- **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)

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.

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.*)