- **Due Date: Thursday, February 27th, 23:59**
- **Late submission** will be accepted **without** penalty until **Saturday, March 1st, 23:59**.
- **No submissions will be accepted after March 1st, 23:59.**

Problem 1 (Selection Algorithm, 35 pts)

Recall the deterministic selection algorithm for the median that we saw in class (Lecture 8 and 9, but see lecture note 7). In there, to compute the "approx-median" (element within the middle 40%), we break the elements into small sets of size 5, compute the median for each small set, and then output the median of these $n/5$ medians as the "approx-median".

(a) If instead of breaking into small sets of size 5, we break into small sets of size 3:

i. Will the "approx-median" we get in this way still be in the middle 40%? Find k, such that the "approx-median" we get will be guaranteed to be in the middle k portion. For example, if it is guaranteed to be in the middle 40%, then $k = \frac{2}{5}$. Justify your answer, and k should be as tight (small) as possible.

ii. Will the deterministic selection algorithm be still correct in this case? Briefly justify your answer.

iii. Give a recurrence relation for the runtime of this version of the deterministic selection algorithm. Can we still argue that it is in linear time? Justify your answer.

(b) If instead of breaking into small sets of size 5, we break into small sets of size 7:

i. Will the "approx-median" we get in this way still be in the middle 40%? Find k, such that the "approx-median" we get will be guaranteed to be in the middle k portion. Justify your answer, and k should be as tight (small) as possible.

ii. Will the deterministic selection algorithm be still correct in this case? Briefly justify your answer.

iii. Give a recurrence relation for the runtime of this version of the deterministic selection algorithm. Can we still argue that it is in linear time? Justify your answer.

(c) Any advantages of using 5 as opposed to some other number? In other words, was our choice of 5 arbitrary? (Just expecting a short, one-line answer here)

Solution

(a) Breaking into small sets of size 3

(i) Approximate Median Location. When we divide the n elements into groups of 3, there are about $\frac{n}{3}$ groups. In each group the median is the second smallest element. When we take the median of these medians (call it m), at least half of the group medians are $\leq m$ and at least half are $\geq m$.

$$\text{Number of groups with median that is less than or equal to } m \geq \frac{n}{6}.$$

In any group with median $\leq m$, the smallest and the median (i.e. 2 elements) are $\leq m$. Thus, there are at least

$$2 \cdot \frac{n}{6} = \frac{n}{3}$$

elements that are $\leq m$. A symmetric argument shows that at least $\frac{n}{3}$ elements are $\geq m$. Therefore, the approximate median m is guaranteed to lie between the $\frac{n}{3}$th and the $\frac{2n}{3}$th order statistics – that is, it lies in the middle $\frac{1}{3}$ portion of the array.

(ii) Correctness. The deterministic selection algorithm remains correct when using groups of 3. The key point is that the algorithm's correctness does not depend on having the pivot exactly in the middle, but rather on ensuring that the true median is never discarded during the partitioning step.

When we use groups of 3, we compute the median for each group and then take the median of these medians as the pivot m. Although this guarantees only that m lies between the $\frac{n}{3}$th and the $\frac{2n}{3}$th smallest elements, this is sufficient for correctness. During the partitioning step, the array is divided into elements less than or equal to m and those greater than or equal to m. Since m is not an extreme element, the true median (the $\lceil n/2 \rceil$th smallest element) is guaranteed to lie in one of these two partitions.

Thus, even with groups of 3, the recursive calls always include the true median, and eventually, the algorithm will correctly select it.

(iii) Running Time. We have

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n).$$

At the root level, the cost is $O(n)$. At the next level, the two recursive calls contribute costs proportional to $O\left(\frac{n}{3}\right)$ and $O\left(\frac{2n}{3}\right)$, whose sum is still $O(n)$. We can see that each level of the recursion tree incurs a total cost of $O(n)$.

