)*, hence our optimal solution will have minimum number of guards $\leqslant$ the minimum number of guards of the other optimal solution. Hence, our Greedy Algorithm is correct.

Problem 2:

(a).

Given that S is a set of tasks $(a_1, a_2, \ldots, a_n)$ and a_i requires p_i processing time and finishes at c_i. We need to minimize the average of the completion times, i.e.:

$$\frac{1}{n}\sum_{i=1}^{n} c_i$$

Algorithm:

1. Initialize result = 0
2. N = *length*(S)
3. Sort all tasks in increasing order of processing time.
4. Run the tasks one-by-one.
5. When a task finishes, add it's completion time to result.
6. After all tasks have been processed, return the average completion time = $\frac{result}{N}$.

We can observe that this follows the optimal substructure of a Greedy algorithm when we sort the tasks in the increasing order of their processing time and run them in that order. Hence, if we run the first task in an optimal way, we can run the remaining tasks in an optimal way that will minimize the average completion time.

Time Complexity Analysis and Proof of Correctness:

Let O be an optimal solution where a_i has the least processing time but a_j is the first task that is run. Using exchange argument, assuming that we get another solution G where we run a_i first and it still has the shortest processing time. Other tasks are run in the same order, hence, the completion times of all tasks between a_i and a_j changes by a factor equal to the difference of processing time between a_i and a_j. Since, all the remaining completion times remain the same, the average completion time of $G \leqslant$ average completion time of O. We also maintain the optimal substructure property by running the task with the minimum processing time first and then scheduling the remaining tasks optimally in the same manner to combine their optimal solutions with the optimal solution of the first task to get an overall optimal solution to the actual problem. Therefore, we can say that the Greedy solution is indeed an optimal one. Since we need to sort the tasks in increasing order of their processing time, this gives us a time complexity of $O(nlogn)$ where n is the number of tasks.

(b).

Given that the tasks can only be started on or after their release time r_i with preemption such that a task can be stopped and started later, we can modify the above Greedy approach so that the job with the **minimum remaining time** is processed first. Whenever a new task is released, we add it to the task processing queue. The task processing queue is basically an augmented Min-Heap data structure that keeps in the tasks in the heap in the order of the minimum remaining time. This approach where at any instant of time we are greedily running the task which will take the minimum time to complete will give the minimum average completion at the end when all the tasks have been processed.

Algorithm:

1. Sort all tasks in increasing order of their release times.
2. Create a min-heap that keeps tasks in the order of minimum remaining time and add the first task to it.
3. Run the task at the root of the heap (task with minimum remaining time currently).
4. When a new task is released, add it to the heap with it's **remaining time = completion time - release time**.
5. Preempt the current task and update the remaining time of the current task as follows:
   current task = a_i, new task = a_j, **remaining time of** $a_i = c_{a_i} - r_{a_j}$

   Push the current task back in to the heap.
6. If a task finishes, remove it from the heap.
7. Repeat the process until there are no more tasks remaining in the heap.

Time Complexity Analysis and Proof of Correctness:

As shown in the part (a). above, if we greedily process the tasks such that the task that finishes first is processed first and here, too, we are processing that task which finishes first at any instant using the heap and preemption. Therefore, this algorithm is correct and will give the minimum average completion time at the end when all tasks are processed.

Sorting all tasks in increasing order of release time takes = *O(nlogn)*

Heap Operations:

1. Preemption of current task = *O(1)* time, since we are simply removing the root of the heap.
2. Add new task to the heap = *O(log n)*.

Overall time for the heap operations = *O(nlogn)* because we are doing the heap operations n times.

Therefore, the overall time complexity = *O(nlogn)* + *O(nlogn)* = *O(nlogn)*.

Problem 3:

(a).

(i). *Selecting activity with the largest weight:*

Consider 5 activities $(a_1, a_2, a_3, a_4, a_5)$ with start times (first column), completion times (second column), and weights (third column) as:

$a_1 = 1$	10	100
$a_2 = 1$	2	20
$a_3 = 2$	4	20
$a_4 = 4$	5	30
$a_5 = 5$	10	50

Hence, by selecting the activity with the largest weight, we select a_1 but had we selected the other 4 activities which also run from time 1 to 10, we would have got the weights as = (20 + 20 + 30 + 50) = 120 which is greater than the weight of a_1. Hence, here, the Greedy selection strategy does not produce the optimal result.

