CS218 Design and analysis of algorithms **Jan-Apr 2024**

Quiz 1 solutions

Total Marks: 20 *Time: 55 minutes*

Instructions.

- Please write your answers clearly and concisely.

Question 1 [5 marks]. There are n students in a class, with roll numbers from 1 to n. There was a quiz which exactly $n-1$ students attended. We have a spreadsheet where these $n-1$ roll numbers are written in a column in increasing order (from row 1 to row $n-1$). You need to find the roll number of the absent student. How will you do this by reading only $O(\log n)$ rows in the spreadsheet?

Answer 1 . Check the $n/2$th row. If the roll number is greater than $n/2$ then the missing roll number is between 1 and $n/2$. Otherwise it's between $n/2+1$ and n. We can use this idea to do a binary search for the missing roll number. When our search range for the missing roll number is between $[i, j]$, then check the row number $mid = \lfloor (i+j)/2 \rfloor$, if the roll number present is greater than $\lfloor (i+j)/2 \rfloor$, then the new search range is the left half $[i, mid]$, otherwise it's the right half $[mid+1, j]$. Since, search range reduces by a factor of 1/2 with every query, we will have to read only $O(\log n)$ rows.

Full marks if the criterion for choosing the left or right half is correctly described. If the condition is correctly checked but the half is incorrectly chosen, then 3 marks.

Pseudocode (not expected in the solution) :

```
Intialize $start \leftarrow 1$; $end \leftarrow n$;
while($start < end$){
    $mid \leftarrow \lfloor (start + end)/2 \rfloor$;
    Check the row number $mid$.
    If the roll number present is more than $mid$ then $end \leftarrow mid$.
    else $start \leftarrow mid + 1$.
    }
return $start$;
```

Question 2 [5 marks]. 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. There is also a rule that one cannot sell the commodity for more than twice the purchase price. So, if they buy it on day i and sell it on j, then the profit they make is $\min\{p_j, 2p_i\} - p_i$, where p_i and p_j are the prices for the i-th and j-th days, respectively. 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 the output is the maximum possible profit.

Sample input: Prices: 70, 100, 140, 40, 60, 90, 120, 30, 60.

Output: Max profit: 70

Answer 2. In this problem, the choice of the order (left to right, or right to left) seems to be quite important.

Idea. Suppose we already have the solution for last $n-1$ days. That is, we know the maximum profit when buying and selling both happen in last $n-1$ days, say profit$_{n-1}$. To find the solution for n days, we now need to consider the case that the trader buys on day 1 (definitely cannot sell on day 1). If they buy on day 1, the purchase price will be p_1. Then to get the maximum profit, they should sell it on a day which has the maximum price among the last $n-1$ day, say it is p_j. We can also assume that this maximum price is already computed by the subproblem. Then, as mentioned in the problem, the profit will be

$$\min\{p_j, 2p_i\} - p_i.$$

We can compare this with profit$_{n-1}$ and give the maximum of the two as the final answer.

The time complexity recurrence seems to be $T(n) = T(n-1) + O(1)$, which gives $O(n)$.

Implementation: We go over the list from **right to left** and maintain the following: (i) max-profit (ii) max-price. When we are at the i-th day, the two variables are updated as follows:

$$\text{selling-price} \leftarrow \min\{\text{max-price}, 2p_i\}$$

$$\text{max-profit} \leftarrow \max\{\text{max-profit}, \text{selling-price} - p_i\}$$

$$\text{max-price} \leftarrow \max\{\text{max-price}, p_i\}.$$

In each iteration the update is in constant time. Hence, the running time is $O(n)$.

Pseudocode :

```
Initialize max-profit ← 0; max-price ← $p_n$;
for ($i \leftarrow n-1$ to 1){
    selling-price ← min{max-price, $2 \times p_i$};
    max-profit ← max{max-profit, selling-price − $p_i$};
    max-price ← max{max-price, $p_i$}.
    }
return max-profit;
```

Solutions in any of the above three ways are acceptable.

If you go over the array from left to right, the update will more complicated. Possibly you can get $O(n \log n)$ time. Divide and conquer will also give $O(n \log n)$.

Question 3 [7+3 marks]. A subsequence of a given sequence is obtained by deleting some elements from the sequence, without changing the order of the remaining elements. For example, (6, 1, 8) is a subsequence of (3, 6, 1, 5, 8, 2). An increasing subsequence is one where the elements are in increasing order. For example, the (6, 8) is an increasing subsequence of (3, 6, 1, 5, 8, 2).

For a given length-n array of distinct integers, for any $1 \leq i \leq n$, let S_i be the number of increasing subsequences ending at the i-th element.

For example, in the sequence (3, 6, 1, 5, 8, 2), there are 8 increasing subsequences ending at the 5-th element.

(8)	(3, 8)
(6, 8)	(1, 8)
(5, 8)	(3, 6, 8)
(3, 5, 8)	(1, 5, 8)

Given the sequence $(a_1, a_2, \ldots, a_n)$ in an array, we want to design an algorithm to compute the number S_i for every $i \in \{1, 2, \ldots, n\}$, using a balanced binary search tree (BST). As we go over the given array, we will keep inserting the new elements in the BST (with array element as the key value).

(i) What all information should we store at the nodes of the BST, so that when we insert a_i in the BST, we are able to compute S_i in $O(h)$ time, where h is the height of the tree. How will you compute S_i?

(ii) When we insert a_i in the BST, how should we update the information in the nodes of BST. Argue that it is doable in $O(h)$ time, where h is the height of the tree.

Answer 3. (i) We assume that each node has a pointer to its parent, left child, and right child in the BST (it is fine if this is not written explicitly). For the node X with value a_j, we store the following information:

- X.subseq $\leftarrow$ the number of increasing subsequences ending at a_j, the key value of node X.
- X.subtree-sum $\leftarrow$ the sum of Y.subseq for all nodes Y in the subtree rooted at X (including X itself).

Actually subseq is not essential, as it can be obtained from subtree-sum of the two children and the node itself.

2 marks if the subtree-sum is mentioned clearly.

When we insert a_i, we can compute S_i with the following formula.

$$S_i = 1 + \sum_{\substack{j<i, \\ a_j<a_i}} S_j$$

Here, 1 is for the subsequence with single element a_i. Then we count all increasing subsequences which can be appended before a_i.

2 marks for the above formula, even if it's not written explicitly.

To compute this sum, Instead of going over all such j, we will make use of subtree sum information.

Initialize $S_i \leftarrow 1$.

Follow the path from the a_i node to the root. For all nodes Z which are on this path, and whose key value is smaller than a_i, we add Z.subseq and (Z.left).subtree-sum to S_i, where Z.left is the left child of Z. This is because all the nodes in the subtree rooted at Z.left have smaller key value than a_i.

3 marks for this update.

Clearly, the computation is in $O(h)$ time.

(ii) When we insert a_i, the subseq field for this node is set to be S_i computed in the first part. Also the subtree-sum at this node will be S_i, as there are no children yet.

Now, the subseq field at any node does not need to updated. We only need to update the subtree-sum field. Follow the path from the a_i node to the root. For all nodes Z on this path, we add S_i to Z.subtree-sum.

Clearly, the update can be done in $O(h)$ time.

3 marks for this update.