The height of the recursion tree is determined by the larger subproblem, which is of size $\frac{2n}{3}$. The recursion terminates when the subproblem size becomes constant; solving

$$\left(\frac{2}{3}\right)^h n \approx 1,$$

yields

$$h = \Theta(\log n).$$

Thus, the total running time is

$$T(n) = O(n) \cdot \Theta(\log n) = \Theta(n \log n).$$

Conclusion: Since $T(n) = \Theta(n \log n)$, it does *not* run in linear time.

(b) Breaking into small sets of size 7

(i) Approximate Median Location. When we divide the n elements into groups of 7, there are roughly $\frac{n}{7}$ groups. In a group of 7 (when sorted) the median is the 4th smallest element. Now, consider the groups whose median is $\leq m$ (where m is the median-of-medians). There are at least $\frac{n}{14}$ such groups. In any such group the smallest 4 elements are $\leq m$, so they contribute at least 4 elements each. Thus, there are at least

$$4 \cdot \frac{n}{14} = \frac{2n}{7}$$

elements that are $\leq m$. A symmetric argument shows that at least $\frac{2n}{7}$ elements are $\geq m$. In other words, the pivot m is guaranteed to lie between the $\frac{2n}{7}$th and the $\frac{5n}{7}$th order statistics. Hence, m is in the middle

$$\frac{5n}{7} - \frac{2n}{7} = \frac{3n}{7}$$

portion of the array, so here we have $k = \frac{3}{7}$ (approximately 42.9%).

(ii) Correctness. The deterministic selection algorithm remains correct when using groups of size 7. In this variant, the input is divided into groups of 7 and the median of each group (specifically, the 4th smallest element) is computed. Then, the median of these medians is selected as the pivot m. This pivot is guaranteed to have at least $\frac{2n}{7}$ elements less than or equal to it and at least $\frac{2n}{7}$ elements greater than or equal to it. Consequently, when the array is partitioned around m, the true median (i.e. the $\lceil n/2 \rceil$th smallest element) cannot be discarded—it must reside in one of the two partitions that are recursively processed. Thus, the algorithm will eventually converge to the correct median.

(iii) Running Time Analysis When using groups of size 7, the recurrence becomes

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + O(n).$$

At the top level, the cost is $O(n)$. The recursive calls now split the problem into one subproblem of size $\frac{n}{7}$ and another of size $\frac{5n}{7}$. The sum of the fractions is

$$\frac{1}{7} + \frac{5}{7} = \frac{6}{7}.$$

Summing over all levels, we have

$$T(n) = O(n) \sum_{i \geq 0} \left(\frac{6}{7}\right)^i = O(n) \cdot \frac{1}{1 - \frac{6}{7}} = O(n) \cdot 7 = O(n).$$

Conclusion: The recurrence for groups of 7 solves to $T(n) = O(n)$, so the algorithm still runs in linear time when groups of size 7 are used.

(c) Advantages of Using 5

The choice of 5 is not arbitrary: it is the smallest group size for which the "median-of-medians" selection algorithm guarantees a linear worst-case running time. Using groups of 3 yields only a $\frac{1}{3}$ guarantee (and a recurrence that solves to $\Theta(n \log n)$), while groups of 7 (though still linear) introduce a larger constant in the running time. Using groups of 5 reaches a good balance between having a pivot that is sufficiently "central" and keeping the overhead small.

Problem 2 (Finding Duplicates, 30 pts)

Consider an array A of n elements where each element appears exactly twice in A, e.g. $A = [9, 7, 7, 1, 9, 1, 3, 5, 3, 5]$. For any two elements $A[i]$, $A[j]$ in the array, we may only compare the elements by testing equality, i.e., $A[i] \stackrel{?}{=} A[j]$. With this in mind,

(a) Give an algorithm that returns two indices in A that have the same element using at most $n - 2$ comparisons/equality tests. *Note: there are multiple pairs of these indices, and only outputting one pair is sufficient. For example,* $(1, 5), (2, 3), (4, 6), (7, 9), (8, 10)$ *are all valid outputs of this algorithm on* A.

