

CS3230 ASSIGNMENT 03: CORRECTNESS AND DIVIDE-AND-CONQUER

Question 1. There are $n \geq 1$ bunkers in a line, indexed $1..n$. A scan query chooses integers $i \leq j$ and returns

$$\text{SCAN}(i, j) = \begin{cases} \text{YES} & \text{if fighter is in one of } i, i+1, \dots, j, \\ \text{NO} & \text{otherwise.} \end{cases}$$

You have two devices, and each day at exactly 6AM you may issue two scans simultaneously.

(a) *Fighter does not move.*

Algorithm (4-way shrinking with up to two intervals of candidates). Maintain the *candidate set* S of bunkers still consistent with all past answers. We will represent S as a union of at most two disjoint intervals. Initially,

$$S = [1, n].$$

Given a current candidate set S (union of at most two intervals), let $N = |S|$ be its size. Choose $t = \lfloor N/4 \rfloor$ (if $N < 4$, handle as a base case by brute force using scans on singletons/pairs). Let S be listed in increasing order as positions $p_1 < p_2 < \dots < p_N$.

Define two scan-intervals using these order statistics:

$$I_1 := [p_{t+1}, p_{3t}], \quad I_2 := [p_{2t+1}, p_{4t}].$$

(If $4t < N$, then p_{4t} exists and I_2 ends at p_{4t} ; the remaining positions p_{4t+1}, \dots, p_N are in the “outside” region.)

At 6AM, issue the two scans on (I_1, I_2) , obtaining a 2-bit outcome $(b_1, b_2) \in \{\text{YES, NO}\}^2$.

Update S to the subset consistent with the outcome, i.e.:

$$S \leftarrow \begin{cases} (I_1 \setminus I_2) & (b_1 = \text{YES}, b_2 = \text{NO}), \\ (I_1 \cap I_2) & (b_1 = \text{YES}, b_2 = \text{YES}), \\ (I_2 \setminus I_1) & (b_1 = \text{NO}, b_2 = \text{YES}), \\ (S \setminus (I_1 \cup I_2)) & (b_1 = \text{NO}, b_2 = \text{NO}). \end{cases}$$

Note that $S \setminus (I_1 \cup I_2)$ is the union of (at most) the “left tail” and “right tail”, hence representable as at most two intervals. Repeat until $|S| = 1$ (then you know the bunker).

Correctness.

Solution. We maintain the invariant that S is exactly the set of bunkers consistent with all scan answers so far.

Initially $S = [1, n]$, which is correct.

On each day we ask membership questions for I_1 and I_2 . Each of the four outcomes corresponds exactly to one of the four disjoint regions in the Venn partition of S :

$$I_1 \setminus I_2, \quad I_1 \cap I_2, \quad I_2 \setminus I_1, \quad S \setminus (I_1 \cup I_2).$$

The fighter is in exactly one bunker, hence exactly one region. Updating S to that region preserves the invariant.

When $|S| = 1$, the invariant implies the unique element of S is the fighter’s bunker, so the algorithm returns the correct bunker. \square

Day complexity.

Solution. Let N_d be the candidate set size after day d updates (with $N_0 = n$). By construction, the four regions have sizes at most $\lceil N_d/4 \rceil$ (up to $\pm O(1)$ due to floors and leftover $N - 4t$ items). Therefore,

$$N_{d+1} \leq \left\lceil \frac{N_d}{4} \right\rceil + O(1).$$

Hence after $D = \lceil \log_4 n \rceil + O(1)$ days, $N_D = O(1)$, and a constant number of additional days suffices to pin down the exact bunker (e.g. scanning singletons/pairs). Thus the number of days is

$$D = \lceil \log_4 n \rceil + O(1).$$

Near-optimality: each day yields only 4 possible outcomes, so distinguishing among n bunkers needs at least $\lceil \log_4 n \rceil$ days information-theoretically. Therefore this is optimal up to an additive constant. \square

(b) *Fighter may move to an adjacent bunker at 7AM each day.* Now the fighter's position at 6AM on day $d + 1$ may differ from day d by at most 1.

