PSO 2 — Week 4: Solutions

CS381, Fall 2026

Problem 1

(a) Point of maximum overlap

Turn each interval into two events, $(\ell_i, +1)$ and $(r_i, -1)$. Sort by coordinate, and at a coordinate where both kinds occur, process all $+1$ events before all -1 events. Sweep with a counter, recording the largest value it ever reaches immediately after an increment, along with the coordinate at which that happens.

Correctness. For a point x, the number of intervals containing x is $|\{i : \ell_i \le x\}| - |\{i : r_i < x\}|$, since the intervals are closed. Under the stated ordering, the counter equals exactly this quantity at the moment the last $+1$ event at coordinate x has been processed and before any -1 event at x. Coverage increases only at left endpoints, so its maximum over $\mathbb{R}$ is attained at some ℓ_i, and sampling the counter after each increment examines every such point. □

Time. $O(n \log n)$ to sort the $2n$ events, $O(n)$ to sweep.

The tie rule matters. Take $[1, 2]$ and $[2, 3]$, which share the point 2, so the answer is 2. Processing -1 before $+1$ at coordinate 2 gives counter values $1, 0, 1, 0$ and reports 1. Closed intervals that touch at a point do overlap there, and the $+1$-first rule is what encodes that.

(b) Counting intersecting pairs

Count the disjoint pairs instead, and subtract from $\binom{n}{2}$.

Intervals i and j are disjoint exactly when $r_i < \ell_j$ or $r_j < \ell_i$, and both cannot hold at once: $r_i < \ell_j$ and $r_j < \ell_i$ would give $\ell_i \le r_i < \ell_j \le r_j < \ell_i$. So

$$\#\{\text{disjoint pairs}\} = \sum_{i=1}^{n} |\{\, j : \ell_j > r_i \,\}|,$$

with each disjoint pair counted once, by whichever of its two intervals ends first. The term $j = i$ never contributes, since $\ell_i \le r_i$.

Sort the array of left endpoints once, then for each i binary search r_i in it to get $|\{j : \ell_j > r_i\}|$. The answer is $\binom{n}{2}$ minus the sum. Sorting is $O(n \log n)$ and the n searches are $O(n \log n)$.

(c) Longest nested chain

Sort the intervals by ℓ descending and let R be the resulting sequence of right endpoints. Then the longest nested chain has the same length as the longest strictly increasing subsequence of R. Endpoints are distinct, so all comparisons below are strict.

Given a chain $i_1 \subsetneq \cdots \subsetneq i_t$, we have $\ell_{i_1} > \cdots > \ell_{i_t}$, so these intervals occur in this order in the sorted sequence, and $r_{i_1} < \cdots < r_{i_t}$, so their right endpoints form an increasing subsequence of length t.

Conversely, take positions $p_1 < \cdots < p_t$ with $R[p_1] < \cdots < R[p_t]$. Their position order means $\ell_{p_1} > \cdots > \ell_{p_t}$, so each consecutive pair satisfies $\ell_{p_{q+1}} < \ell_{p_q}$ and $r_{p_q} < r_{p_{q+1}}$, that is $p_q \subsetneq p_{q+1}$.

Containment is transitive, so the whole sequence is a nested chain of length t. The two maps are inverse to each other, so the maxima agree. □

Proving only the first direction would show that the LIS is at least the longest chain, which is the wrong inequality for an algorithm that returns the LIS.

Compute the LIS in $O(n \log n)$ by patience sorting: scan R maintaining an array *tails*, where *tails*$[q]$ is the smallest possible last element of an increasing subsequence of length q; for each value, binary search for the first entry that is at least as large and overwrite it, appending when there is none. The answer is the final length of *tails*. With the initial sort, $O(n \log n)$ overall.

Problem 2

(a) Inversions

Have each merge-sort call return its sorted subarray together with its inversion count. Pairs $i < j$ are of three kinds: both in the left half, both in the right half, or split across. The recursion handles the first two. For the third, whenever the merge outputs an element of the right half, every element still remaining in the left half has a smaller index and a larger value, so add the number remaining.

Each split pair is counted once, when its right element is output, and sorting within a half changes neither which pairs are split nor which side an element is on. $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.

(b) Significant inversions

The merge compares $L[i]$ with $R[j]$, but the predicate is $L[i] > 3R[j]$. The qualifying elements of L form a suffix cut at a different threshold from the merge's own comparison, and that cut does not advance in step with the merge pointers, so the count of remaining left elements at the time $R[j]$ is output is not the quantity wanted.

Run a separate two-pointer pass before merging:

```
count := 0; i := 1
for j = 1 to |R|:                  // R sorted ascending
    while i <= |L| and L[i] <= 3*R[j]: i := i + 1
    count := count + (|L| - i + 1)
merge L and R as usual
```

For a fixed $R[j]$ the qualifying elements of L are exactly the suffix $L[i..|L|]$, since L is sorted, and every element of L has an original index below every element of R. The thresholds $3R[j]$ are nondecreasing, so i never moves backward and the pass is $O(|L| + |R|)$. This adds a second linear pass per level rather than a second recursion, so the bound stays $\Theta(n \log n)$.

(c) D-local inversions

Sort the values once and replace each by its rank in $1..n$. This step is where the size of the values matters: the algorithm below indexes an array by value, which is impossible when the values are unbounded in terms of n, and rank compression costs $O(n \log n)$, which is already the target bound.

Sweep with a Fenwick tree holding exactly the window $A[\max(1, j - D) .. j - 1]$:

```
for j = 1 to n:
```

```
w := current window size
answer += w - BIT.query(rank(A[j]))     // window elements greater than A[j]
BIT.insert(rank(A[j]))
if j - D >= 1: BIT.delete(rank(A[j-D]))
```

Each element is inserted and deleted once and each of the n queries costs $O(\log n)$, so the total is $O(n \log n)$ for every D. Taking $D = n - 1$ recovers the count from part (a).

Why the divide and conquer does not adapt. The merge rule in (a) is licensed by a property of the split rather than of the elements: every left-half element has an index below every right-half element, so the number remaining in L is a correct count whichever elements those happen to be. A window constraint makes the admissible partners of an element depend on that element's own index, and after a half has been sorted by value its elements are no longer in index order, so the admissible partners are an index interval that is not a contiguous block of the sorted array.

Why the block repair miscounts. Every D-local pair does lie within some adjacent block pair, so nothing is missed, but two adjacent blocks span $2D$ consecutive positions, so the count also includes pairs at distances between $D+1$ and $2D-1$. With $D = 2$ and blocks $\{A[1], A[2]\}$ and $\{A[3], A[4]\}$, the pair $(1, 4)$ sits at distance 3 and is counted. Removing the surplus requires counting pairs at distance in $(D, 2D)$, which is another instance of the original problem.