CS218 Design and analysis of algorithms **Jan-Apr 2025**

Quiz 1

Total Marks: 20 *Time: 55 minutes*

Instructions.

- Please write your answers clearly and concisely.

Question 1 [3 marks]. For any given number a, we want to compute a^{27}. Show that you can do it in at most seven multiplications.

Answer 1. Here is one way. Compute a, a^2, a^4, a^8, a^{16}, each by squaring the previous term. We have done 4 multiplications so far. Now, we will directly use these powers. Note that $a^{27} = a^{16} \times a^8 \times a^2 \times a$, which needs 3 more multiplications.

Question 2 [3 marks]. We want to find an approximate value of $\log_2 3$ (that is, x such that $2^x = 3$) using **binary search**. To begin with, we know that $1 < \log_2 3 < 2$. So, our initial search range is $(\text{start} = 1, \text{end} = 2)$. Show the steps of binary search till the size of the search range becomes at most $1/8$. In each step, clearly show what is the search range and how you decide whether to go in the left half or right half.

Hint: you may need to compare $2^{\frac{\alpha}{\beta}}$ with 3; instead, you can compare 2^α with 3^β.

Answer 2.

- $start = 1, end = 2$.
- $mid = (1 + 2)/2 = 3/2$.
- Compare $2^{3/2}$ with 3. $2^3 < 3^2$, hence, $2^{3/2} < 3$.
- Thus, we go to right half.
- $start = 3/2, end = 2$.
- $mid = (3/2 + 2)/2 = 7/4$.
- Compare $2^{7/4}$ with 3. $2^7 > 3^4$, hence, $2^{7/4} > 3$.
- Thus, we go to left half.
- $start = 3/2, end = 7/4$.
- $mid = (3/2 + 7/4)/2 = 13/8$.
- Compare $2^{13/8}$ with 3. $2^{13} > 3^8$, hence, $2^{13/8} > 3$.
- Thus, we go to left half.
- $start = 3/2, end = 13/8$.

Question 3 [7 marks]. (Moving Median): In statistics, a robust estimate of a trend is obtained by considering *moving medians*, which is not susceptible to rare anamolies.

For a given array of n integers $a_1, a_2, \ldots, a_n$, and a given parameter k, we want to compute an array M such that

$$M[j] = \text{median}(a_{j-k}, a_{j-k+1}, \ldots, a_j, \ldots, a_{j+k-1}, a_{j+k}), \text{ for any } j \text{ between } k+1 \text{ and } n-k.$$

Other entries in M are supposed be 0. Recall that median of an odd size set of numbers is just the middle number in the sorted order.

Design an $O(n \log k)$ time algorithm for this.

Hint: A balanced binary tree may help.

Example.
Input: 2, 8, 4, 7, 8, 1, 5, 3 and $k = 2$
Output: 0, 0, 7, 7, 5, 5, 0, 0.

Answer 3. First let us see how to find predecessor or successor of an element x in a binary tree.

- Predecessor: Find x in the tree using the standard binary tree search. If x has a left child y, then from y keep going to the right child till there is no right child. The last node reached is predecessor.

  Consider the case when x has no left child. Then go to closest ancestor y such that x is in the right subtree of y. This ancestor y is the predecessor. If there is no such ancestor, then there is no predecessor.

  Similarly one can define successor.

In the answer, the algorithm to find predecessor or successor is not expected. The time to find them for any node is $O(h)$, where h is the height of the tree.

We will insert the first $2k+1$ numbers in a balanced binary tree one by one. We will find the median of these $2k+1$ numbers by first finding the largest node in the binary tree and then using predecessor operation k times. We will set $M[k+1]$ as the value of the median. And we will also keep a pointer *med* to the node storing the median. Finding predecessor takes $O(\log k)$ time, hence, total time spent here is $O(k \log k)$.

For simplicity, we are assuming that all numbers are distinct. If they are not, we make a convention that among two equal numbers, the one which comes before in the array is smaller. We can also store the index information in the binary tree.

For $j = k+2$ to $n-k$, we will do as follows:

- Insert a_{j+k} in the binary tree.
- If $a_{j+k}, a_{j-k} > med$ then set $M[j]$ as the value of *med*.
- If $a_{j+k}, a_{j-k} < med$ then set $M[j]$ as the value of *med*.
- If $a_{j-k} \geq med$ and $a[j+k] < med$, then set $M[j]$ as the value in the predecessor of *med*. Also update the *med* pointer to its predecessor.
- If $a_{j-k} \leq med$ and $a[j+k] > med$, then set $M[j]$ as the value in the successor of *med*. Also update the *med* pointer to its successor.
- Remove a_{j-k} from the binary tree. Both insertion and removal can be done in $O(\log k)$ time in a balanced binary tree.

Question 4 [7 marks]. We are given n rectangular boxes $B_1, B_2, \ldots, B_n$ with same heights. We are given their widths and lengths. Suppose the box B_i has width w_i and length ℓ_i (assume $w_i \leq \ell_i$) for every $1 \leq i \leq n$. The box B_i fits inside box B_j if and only if $w_i \leq w_j$ and $\ell_i \leq \ell_j$. For each box B_i, we want to output the number of other boxes that can fit into B_i. That is, you have to output n numbers. For simplicity, you can assume that all widths and lengths are distinct numbers.

Design a **divide and conquer** algorithm for this problem. You may want to sort the boxes in some order before running the algorithm. Do a runtime analysis for the complete algorithm.

Full marks for $O(n \log n)$ time algorithm. Only 5 marks for $O(n \log^2 n)$ time algorithm.

Hint: If it helps, you can try to ensure that after a recursive call on the left/right half, you also get that half sorted in some way.

Example:
Input: (5, 7), (2,10), (1, 6), (4, 16)
Output: 1, 1, 0, 2

Answer 4. Sort the boxes in increasing order of lengths. While sorting, also store the original index in the given list of boxes. Let $FitCount$ be an array whose i-th entry is supposed to the number of other boxes that can fit into B_i, for each i. Initially, it is all zeros.

Count-and-Sort:

Input: a list S of boxes sorted in increasing order of lengths.

Output: updates the $FitCount$ so that for each box B_i in S, the entry $FitCount[i]$ is the number of other boxes in S that can fit into B_i. It also sorts S in increasing order of widths.

1. If S has only one box then return.
2. Divide S into two halves (based on length): let S_1 be the first half and S_2 be the second half.
3. Count-and-Sort(S_1)
4. Count-and-Sort(S_2)
5. Count(S_1, S_2)
6. $S \leftarrow$ Merge-width(S_1, S_2).

Here, Merge-width is the standard procedure that takes two lists of boxes sorted in increasing order of widths and outputs the union of the two lists sorted in increasing order of widths. Now, we describe the Count procedure, which will count for every box in S_2, how many boxes in S_1 fit into it, and update $FitCount$ accordingly.

Observe that each box in S_2 has a larger length than each box in S_1. We need to count for every box in S_2, how many boxes in S_1 have smaller width than it. We will traverse over S_1 and S_2, which are sorted in increasing order of widths. We will maintain two pointers: i for S_1 and j for S_2.

Count(S_1, S_2)**:**

Initially $i = 0$
for $j = 0$ to $n/2 - 1$ {
  Keep increasing i till the first point we get $S_2[j].width < S_1[i].width$
  Let k_j be the original index of the box $S_2[j]$.
  $FitCount[k_j] \leftarrow FitCount[k_j] + i$ (because $S_2[j].width$ is larger than exactly i widths in S_1)
}