Algorithm (shrink-then-expand). We again maintain a candidate set S_d of positions the fighter could be in at 6AM of day d , represented as a union of at most two disjoint intervals.

- (1) Initialize $S_0 = [1, n]$.
- (2) On day d , run the same two-scan 4-way partition procedure as in part (a) but restricted to S_d ; let S'_d be the region consistent with the two scan answers.
- (3) Account for movement at 7AM: define

$$\text{EXPAND}(S'_d) := \{x \in [1, n] \mid \exists y \in S'_d \text{ with } |x - y| \leq 1\}.$$

Set $S_{d+1} := \text{EXPAND}(S'_d)$.

- (4) Stop once $|S_d|$ is a small constant (say ≤ 4). Then, in $O(1)$ more days, identify the exact bunker at 6AM by scanning individual bunkers (and immediately destroy it before 7AM).

Note: expanding a union of at most two intervals yields a union of at most two intervals after merging overlaps.

Correctness.

Solution. By induction on days, we show:

Invariant: S_d equals exactly the set of bunkers where the fighter could be at 6AM on day d consistent with all scans so far and the movement rule.

Base ($d = 0$). Before any scans, the fighter can be anywhere in $[1, n]$, so $S_0 = [1, n]$ is correct.

Step. Assume the invariant holds for S_d . The two scans partition S_d into four disjoint regions as in part (a). The fighter's true position at 6AM day d lies in exactly one region, and the scan answers identify that region uniquely. Thus S'_d is exactly the set of possible positions at 6AM day d after incorporating today's scan answers.

At 7AM the fighter moves at most one step. Therefore the set of possible 6AM positions on day $d + 1$ is precisely all positions within distance 1 of some $y \in S'_d$, i.e. $\text{EXPAND}(S'_d)$. Hence S_{d+1} satisfies the invariant.

When $|S_d|$ is reduced to a constant, scanning singletons over a constant number of days identifies the exact 6AM position on some day; destroying that bunker before 7AM succeeds. \square

Day complexity.

Solution. Let $N_d = |S_d|$. The 4-way partition step gives a consistent region S'_d of size at most $\lceil N_d/4 \rceil + O(1)$. Then expanding by distance 1 increases the size by at most a constant: since

S'_d is a union of at most two intervals, expanding each interval increases its length by at most 2, so the total increase is at most 4 (after merging overlaps). Thus

$$N_{d+1} \leq \left\lceil \frac{N_d}{4} \right\rceil + O(1) + 4.$$

This recurrence reaches $N_d = O(1)$ after

$$D = \lceil \log_4 n \rceil + O(1)$$

days, after which a constant number of additional days suffices to identify the exact bunker. Therefore, the total number of days is

$$D = \lceil \log_4 n \rceil + O(1).$$

□

Question 2. You are given two sorted arrays $A[1..m]$ and $B[1..n]$ with all $m + n$ numbers distinct, and an integer k with $1 \leq k \leq m + n$. Find the k -th smallest among all elements in time proportional to $\log k$.

Algorithm (discard-by-halves).

```
KTH(A[1..m], B[1..n], k):
    if m = 0: return B[k]
    if n = 0: return A[k]
    if k = 1: return min(A[1], B[1])

    i = min(m, floor(k/2))
    j = min(n, floor(k/2))

    if A[i] < B[j]:
        // discard A[1..i]
        return KTH(A[i+1..m], B[1..n], k - i)
    else:
        // discard B[1..j]
        return KTH(A[1..m], B[j+1..n], k - j)
```

Correctness.

Solution. We prove correctness by showing each recursive step preserves the identity of the k -th smallest element.

If $A[i] < B[j]$, then there are at least i elements in $A \cup B$ that are $\leq A[i]$ (namely $A[1..i]$), and also $A[i]$ is smaller than $B[j]$, hence smaller than at least j elements of B beyond $B[1..j]$ as well. Crucially, among the combined arrays, the k -th smallest element cannot lie in $A[1..i]$ because even if *all* of $B[1..j]$ were smaller, $A[1..i]$ still contribute i elements that occur before any element strictly larger than $A[i]$, and removing $A[1..i]$ reduces the rank we seek by exactly i . Thus the k -th smallest of (A, B) equals the $(k - i)$ -th smallest of $(A[i+1..m], B)$.