(ii). *Selecting activity with the earliest finishing time:*

Again, let's consider 2 activities (a_1, a_2) with start times (first column), completion times (second column), and weights (third column) as:

$a_1 = 1$	10	100
$a_2 = 1$	20	200

Here, again we see that by selecting the activity which finishes earliest, we pick a_1 having weight = 100 but had we chosen a_2, the weight would have been 200. Hence, the Greedy approach fails here again.

(b).

Let's define a **Dynamic Programming** based solution and the overlapping subproblems and the optimal substructure which we can then use to come up to the optimal solution for the bigger problem.

Let $\mathcal{A}$ be the set of activities $(a_1, a_2, \ldots, a_n)$ with weights $\mathcal{W}$ $(w_1, w_2, \ldots, w_n)$. We start by sorting the activities based on their finish time so that we get activities such that a_i comes before a_j if finish time of a_i is less than a_j, therefore $i < j$. Then we compute another list $\mathcal{L}(l_1, l_2, \ldots, l_n)$ where every l_i is the activity whose finish time is just before the start time of a_i. If an activity does not exist, then we set l_i to 0. Next, we come up with the recursive relation:

Let $\mathcal{O}_i$ be the optimal solution for activities $(a_1, a_2, \ldots, a_i)$, then there are 2 cases of whether an activity a_i is present in the optimal solution or not. When present, $\mathcal{O}_i = w_i + \mathcal{O}_{l_i}$, where w_i is the weight of the i_{th} activity and if the activity is not present, then $\mathcal{O}_i = \mathcal{O}_{i-1}$. If there is no activity, $\mathcal{O}_i = 0$, i.e.:

$$\mathcal{O}_i = \begin{cases} 0 : i = 0, \text{ i.e., no activity} \\ \textbf{max}(\mathrm{w}_i + \mathcal{O}_{l_i},\ \mathcal{O}_{i-1}) : \text{ otherwise} \end{cases}$$

Algorithm 2 MaxTotalWeight(A, W, L, O):

```
n = length(A)
for i = 1 to n do
    O[i] = 0
end for
for i = 1 to n do
    O[i] = max(w_i + O[l[i]], O[i-1])
end for
optimalWeight = O[n]

S = set()
i = n
while i ⩾ 1 do
    if (w_i + O[l[i]]) > O[i − 1] then
        S.append(a_i)
        i = l[i]
    else
        i = i - 1
    end if
end while
return (S, optimalWeight)
```

Time Complexity Analysis and Proof of Correctness:

Sorting the activities will take *O(nlogn)* time, creation of $\mathcal{L}$ takes *O(logn)* for n activities using Binary Search (complexity = *O(log n)*), hence a total of *O(nlogn)* for all activities, and the function above uses a single **for** and a single **while** loop, each of which run separately in linear time. Hence, the overall time complexity is $O(nlogn) + O(nlogn) + O(n) + O(n) = O(nlogn)$.

Since we have overlapping subproblems defined recursively, the dynamic programming algorithm breaks the problem in to smaller subproblems and the optimal solutions to these smaller subproblems combines to give the optimal solution to the actual problem, hence, the dynamic programming algorithm works correctly.

Problem 4:

(a).

The recurrence relations for $\alpha(v)$ and $\beta(v)$ will vary depending on whether v is a leaf node or not.

1. **If v is a leaf node:**

   C(v) is empty and T[v] will contain only 1 node (v) implying that $\alpha(v) = w(v)$ and if we remove v from S, then S becomes empty, hence, $\beta(v) = 0$. Therefore the recurrence relations are:

$$\alpha(v) = w(v)$$
$$\beta(v) = 0$$

2. **If v is not a leaf node:**

   C(v) is not empty and if we remove v from S (S is an independent set of T[v]) then there is no bound on including any child $\in C(v)$ in S, hence, we get $\beta(v) = \sum_{c \in C(v)} \alpha(c)$. For defining $\alpha(v)$, we encounter 2 cases:

   (a) **v is present in S:**

   We cannot include any child $c \in C(v)$ in S, hence, we add $\sum_{c \in C(v)} \beta(c)$ to $w(v)$.

