15-451/651 Algorithm Design & Analysis, Spring 2026

Oral Homework #4 Solutions

1 Problem 1

1. Fix any deterministic algorithm, and let an adversary answer `false` to every query SAME(i, j).

   Suppose the algorithm stops after asking fewer than $\binom{n}{2}$ queries. Then there is some unordered pair $\{p, q\}$ that was never queried. The answers seen so far are consistent with both of the following inputs:

   (a) all n numbers are distinct;

   (b) $A[p] = A[q]$, and every other pair of positions contains different values.

   Indeed, every queried pair was answered `false`, and in the second input we may choose the values so that the only equal pair is (p, q).

   These two inputs require different correct outputs, but the algorithm sees exactly the same transcript on both of them. So any correct algorithm must, in the worst case, query every pair. Hence the number of calls to SAME is at least

   $$\binom{n}{2} = \Omega(n^2).$$

2. This uses the pair-cancellation idea (which is standard and good to know!). If we delete a pair of different elements from the array, then any majority element that existed before the deletion is still a majority afterwards: if a value x occurs more than $m/2$ times in a multiset of size m, then after removing one copy of x and one copy of a different value, x still occurs more than $(m-2)/2$ times. Thus repeated cancellation of unequal pairs can never destroy a true majority element.

   This algorithm as described above can be implemented using SAME-queries. Maintain a candidate index c and a counter s. Process the array from left to right:

   (a) If $s = 0$, set $c \leftarrow i$ and $s \leftarrow 1$.

   (b) Otherwise query SAME(i, c). If it returns true, increment s. If it returns false, decrement s.

   The invariant maintained by this algorithm is: after processing $A[1], \ldots, A[i]$, those elements can be partitioned into

   - s unpaired copies of the value $A[c]$, and
   - some number of disjoint pairs of unequal elements.

   Therefore, if a majority element exists in the whole array, it cannot be fully canceled, so after the first pass the candidate $A[c]$ must be that majority element.

   A majority need not exist, however, so we must verify. Make one more pass through the array, comparing every position to c with SAME, and count how many matches occur. If the count is greater than $n/2$, then $A[c]$ is a majority element; otherwise there is no majority element.

   The first pass uses $O(n)$ queries, and the verification pass also uses $O(n)$ queries. So the total number of queries is $O(n)$.

2 Problem 2

Assume $2 \le t \le n$, which is the nontrivial regime. (If $t = 1$, then the array is forced to be strictly decreasing, so the sorted permutation is already determined.) For $t \ge 2$, proving $\Theta(n \lg t)$ is the same as proving $\Theta(n(1 + \lg t))$.

Lower bound. For simplicitly, we will assume that $t \mid n$, and let $m = n/t$. We construct a family of inputs with longest increasing subsequence at most t, but with many different correct outputs.

Partition the array positions into m consecutive blocks of length t. The first block contains the t largest values, the second block contains the next t largest values, and so on, with the last block containing the t smallest values. Inside each block, the values may be arranged in an arbitrary permutation.

There are $(t!)^m$ such arrays.

Every element in an earlier block is larger than every element in a later block. Therefore an increasing subsequence cannot use elements from two different blocks: once it moves to a later block, all values become smaller. So every increasing subsequence is contained in a single block, and hence has length at most t.

Since the values are all distinct, each different sequence needs to be rearranged by a different permutation to become sorted. Hence any comparison-based sorting algorithm for this promise class must distinguish at least $(t!)^m$ possible outputs, so its comparison tree must have depth at least $m \lg(t!)$.

It remains to bound $\lg(t!)$ from below. Since at least $t/2$ factors in $t!$ are at least $t/2$, we have

$$\lg(t!) \geq \frac{t}{2} \lg(t/2) = \Omega(t \lg t).$$

Therefore the number of comparisons is at least $m \cdot \Omega(t \lg t) = \Omega(n \lg t)$.

Upper bound. We partition the input into at most t decreasing subsequences, then merge them.

The partition is done with a particular version of the LIS algorithm. Process the array from left to right, maintaining piles whose top values are increasing from left to right. When a new element $A[i]$ arrives, place it on the leftmost pile whose current top is larger than $A[i]$; if no such pile exists, start a new pile. Since the pile tops are increasing, the correct pile can be found by binary search among the current pile tops, so each insertion uses $O(\lg t)$ comparisons once we know there are at most t piles.

Each pile is a decreasing subsequence, because every new element placed on a pile is smaller than the previous top of that pile.

Why are there at most t piles? The invariant is that after any prefix of the input, the number of piles equals the LIS length of that prefix:

- an increasing subsequence can use at most one element from each pile, since each pile decreases;
- if an element is placed on pile j, then at that moment it extends an increasing subsequence of length $j-1$, so there is an increasing subsequence of length j.

Hence the number of piles never exceeds the LIS length of the processed prefix, and therefore never exceeds t.

