CS601 Algorithms and Complexity **2021 Jul-Nov**

Assignment 1 Solution

Total Marks: 30 *Deadline:*

A detailed explanation is given for each algorithm, on how exactly it was obtained. Such details are not expected in your solution. But, if you are writing a pseudocode, then an explanation of the code is expected, for example what a variable is supposed to compute after each iteration.

Que 1 [9+1 marks]. (a) Suppose there is a trader for a particular commodity, whose license allows them to buy it only once and sell it only once during a season. Naturally, the commodity can be sold only after it is bought. The price of the commodity fluctuates every day. Once the trader buys the commodity, they have to also pay a rent on per day basis till they sell it. The trader wants to compute the maximum profit they could have made in the last season.

Design an $O(n)$-time algorithm, where the input is the list of prices $\{p_1, p_2, \ldots, p_n\}$ for the n days in the last season and a rent r (per day rate), and the output is the maximum possible profit. Assume addition, subtraction, comparison etc are unit cost. Only six marks if the running time is $O(n \log n)$. No marks for $O(n^2)$ time.

Sample input: Prices: 70, 100, 140, 40, 60, 90, 120, 30, 60. Rent per day: 15

Output: Max profit: 40

Buying at price 70, selling at 140, paying rent for the two days gives net profit of 40. Another option, for example, was buying at 40 and selling at 120, but then one has to pay the rent for three days, which means a lower profit of 35).

Ans 1 (a). Let us try to design an algorithm using the subproblem idea. Let us assume that we already have a 'solution' for the $n-1$ days, that is, the maximum profit Q_{n-1} one can make during the first $n-1$ days. When we bring in the n-th day, what are the possibilities for the optimal solution. Either the optimal solution is confined to the first $n-1$ days (in that case Q_{n-1} would be the answer) or the n-th day is involved in the optimal solution. The second possibility simply means that the commodity is sold on the n-th day. So, we need to figure out what is the maximum profit when the commodity is sold on the n-th day. For that, we need to find the minimum possible value of the (buying price+rent paid). Let us ask the subproblem itself to compute this value.

Let us define B_i to be the minimum possible value of the (buying price+rent paid) during the first i days (assuming that we have not sold so far). Here, we would be adding the rent from the day of purchase till the i-th day. To be more precise,

$$B_i = \min_{1 \leq j \leq i} \{p_j + r \times (i - j + 1)\}.$$

To understand the above equation recall that if you buy it on the j-th day, the rent accumulated till the i-th day is $r \times (i - j + 1)$. If we can compute the value of B_{n-1} recursively, then the maximum profit after n days can be written as

$$Q_n = \max\{Q_{n-1}, p_n - B_{n-1}\}.$$

Now, can we compute B_i recursively? For that let us look at the equation for B_{i+1}.

$$\begin{aligned} B_{i+1} &= \min_{1 \le j \le i+1} \{p_j + r \times (i - j + 2)\} \\ &= \min_{1 \le j \le i+1} \{p_j + r \times (i - j + 1) + r\} \\ &= \min_{1 \le j \le i+1} \{p_j + r \times (i - j + 1)\} + r \\ &= \min \left\{ \min_{1 \le j \le i} \{p_j + r \times (i - j + 1)\}, p_{i+1} \right\} + r \\ &= \min \{B_i, p_{i+1}\} + r. \end{aligned}$$

The following code implements the above idea. B will be a variable maintaing the current minimum possible value of the (buying price+rent paid). Q will be a variable maintaining the current value of maximum profit.

```
B=\infinity;
Q=0;
for i = 1 to n {
        Q = max(Q, p_i - B);
        B = min(B, p_{i}) + r;
}
Output Q;
```

The running time is clearly $O(n)$, since there n iterations and each iterations sees constantly many operations.

There could be variation in the solution depending on whether the buying or selling days have been included/excluded from the rent calculations. Any assumption is fine.

Divide and conquer approach: There is also a divide and conquer approach that can solve this problem in $O(n \log n)$ time. As mentioned, this will only get you 6 marks. The idea is to divide the n days of the season into two halves and compute the maximum profit for each set of $n/2$ days recursively. Now, there are three possibilities for the optimal solution for n days.

- Both buying and selling happens in first $n/2$ days.
- Both buying and selling happens in last $n/2$ days.
- Buying happens in the first $n/2$ days and the selling happens in the last $n/2$ days.