   (b) **v is not present in S:**

   Since any child $c \in C(v)$ can be in S, we add $\sum_{c \in C(v)} \alpha(c)$ and not $w(v)$, which is same as $\beta(v)$.

   Hence, the recurrence relations, in this case, are:

$$\beta(v) = \sum_{c \in C(v)} \alpha(c)$$
$$\alpha(v) = \mathbf{max}([w_v + \sum_{c \in C(v)} \beta(c)], \sum_{c \in C(v)} \alpha(c))$$
$$\implies \alpha(v) = \mathbf{max}([w_v + \sum_{c \in C(v)} \beta(c)], \beta(v))$$

(b).

To get an optimal solution in *O(n)* run-time, we will apply a **Dynamic Programming** idea to calculate the values of α and β from the leaves to the root.

We start with a Post-order ordering (Algorithm 3) of the nodes instead of any other ordering so that it simplifies operations. Let the post-order order of nodes be denoted by $\mathcal{POST}(v_1, v_2, \ldots, v_n)$ such that if a node v_i is a child node of another node v_j, then v_i comes before v_j in the post-order ordering, i.e. $i < j$. Once we have a post-order ordering, we will initialize 2 lists for α and β of size n each to get the maximum weight (Algorithm 4).

To get an optimal set of nodes, we will run a Breadth First Search over the two lists α and β to compute and return the optimal list/set of nodes (Algorithm 5). We keep a check on a node v such that if v is in the optimal set of nodes, then we add only the grandchildren of v to the queue, skipping the children of node v. However, if node v is not in the optimal set of nodes, then add it's children to the queue.

Algorithm 3 PostOrderTraversal(node, POST []):

```
if root is NULL then
    return
end if
PostOrderTraversal(node.left, POST)
PostOrderTraversal(node.right, POST)
POST.add (node)
return POST
```

Algorithm 4 MaximumWeights(POST [], W []):

```
N = length(POST)
α = []
β = []
for i = 1 to N do
    α[i] = 0
    β[i] = 0
end for
for i = 1 to N do
    if POST[i] is a Leaf node then
        β[i] = 0
        α[i] = W[i]
    else
        β[i] = Σ_{c∈C(POST[i])} α[c]
        α[i] = max([w[POST[i]] + Σ_{c∈C(POST[i])} β[c]], β[i])
    end if
end for
maximumWeight = α[N]
return maximumWeight
```

Algorithm 5 OptimalSetOfNodes(POST [], α[], β[]):

```
result = []
N = length(POST)
Initialize queue = [POST[N]]
while queue is not empty do
    node = queue.remove()
    if α[node] > β[node] then
        result.append (node)
        for every child of node do
            for every child of child of node do
                queue.add (child of child)
            end for
        end for
    else
        for every child of node do
            queue.add (child)
        end for
    end if
end while
return result
```

Time Complexity Analysis and Proof of Correctness:

To compute α[] and β[], every element is accessed twice, once when we are calculating $\alpha[i]$ and $\beta[i]$ and once when parent of $POST[i]$ is calculated, hence it is linear in time giving us a run-time complexity of *O(N)*. The run-time complexity of BFS is *O(V + E)* where V is the number of nodes and E is the number of edges. In this tree, E = V - 1, hence, our complexity for BFS becomes O(2V - 1) and since the number of vertices = N, we have a linear complexity again, i.e. *O(N)*. Postorder also traverses a node twice, hence giving a linear run-time complexity again. Hence, the final run-time complexity is *O(N)*.

We include a node v in the optimal set only when $\alpha[v] > \beta[v]$ because the weight of v is optimal and we do not include the children of v because they are directly connected to v, therefore, we go on adding the grandchildren of v. On the contrary, if $\beta[v] \geqslant \alpha[v]$, this implies that the weight of v is not optimal, hence, we do not add it to the result, consequently adding all it's children to the queue to be processed for optimal weight. Hence, this signifies that during the algorithm, we are ensuring that only the nodes with optimal weights are processed in the queue, thereby proving the correctness of the algorithm.