The case $A[i] > B[j]$ is symmetric, discarding $B[1..j]$ and seeking rank $(k - j)$.

Base cases handle exhaustion of one array or $k = 1$. Therefore the algorithm returns the correct k -th smallest element. □

Running time.

Solution. At each recursive call, k decreases to $k - i$ or $k - j$, where $i, j \geq \lfloor k/2 \rfloor$ unless capped by array sizes; in any case, k decreases by at least $\lfloor k/2 \rfloor$, so the new k is at most $\lceil k/2 \rceil$. Therefore, the recursion depth is $O(\log k)$, and each step does $O(1)$ work besides recursion. Hence the runtime is $\Theta(\log k)$. □

Question 3. Consider the given sorting algorithm (Bubble Sort). Provide invariants and prove it sorts.

(a) *Invariant 1 (outer loop).*

Invariant 1. At the beginning of the outer loop iteration with index i ($1 \leq i \leq n - 1$):

$A[n-i+2], A[n-i+3], \dots, A[n]$ are the $(i-1)$ largest elements of the array and are in increasing order.

Equivalently, the suffix $A[n - i + 2..n]$ is sorted and every element in this suffix is larger than every element in the prefix $A[1..n - i + 1]$.

(b) *Invariant 2 (inner loop).*

Invariant 2. At the beginning of the inner loop iteration with index j ($1 \leq j \leq n - i$):

$$A[j] = \max\{A[1], A[2], \dots, A[j]\} \quad (\text{within the current outer-loop pass}).$$

In words: after completing comparisons/swaps up to position $j - 1$, the maximum among the first j positions has “bubbled” to index j .

(c) *Proof using the invariants.*

Solution. **Invariant 1 initialization.** For $i = 1$, the suffix $A[n - i + 2..n] = A[n + 1..n]$ is empty, so it vacuously contains the 0 largest elements in sorted order. Thus Invariant 1 holds initially.

Invariant 2 initialization. At the start of the inner loop, $j = 1$. Then $A[1] = \max\{A[1]\}$, so Invariant 2 holds.

Invariant 2 maintenance. Assume at the beginning of some inner iteration j we have $A[j] = \max\{A[1..j]\}$. The algorithm compares $A[j]$ and $A[j + 1]$:

- If $A[j] \leq A[j + 1]$, no swap occurs, and then $A[j + 1]$ is the maximum of $A[1..j + 1]$ because it is at least $A[j] = \max(A[1..j])$.
- If $A[j] > A[j + 1]$, a swap occurs; after swapping, $A[j + 1]$ becomes the old $A[j]$, which equals $\max(A[1..j])$ and is also $> A[j + 1]$ (old), hence it is $\max(A[1..j + 1])$.

So Invariant 2 holds for the next inner iteration $(j + 1)$.

Consequence at termination of the inner loop. When the inner loop finishes, $j = n - i$. By Invariant 2 (applied iteratively), we get

$$A[n - i + 1] = \max\{A[1], A[2], \dots, A[n - i + 1]\},$$

so the largest element of the unsorted prefix has been placed at position $n - i + 1$.

Invariant 1 maintenance. Assume Invariant 1 holds at the start of outer iteration i . By the consequence above, after completing the inner loop, the element placed at $A[n - i + 1]$ is the largest among the remaining unsorted prefix $A[1..n - i + 1]$. Therefore, the suffix $A[n - i + 1..n]$ now contains the i largest elements in increasing order: the newly placed $A[n - i + 1]$ is \leq every element in the previous suffix $A[n - i + 2..n]$ (since those were the $(i - 1)$ largest overall). Thus Invariant 1 holds at the beginning of the next outer iteration $i + 1$.

Termination. The outer loop terminates after $i = n - 1$. Then Invariant 1 implies the suffix of length $n - 1$ (and thus the whole array) is sorted in increasing order. Hence the algorithm sorts A . \square