In the first two cases, the optimal value has already been computed recursively.

Let us see how to compute the optimal value in the third case – buying in first $n/2$ days and selling in last $n/2$ days. If you try all possible combinations of buying days and selling days, there will be $n/2 \times n/2$ combinations, which is too much. Here is a better idea. Observe that we can decide the best buying day and best selling day independent of each other.

The best buying day is given by the number $j \in \{1, 2, \dots, n/2\}$ that minimizes $p_j + (n/2 - j + 1) \times r$ (buying price on the j-th day plus the total rent from j-th day till $(n/2)$-th day).

Similarly, the best selling day is given by the number $i \in \{n/2 + 1, n/2 + 2, \dots, n\}$ that maximizes $p_i - (i - n/2 - 1) \times r$ (selling price on the j-th day minus the total rent from $(n/2 + 1)$-th day till i-th day).

Both these days can found in $O(n)$. Thus, we can compute the maximum profit possible for the third case in $O(n)$. Thus, we get the following recurrence for the running time

$$T(n) = 2T(n/2) + O(n).$$

This gives us $O(n \log n)$.

(b) If instead, the rent was charged as a fixed percentage of the selling price, can you still design an $O(n)$-time algorithm? Yes or No? Give one or two sentences justifying your answer. No need to give the actual algorithm.

Ans 1 (b). It seems unlikely. In the first algorithm, we could compute the variable B_i that was the minimum possible value of the (buying price+rent paid) during the first i days. This is no longer possible to compute independently of the selling day information.

Que 2 [9+1 marks]. (a) Suppose there is a rectangular field with multiple cameras installed at various points in the field. Let the four corners of the field be $(0,0), (N,0), (0,M), (N,M)$ for some $N, M \geq 0$. Each camera has a 90 degrees *field of view* and is oriented such that a camera at point (α, β) can cover all points in the rectangle described by $0 \leq x \leq \alpha$ and $0 \leq y \leq \beta$ (see figure 1). A point is said to be covered, if it is covered by at least one camera.

Design an algorithm that takes the list of 2-D coordinates for the n cameras as input and outputs the total area of the covered points. The running time should be $O(n \log n)$, where comparison, addition, multiplication etc are assumed to be unit cost. Only three marks if the running time is $O(n^2)$, no marks for a higher running time.

Sample input: (4,5), (3,9), (7,4), (5,7) . Output: 49 (see Figure 1)

Hint: When the target running time is $O(n \log n)$, your two friends are sorting and divide and conquer. You can take help from one of them or both.

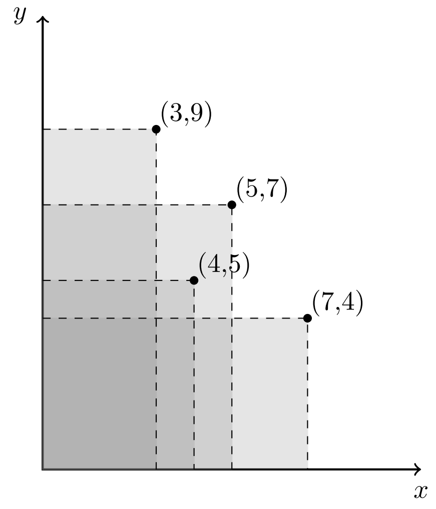

Figure 1: Shaded area is covered by the cameras.

Ans 2 (a). There are multiple ways to design this algorithm, like the hidden/visible lines problem discussed in the class. We will discuss two approaches, one with divide and conquer and the other iterative approach with sorting. There is possibly a third approach which doesn't require sorting, but requires a data structure that can support $O(\log n)$ time insertion/deletions. We will not discuss it here.

Any of the approaches will get full marks, if correct.

To compute the area covered, we will basically find out the set of 'useful' cameras. A camera at (α, β) is said to be useless if it is covered by another camera, that is, there is a camera at (γ, δ) with $\gamma \geq \alpha$ and $\delta \geq \beta$. Any camera which is not useless is said to be useful. We will maintain the coordinates of the useful cameras in the following form: $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$, where $0 < x_1 < x_2 < \cdots < x_k$. By the definition of useful cameras we will have $y_1 > y_2 > \cdots > y_k > 0$. It is not hard to see that the set of points covered by these useful cameras is given by the following union of rectangles

$$[0, x_1] \times [0, y_1] \bigcup [x_1, x_2] \times [0, y_2] \bigcup [x_2, x_3] \times [0, y_3] \bigcup \cdots \bigcup [x_{k-1}, x_k] \times [0, y_k].$$

The area of this set of points is

$$x_1 y_1 + (x_2 - x_1) y_2 + (x_3 - x_2) y_3 + \cdots + (x_k - x_{k-1}) y_k.$$

To compute the set of useful cameras, we first discuss the divide and conquer approach.