(b) Prove that your algorithm really needs $n - 2$ comparisons in the worst case (i.e., there are inputs where it uses that many comparisons before terminating). Specifically, give an example of a worst-case input for $n = 10$. Note that this worst-case input only needs to work for *your* algorithm in part (a), not *any* algorithm.

Solution

(a) Algorithm

Consider the following procedure which uses at most $n - 2$ comparisons:

1. **Searching with $A[1]$ as a Candidate:** For $i = 2, 3, \ldots, n - 1$:
   - Compare $A[1]$ with $A[i]$.
   - If $A[1] = A[i]$, output the pair $(1, i)$.

   This loop from $i = 2$ to $i = n - 1$ uses $n - 2$ comparisons.
2. **Final Step:** If no match is found in the previous steps, then by the problem's hypothesis (each element appears exactly twice), the duplicate of $A[1]$ must be in position n. Thus, output the pair $(1, n)$.

(b) Worst-Case Analysis for $n = 10$

To prove that our algorithm uses $n - 2$ comparisons in the worst case, we provide a worst-case input where $A[1]$ does not match any element from $A[2]$ to $A[9]$, forcing the algorithm to make all 8 comparisons.

Consider the following array:

$$A = [1,\ 2,\ 3,\ 4,\ 5,\ 2,\ 3,\ 4,\ 5,\ 1].$$

Here, $A[1] = 1$ and its duplicate appears only at $A[10]$. The algorithm proceeds as follows:

1. **Step 1:** Compare $A[1]$ with $A[2], A[3], \ldots, A[9]$ (8 comparisons total) $\rightarrow$ no match found.
2. **Step 2:** Since no match is found, conclude that the duplicate of $A[1]$ is at $A[10]$ and output the pair $(1, 10)$.

Thus, the worst-case number of comparisons is 8 for $n = 10$.

Problem 3 (Minimum and Maximum — Lower Bound, 35 pts)

Suppose we have an array A of n distinct integers, and we want to find both the minimal and the maximal number. In Lecture 8 (see lecture note 7 though) we saw how to do this using $\frac{3n}{2}+O(1)$ comparisons, as opposed to the naïve algorithm that uses $2(n-1)$ comparisons by making two separate passes to find the minimum and the maximum separately. Here, we will show that this is actually optimal in the comparison model by proving a lower bound of $\frac{3n}{2} - O(1)$ (and therefore the complexity of this problem is $\frac{3n}{2} + \Theta(1)$).

For the lower bound, suppose there is an *arbitrary* comparison-based algorithm that finds the minimum and maximum of A. Suppose we run this algorithm on some input and consider the situation after some step of the algorithm. Let us say that the element of the array has *lost* comparison if it was compared with some other element and turned out to be smaller. Let us say that the element has *won* comparison if it was compared to another element and it turned out to be larger. Without loss of generality, we will assume all elements are distinct, so the comparison result is either "<" or ">".

Consider the following three parameters:

1. Let a be the number of elements of the array that have not been compared at all.
2. Let b be the number of elements that have both lost and won comparisons.
3. Let c be the number of elements that have either only lost or only won comparisons (at least 1 comparison).

(a) What are the values of a, b, and c at the beginning of the algorithm — before any comparisons are made?

(b) What are the values of a, b, and c at the end of the algorithm — after the algorithm successfully found the largest and the smallest numbers?

(c) What relationship do a, b, and c fulfill throughout the execution of the algorithm?

(d) Describe an adversarial strategy for answering comparison queries (consistent with some input) that makes the parameters a, b, and c change as little as possible through the execution of the algorithm. Notice that you cannot control which elements are compared each time; you can only control the outcome. So you might want to produce the comparison outcome depending on the inputs. Deduce the $\frac{3n}{2} - O(1)$ lower bound in the comparison model.