Problem 5:

For cutting a piece of cloth, we have 3 options:

1. Cut it along the X-axis.
2. Cut it along the Y-axis.
3. Do not cut it (Base Case).

The base case is when the dimension is 1 x 1, i.e., the cloth cannot be cut further. For a given dimension **i x j**, let the optimal solution be S(i, j). So we can make the cuts in the following ways, for a random dimension (i x j) of the cloth:

1. Make a cut k horizontally. So we need to find the optimal solution of the subproblems *S(k, j)* and *S(i-k, j)*. We subtract the cost of cutting, j and then select the maximum result from the most optimal k, where $k \in [1, i)$.
2. Make a cut l vertically. So we need to find the optimal solution of the subproblems *S(i, l)* and *S(i, j-l)*. We subtract the cost of cutting, i and then select the maximum result from the most optimal l, where $l \in [1, j)$.
3. We do not cut the piece and therefore add the maximum of the 3 - price of cloth dimensions *(i x j)*, price of cloth dimensions *(j x i)*, and 0 if the price of the cloth of this dimension is not given. **Here, we are assuming that the price of cloth of dimension (i x j) and (j x i) can be different. If the prices are same, then we simply check for prices of i x j or prices of j x i and if present, we add that value, else we add 0, so effectively, this does not cause a problem in the below mentioned algorithm, or, in any way, hinders the run-time complexity of the algorithm. The below mentioned algorithm is a generalized case which will also work when the prices of rotated pieces are same**.

To summarize:

$$\text{S(i, j)} = \textbf{max} \begin{cases} \textbf{max}(S(k,j) + S(i-k,j) - j) \colon k \in [1, i), \\ \textbf{max}(S(i,l) + S(i,j-l) - i) \colon l \in [1, j), \\ \text{price of dimension (i x j)}, \\ \text{price of dimension (j x i)}, \\ 0 \end{cases}$$

... which is our required recurrence relation. We provide the inputs *prices*, which is the provided list of prices for given dimensions, and the initial dimensions of the cloth represented by X and Y. We will follow a top-down approach of Dynamic Programming where we create a X x Y matrix (*result [][]*) to store the results and our final result will the overall optimal result (stored at *result [X][Y]*) coming from the optimal solutions of our subproblems.

Time Complexity Analysis and Proof of Correctness:

To cover the entire dimension of the cloth, we are traversing it in a matrix fashion with dimensions (X x Y) giving a run-time complexity of *O(XY)*. Now, we are checking for the optimal k and optimal l every time while traversing the cells of this matrix, each running along a row and a column respectively for a possible cut. Also, the lookup of the price in the *prices* takes linear time. Hence, we get the overall run-time complexity of $O(XY(X + Y + n))$, where n is the number of tuples in *prices*.

For proof of correctness, we have shown above how the bigger problem is being divided into smaller subproblems giving us 5 cases according to if and where a cut is made (4 cases when prices [i x j] = prices [j x i]). The dynamic programming approach stores the optimal solutions of the subproblems and then uses them to find the optimal solution of the bigger problem, thereby maintaining the optimal substructure and removing the need to calculate the solutions of overlapping subproblems again and again, thereby solving the recurrence relation defined above. Therefore, the algorithm is correct.

Algorithm 6 ClothCutting (prices, X, Y):

```
Initialize result [][]
for i = 1 to X do
    for j = 1 to Y do
        result [i][j] = −∞
    end for
end for

if prices [1 x 1] is present then
    result [1][1] = prices [1 x 1]
else
    result [1][1] = 0
end if

for i = 1 to X do
    for j = 1 to Y do

        if i = 1 and j = 1 then
            continue
        end if

        for k = 1 to i-1 do
            result[i][j] = max ((result[k][j] + result [i-k][j] - j), result [i][j])
        end for

        for l = 1 to j-1 do
            result[i][j] = max ((result[i][l] + result [i][j-l] - i), result [i][j])
        end for

        if prices [i x j] is present then
            result [i][j] = max (result[i][j], prices [i x j])
        end if

        if prices [j x i] is present then
            result [i][j] = max (result[i][j], prices [j x i])
        end if

        result [i][j] = max (result [i][j], 0)

    end for
end for

return result [X][Y]
```