Approach 1: Divide and conquer. We will divide the given set of n cameras into two groups with $n/2$ cameras each. We will recursively apply the same algorithm to compute the set of useful cameras in each of the two groups. Let the two sets of useful cameras (in increasing order of x-coordinates) be located at

$$(a_1, b_1), (a_2, b_2), \dots, (a_k, b_k) \text{ and } (p_1, q_1), (p_2, q_2), \dots, (p_\ell, q_\ell).$$

We will describe the 'merge' algorithm that will take these two lists and output the list of useful cameras from the whole set of n cameras. The idea is to scan both the lists from left to right. Our goal is to detect if a camera from the first list is covered by a camera from the second list (or vice versa), and if not so then add it in the output list of useful cameras. Consider for example a camera in the first list located at (a_i, b_i). If (a_i, b_i) is covered by some camera in the second list located at (p_j, q_j) then it must be that $p_j > a_i$. Moreover, since $q_1 > q_2 > \cdots > q_\ell$, we can say that if (a_i, b_i) is covered by any camera in the second list then it must also be covered by the first j such that $p_j > a_i$. This means that to check if (a_i, b_i) is covered or not we don't need to go through the complete second list. We can simply check the camera from the second list which comes immediately after (a_i, b_i). If that particular camera doesn't cover (a_i, b_i), then no other camera from the second list will. The following pseudocode implements this idea for the 'merge' algorithm.

Algorithm 1 The merge algorithm

```
 1: Input: $(a_1, b_1), (a_2, b_2), \dots, (a_k, b_k)$ and $(p_1, q_1), (p_2, q_2), \dots, (p_\ell, q_\ell)$.
 2: $i \leftarrow 1$; // index to traverse the first list
 3: $j \leftarrow 1$; // index to traverse the second list
 4: OutputList $\leftarrow$ Empty List;
 5: Define $(a_{k+1}, b_{k+1}) = (\infty, 0)$ and $(p_{\ell+1}, q_{\ell+1}) = (\infty, 0)$
 6: while $i \leq k$ OR $j <= \ell$ do
 7:     if $(a_i < p_j)$ then
 8:         if $(b_i > q_j)$ then
 9:             add $(a_i, b_i)$ to OutputList.
10:         end if
11:         $i \leftarrow i + 1$;
12:     else if $(p_j < a_i)$ then
13:         if $(q_j > b_i)$ then
14:             add $(p_j, q_j)$ to OutputList.
15:         end if
16:         $j \leftarrow j + 1$;
17:     end if
18: end while
```

Let us see what's happening in each iteration of the while loop. In any iteration, a_i and p_j are the next x-coordinates in the two lists, respectively, during the left to right scan. Line 7 and 12 simply checks which x-coordinate will come first, a_i or p_j. Suppose a_i comes first. Then (a_i, b_i) is added to the output list only if (a_i, b_i) is **not covered** by the next camera in the other list, i.e., (p_j, q_j). As mentioned earlier, if it is not covered by (p_j, q_j), it will not be covered by anything that comes after. The other case, when p_j comes first, is similar.

It is easy to see that the running time of the merge algorithm is $O(\ell + k)$, which is in turn bounded by $O(n)$. Overall running time of the divide and conquer approach will be given by

$$T(n) = 2T(n/2) + O(n) \implies T(n) = O(n \log n).$$

Approach 2: Sorting. Let's first sort all the camera locations in an increasing order with respect to their x-coordinates. Suppose after the sorting, the coordinates are $(\alpha_1, \beta_1), (\alpha_2, \beta_2), \dots, (\alpha_n, \beta_n)$. We will go over the cameras in this order and after each iteration we will maintain the set of useful cameras among the ones seen so far. Let A_i be the set of useful cameras in the first i cameras. We will show how to compute A_{i+1} from A_i. Suppose the coordinates for A_i are given by the following: $(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k)$, for $0 < x_1 < x_2 < \cdots < x_k$ and $y_1 > y_2 > \cdots > y_k > 0$. Now, we introduce a camera at $(\alpha_{i+1}, \beta_{i+1})$.