(*Hint: Consider you playing the role of the comparison oracle, and your friend playing the role of an algorithm. When your friend gives two elements for you to compare, you can produce an arbitrary answer, either "<" or ">" (you should imagine the case where there are no actual values associated with the elements, so you can answer arbitrarily). Think about which answer should you give, if you want your friend to take the longest time (i.e. to make the most number of queries) possible. You might want to break into cases where each of the elements has won/lost/both/neither.*)

Solution

(a) Initial Values:
Before any comparisons are made, every element has not been compared. Thus, we have

$$a = n, \quad b = 0, \quad c = 0.$$

(b) Final Values:
When the algorithm terminates, it must have correctly identified the minimum and the maximum. Note that:

- The *minimum* element must have lost at least one comparison but never won any.
- The *maximum* element must have won at least one comparison but never lost any.
- Every other element must have lost at least once (so it cannot be the maximum) and won at least once (so it cannot be the minimum), meaning they have been *eliminated* as candidates for either extreme.

Thus, at termination:

$$a = 0, \quad c = 2, \quad b = n - 2.$$

(c) Invariant Relationship Throughout Execution:
At any point in the algorithm, we have the invariant

$$a + b + c = n.$$

Moreover, the algorithm must eventually eliminate all but two elements (the minimum and maximum), i.e., it must achieve $b = n - 2$. An important observation is that in each comparison, the adversary can force the parameters to change so that at most one new element is moved into the eliminated group (increasing b by at most 1).

(d) Adversarial Strategy and the Lower Bound:
The adversary's goal is to answer comparisons in such a way that the parameters a, b, and c change as little (slow) as possible. The strategy is as follows:

1. **When both elements are in a:** The adversary can answer arbitrarily. This moves both elements from a to c (one becomes a "winner-only" candidate and the other a "loser-only" candidate). This requires one comparison, reducing a by 2 and increasing c by 2.
2. **When one element is in a and the other in c:** The adversary answers so that the a-element acquires the same mark as the c-element. For instance, if the c-element is currently only a winner (a candidate for maximum), then declare that the a-element loses. This way, the c-element remains "half-decided" (staying in c) and the a-element joins c without forcing any element into b.
3. **When both elements are in c:**
    - If both have the same mark (either both only winners or both only losers), the adversary can choose the outcome so that one element remains with only one mark. This avoids giving any element the missing mark and thus prevents moving an element into b.

- If the two elements have opposite marks (one is a candidate for the maximum and the other for the minimum), then regardless of the answer, one of the elements will obtain the missing mark and will move into b (i.e., be eliminated). In this case, the adversary accepts that one elimination (an increase in b) is unavoidable.

Note that it is not necessary to consider cases where one or both elements are already in b because once an element is in b, it has already been eliminated as a candidate for either the minimum or maximum. The only comparisons that affect progress toward isolating the two extreme elements are those that move an element from a to c or from c to b. Comparisons involving an element in b do not contribute any further to eliminating candidates, and thus they do not affect the overall lower bound calculation.

Since initially all elements are in a, we must first "open" them up by making about $\frac{n}{2}$ comparisons (pairing the n elements to move them from a to c). Then, to eliminate the remaining $n-2$ elements (i.e., to have $b = n-2$ at termination), the algorithm must perform at least $n-2$ additional comparisons (each of which, in the worst case, eliminates at most one element). Therefore, the total number of comparisons is at least

$$\frac{n}{2} + (n-2) = \frac{3n}{2} - 2.$$

Thus, the lower bound on the number of comparisons is

$$\frac{3n}{2} - O(1).$$

Since there exists an algorithm that finds the minimum and maximum in $\frac{3n}{2} + O(1)$ comparisons, we conclude that the optimal complexity in the comparison model is

$$\frac{3n}{2} + \Theta(1).$$

□