So we obtain a partition of the input into at most t decreasing subsequences using $O(n \lg t)$ comparisons. Reverse each subsequence; then each becomes an increasing list.

Finally, merge these at most t sorted lists. It's known (by e.g. the mergesort algorithm) that one can merge two sorted lists in a linear number of queries. By merging in pairs, the total cost is $O(n \lg t)$, because there are $O(\lg t)$ recursive layers, each costing $O(n)$.

Combining the two steps, the whole sorting algorithm uses $O(n \lg t) = O(n(1 + \lg t))$ comparisons. Combining the lower and upper bounds, we conclude that the comparison complexity is $\Theta(n(1 + \lg t))$.

3 Problem 3

Write the horizontal bar as

$$\{(h, j) : a \leq j \leq b\}, \qquad b - a + 1 \geq 2,$$

and the vertical bar as

$$\{(i, c) : u \leq i \leq d\}, \qquad d - u + 1 \geq 2.$$

Lower bound. It is enough to count a large subset of valid configurations. Let $m = \lfloor n/4 \rfloor$. Consider only configurations in which

- the horizontal bar lies in one of the first m rows, and
- the vertical bar lies entirely in one of the last m rows.

Then the two bars are separated by at least $2m$ rows, so they certainly do not touch, regardless of their columns.

Within this restricted family:

- there are m choices for the horizontal row h;
- there are $\binom{n}{2}$ choices for the horizontal endpoints $a < b$;
- there are n choices for the vertical column c;
- there are $\binom{m}{2}$ choices for the vertical endpoints $u < d$ inside the last m rows.

So the number of valid outputs in this subfamily is

$$m\binom{n}{2} \cdot n\binom{m}{2} = \Theta(n^6).$$

Each query has only two possible answers, so a decision tree of depth q has at most 2^q leaves. Hence we must have

$$2^q \geq \Theta(n^6),$$

which gives

$$q \geq 6 \lg n - O(1).$$

Upper bound. We show that $6 \lg n + O(1)$ rectangle queries suffice. Every "search" below is an ordinary binary search on a monotone predicate, so each search costs $\lg n + O(1)$ queries.

Step 1: find the first occupied row. Binary search on prefixes of rows to find

$$T = \min\{i : \text{row } i \text{ contains a 1}\}.$$

Step 2: inspect row T. Binary search on row T to find its leftmost occupied column x.

Now make one extra query asking whether row T contains any 1 in columns $x+1, \ldots, n$.

1. **If the answer is no, then row T has exactly one 1.** So T cannot be the horizontal bar row, because the horizontal bar has length at least 2. Therefore (T, x) is the top cell of the vertical bar, so

   $$u = T, \qquad c = x.$$

   Now binary search down column c to find the bottom endpoint d of the vertical bar.

   To find the horizontal bar, first choose one side of column c on which the horizontal bar definitely appears: query the rectangle consisting of all cells strictly left of column c. If the answer is yes, use the left side; otherwise use the right side. On the chosen side, only the horizontal bar can contribute any 1's, so binary search over rows to find its row h.

   Finally, use one constant-time query on row h to see whether the horizontal bar also extends to the other side of column c. Then do two binary searches on row h to recover its endpoints:

   - if the bar stays entirely on one side of c, search on that side for its left and right endpoints;

- if it extends to both sides of c, search on the left side for a and on the right side for b.

In this case we used exactly six searches:

$$T,\ c,\ d,\ h,\ a,\ b.$$

2. **If the answer is yes, then row T contains the horizontal bar.** Binary search on row T to find its rightmost occupied column y.

   Now query row $T+1$ restricted to columns $x,\dots,y$.

   (a) **If that answer is no,** then row T contains only the horizontal bar. So

   $$h = T, \qquad a = x, \qquad b = y.$$

   Moreover, below row T only the vertical bar remains. Hence we can binary search in the subgrid of rows $T+1,\dots,n$ to find the vertical column c, and then binary search in that column to find u and d.
   Again this is six searches total:

   $$h,\ a,\ b,\ c,\ u,\ d.$$

   (b) **If that answer is yes,** then the vertical bar already occupies row $T+1$. Because the two bars do not touch, that vertical column cannot lie strictly between the endpoints of the horizontal segment. So among the occupied cells of row T, the vertical column must be one of the two extremes x or y.
   A single extra query to the cell $(T+1, x)$ tells which one:

   - if it is 1, then $c = x$;
   - otherwise $c = y$.

   Once c is known, one endpoint of the horizontal bar is already known (the opposite extreme), and one binary search on row T finds the other endpoint. Since $u = T$, one more binary search down column c finds d.
   So this subcase uses at most five searches.

In every branch, the number of searches is at most six, plus only $O(1)$ extra case-distinguishing queries. Therefore the total number of rectangle queries is

$$6\lg n + O(1).$$

Combining the lower and upper bounds, the optimal query complexity is

$$6\lg n \pm \Theta(1).$$