Since α_{i+1} is the largest x-coordinate seen so far, clearly, the corresponding camera not covered by any of the ones seen before. Thus, we would like to add $(\alpha_{i+1}, \beta_{i+1})$ to the set of useful cameras. But, before that we must remove any previous ones that covered by $(\alpha_{i+1}, \beta_{i+1})$. Since $x_j < \alpha_{i+1}$ for every $1 \le j \le k$, the set we need to remove is given by

$$\{(x_j, y_j) : 1 \le j \le k, y_j < \beta_{i+1}\}.$$

Moreover, recall that $y_1 > y_2 > \cdots > y_k$. Let $j^* \in \{1, 2, \ldots, k\}$ be the minimum index j such that $y_j < \beta_{i+1}$ (if there is no such index then simply define $j^* = k + 1$). Clearly, the set to be removed is

$$\{(x_j, y_j) : j^* \le j \le k\}.$$

We can do a binary search for j^* and compute A_{i+1} as

$$A_{i+1} \leftarrow \text{remove } \{(x_j, y_j) : j^* \le j \le k\} \text{ from } A_i \text{ and add } (\alpha_{i+1}, \beta_{i+1}).$$

The following pseudocode summarizes this whole idea.

Pseudocode is not crucial here, just describing the algorithm in words will be sufficient.

Algorithm 2 The iterative algorithm

```
1: Input: An array $I$ with entries as 2D coordinates.
2: Output: an array with the set of coordinates that are not covered by any other input coordinate.
3: Sort the input array in increasing order of $x$ coordinates.
4: $A \leftarrow$ an array of size $n$ with all its entries as $(\infty, 0)$; // stores the coordinates that are not covered by
   any other seen so far.
5: $k \leftarrow 0$; // indicates the index up to which array $A$ has been filled up.
6: for $i = 1$ to $n$ do
7:     Binary search for the minimum index $j^* \in \{1, 2, \ldots, k+1\}$ such that $A[j].y < I[i].y$;
8:     $A[j^*] \leftarrow I[i]$;
9:     $k \leftarrow j^*$;
10:    $A[k+1] \leftarrow (\infty, 0)$;
11: end for
12: Output $A[1 : k]$
```

(b) Note that if we have a camera at $(5, 7)$ then it is useless to have another camera at $(4, 5)$, because the field of view of the second camera is completely covered by the first camera. If we assume that there are no cameras at such 'useless' locations, does the problem become easier? Yes or No? Give one or two sentences justifying your answer. No need to give the algorithm.

Ans 2 (b). Yes. As discussed in the beginning of part (a), if we just have useful cameras, then computing the area covered is straightforward.

Que 3 [9+1 marks]. (a) Design a randomized algorithm for finding the third largest element in a given array. The expected number of comparisons should be $n + O(\log n)$ for array size n and you also need to prove that.

Hint: Here is one way to do it. First randomly permute the array. Then go over the array iteratively while maintaining the top 3 elements seen so far. Design the algorithm so that in most iterations, it is more likely that only one comparison happens. To find the expected number of total comparisons, we can first find the expected number of comparisons in each iteration and then add them all up. To analyze a particular iteration, you will need to find out the probabilities of having one comparison, two comparisons, three comparisons and so on. Then expectation for a particular iteration can be computed by simply applying the expectation formula.

Ans 3 (a). The algorithm is along the lines of the one for second minimum element that we discussed in the class. The idea is to maintain the top 3 elements seen so far. Whenever a new element comes, we first compare it with the third largest element. The hope is that in most of the iterations, the new element will be smaller than the third largest one, and there will no need for further comparisons in this iteration. If the new element is larger than the third largest, we need to further make appropriate comparisons with largest and second largest. Here is the pseudocode.

Algorithm 3 The iterative algorithm

```
 1: **Input:** An array $A$ of $n$ elements.
 2: **Output:** Largest, second-largest, and the third-largest elements in $A$.
 3: Randomly permute the array so that each permutation has probability $1/n!$.
 4: largest, second-largest, third-largest $\leftarrow -\infty$.
 5: **for** $i = 1$ to $n$ **do**
 6:     **if** $A[i] >$ third-largest **then**
 7:         third-largest $\leftarrow A[i]$;
 8:         **if** $A[i] >$ second-largest **then**
 9:             Swap(second-largest, third-largest);
10:             **if** $A[i] >$ largest **then**
11:                 Swap(largest, second-largest);
12:             **end if**
13:         **end if**
14:     **end if**
15: **end for**
```

Let us first find out the expected number of comparisons in each iteration. In the i-the iteration (for $i \leq 3$), the number of comparisons X_i have the following three possibilities

$$X_i = \begin{cases} 1 & \text{if } A[i] \leq \text{third-largest}, \\ 2 & \text{if third-largest} < A[i] \leq \text{second-largest}, \\ 3 & \text{if second-largest} < A[i]. \end{cases}$$

Note that here the variables 'third-largest' and 'second-largest' mean third-largest and second-largest among the first $i-1$ elements. Let's find out the probabilities for each of these possibilities. Focus on the **set** of first i elements of the array. Once you fix this set, each of the $i!$ permutations of the i elements is equally likely. In particular, the probability that $A[i]$ is the largest among the first i is $1/i$. Similarly, $1/i$ is the probability for each of the two events – $A[i]$ is the second-largest element among the first i and $A[i]$ is the third-largest element among the first i. Remaining probability of $1 - 3/i$ is for the event that $A[i]$ is fourth-largest or smaller.

Now, coming back to the number of comparisons X_i, we can write the following (for $i \leq 3$)

$$X_i = \begin{cases} 1 & \text{if } A[i] \text{ is the fourth-largest or smaller among the first } i, (\text{probability } 1 - 3/i) \\ 2 & \text{if } A[i] \text{ is the third-largest among the first } i, (\text{probability } 1/i) \\ 3 & \text{if } A[i] \text{ is the second-largest or largest among the first } i (\text{probability } 2/i). \end{cases}$$

By the above probabilities, the expectation of X_i can be written as (for $i \leq 3$)

$$\mathbb{E}[X_i] = 1 \times (1 - 3/i) + 2 \times 1/i + 3 \times 2/i = 1 + 5/i.$$

Now, to compute the expected number of total comparisons, we will use linearity of expectation. We know that that the total number of comparisons X is

$$X = \sum_{i=3}^{n} X_i + \text{ number of comparisons in first two iterations.}$$

By linearity of expectation,

$$\mathbb{E}[X] = \sum_{i=3}^{n} \mathbb{E}[X_i] + \text{ number of comparisons in first two iterations.}$$

Depending on how you count the comparisons with infinity, the number of comparisons in the first two iterations are at most 6.

$$\begin{aligned}
\mathbb{E}[X] &\leq \sum_{i=3}^{n}(1+5/i)+6 \\
&\leq n-3+\sum_{i=3}^{n} 5/i+6 \\
&\leq n-3+\sum_{i=3}^{n} 5/i+6 \\
&\leq n+3+5\log n
\end{aligned}$$

Hence the expected number of total comparisons is $n + O(\log n)$.

(b) If you want to design such an algorithm to find the k-th largest element, what can be the best possible expected number of comparisons? $n+O(k\log n)$, $n+O(k^2\log n)$, $n+O(k\log k\log n)$, or some other function of n, k ? Don't need to give a detailed analysis, just write one or two sentences to justify your answer.

Ans 3 (b). If we just follow the same idea to maintain top k elements, the expression for expected number of comparisons in iteration i will look like

$$\mathbb{E}[X_i] = (1-k/i)1 + (1/i)2 + (1/i)3 + \cdots + (1/i)(k-1) + (2/i)k = 1 + \frac{(k)(k-1)/2-2}{i}$$

Summing up this over all values of i, we will get the expected number of total comparisons as

$$\mathbb{E}[X] = n + O(k^2 \log n).$$

A better idea would be to maintain a min-heap of top k elements. For any new element, first compare it with the k-th largest element so far, that is, the root of the heap. If the new element is smaller then do nothing. If the new element is larger then remove the current root of the heap and insert the new element in the heap (and heapify). This takes $O(\log k)$ comparisons. In summary, in the i-th iteration, we are doing only one comparison with probability $1-k/i$ and $O(\log k)$ comparisons with probability k/i. Putting everything together, the expected number of total comparisons comes out as

$$\mathbb{E}[X] = n + O(k \log k \log n).$$

Any of the above two answers will get 1 mark.