PSO 2 — Week 3: Solutions

CS381, Fall 2026

Problem 1

(a) When BFS is correct

The algorithm is correct on every min-heap of n distinct keys if and only if $k = 1$ or $n \leq 2$.

If $k = 1$, the first node dequeued is the root, which is the minimum. If $n \leq 2$, the BFS order is the array in index order, which is sorted for a 1- or 2-node heap.

For $n \geq 3$ and $k \geq 2$, start from $A = [1, 2, \ldots, n]$ in array order. This is a valid min-heap, since $A[i] = i < 2i \leq A[2i], A[2i+1]$. Modify it as follows.

- $2 \leq k \leq n-1$: swap the values at positions k and $k+1$. Position k now holds $k+1$, whose parent $\lfloor k/2 \rfloor$ holds a smaller value and whose children hold values $\geq 2k > k+1$ (using $k \geq 2$). Position $k+1$ holds k, whose parent holds $\lfloor (k+1)/2 \rfloor < k$ and whose children hold values $\geq 2k+2$. The two positions are not in an ancestor relation since $2k \geq k+2$. BFS dequeues $k+1$ at step k, but the k-th smallest is k.

- $k = n$: swap the values at positions $n-1$ and n. Neither has children, since those would sit at index $\geq 2n-2 > n$, and both parents hold values $\lfloor n/2 \rfloor < n-1$. BFS dequeues $n-1$ at step n, but the n-th smallest is n.

The smallest failing instance is $n = 3$, $k = 2$: the heap $[1, 3, 2]$ dequeues $1, 3, 2$, so BFS returns 3 while the second smallest is 2. $\square$

(F1) bounds the depth at which the k-th smallest can sit, but the heap property says nothing about the order of nodes at equal depth or in different subtrees, which is what BFS would need.

(b) d-ary heaps

A complete d-ary heap on n nodes has height $\Theta(\log n / \log d)$.

Insert sifts the new key up, comparing it with its parent once per level: $O(\log_d n) = O(\log n / \log d)$.

ExtractMin moves the last key to the root and sifts down. At each level it must find the smallest of up to d children before deciding where to go, costing $d-1$ comparisons per level: $O(d \log_d n) = O(d \log n / \log d)$.

So ExtractMin pays for d and Insert does not: sifting up compares against one node per level, sifting down against d.

For I inserts and E ExtractMins the total is

$$\Theta\left((I + Ed) \frac{\log n}{\log d} \right).$$

Write $g(d) = (I + Ed)/\ln d$. Setting $g'(d) = 0$ gives $d(\ln d - 1) = I/E$, so the optimum is $d = \Theta(I/E)$ up to a logarithmic factor. At $d = \Theta(\max(2, I/E))$ the term Ed is $\Theta(I)$, and the total becomes

$$\Theta\left(I \cdot \frac{\log n}{\log(I/E)} \right).$$

Taking d much larger inflates Ed faster than $\log d$ grows; taking it much smaller gives up the denominator.

The optimum is non-constant whenever inserts outnumber extractions by more than a constant factor. Dijkstra on a graph with m edges and n vertices is the standard case: $I = \Theta(m)$ key updates against $E = \Theta(n)$ extractions, so $d = \Theta(m/n)$ gives $O(m \log_{m/n} n)$, which beats the binary heap on dense graphs.

(c) At least k keys below x, in $O(k)$

```
count := 0
Probe(v):
    if v = nil or key(v) >= x: return
    count := count + 1
    if count = k: abort the entire traversal
    Probe(left(v)); Probe(right(v))
Probe(root); return (count >= k)
```

Correctness. Pruning at key$(v) \geq x$ discards nothing, since every descendant of such a v has a larger key. So without the abort the traversal would visit every node with key below x. If $m \geq k$ the counter therefore reaches k and the abort fires, giving the answer yes. If $m < k$ the abort never fires, the traversal finishes with `count` $= m < k$, and the answer is no. □

Time. Every visited node is the root or a child of a visited node with key below x, and at most $\min(m, k)$ visited nodes have key below x, since the traversal stops when the counter hits k. So at most $1 + 2\min(m, k)$ nodes are visited and the cost is $O(1 + \min(m, k)) = O(k)$. □

Pruning is what removes the dependence on n; the abort is what removes the dependence on m. Neither alone suffices: (F2) is $O(1+m)$ with m unbounded in terms of k, and an unpruned traversal is $\Omega(n)$.

(d) Every leaf must be read

Every internal node of a min-heap has a larger child, so the maximum is a leaf, and there are $\lceil n/2 \rceil$ leaves.

Suppose a correct algorithm halts without reading key(u) for some leaf u. Consider two completions that agree on every key the algorithm read: one gives u a key larger than everything read, the other a key smaller than everything read but still above u's parent. Both are valid min-heaps, since u has no children and its only constraint is with respect to its parent. They have different maxima, but the algorithm sees the same transcript in both and returns the same answer, so it is wrong on one of them. Hence every leaf is read, which gives $\Omega(n)$. □

The bound is tight up to the constant: scanning the $\lceil n/2 \rceil$ leaves and taking their maximum is correct, so the problem is $\Theta(n)$.

Problem 2

(a) The two standard bounds

$O(n + k \log n)$: heapify A bottom-up in $\Theta(n)$, using $\sum_h \lceil n/2^{h+1} \rceil O(h) = O(n)$, then perform k ExtractMins at $O(\log n)$ each. They come out in increasing order, so no final sort is needed.

$O(n \log k)$: maintain a max-heap H of size at most k. For each element x, insert if $|H| < k$, and otherwise replace $\max(H)$ by x when $x < \max(H)$. The invariant is that after the prefix $A[1..i]$, H holds the $\min(i, k)$ smallest elements of $A[1..i]$: an x above the current k-th smallest belongs to no prefix's top k, and any smaller x displaces exactly the current maximum. Sorting H at the end costs $O(k \log k)$.

The first bound is never worse. For $2 \le k \le n$ we have $k \log n \le n \log k$, because $k \mapsto k/\log k$ is increasing on $[2, \infty)$, so $\frac{k \log n}{n \log k}$ is maximized at $k = n$, where it equals 1. Hence $n + k \log n = O(n \log k)$.

They differ in space. The first needs all n elements resident and n known in advance, using $\Theta(n)$ working memory; the second is a single pass with $\Theta(k)$ memory. Only the second runs on a stream: log records arriving over a network, a file larger than memory, a sensor feed with no fixed end.

(b) Streaming top-k in $O(n + k \log k)$

```
B := buffer of capacity 2k
for each arriving element x:
    append x to B
    if |B| = 2k:
        p := Select(B, k)                          // k-th smallest, linear time
        B := the k elements of B that are <= p     // one partition pass
if |B| > k: trim B to its k smallest the same way
sort B and output
```

Invariant. After any prefix P of the stream, B contains the $\min(|P|, k)$ smallest elements of P, and $|B| \le 2k$. Appending preserves this. At a prune, B is a superset of the k smallest of P and we keep the k smallest elements of B; each discarded element exceeds all of those, hence exceeds the k-th smallest of P and cannot belong to the k smallest of P. $\square$

Time. Appends cost $O(1)$ each. Each prune costs $O(k)$, since selection and partition are linear in $|B| = 2k$. Counting a prune at every step would give $O(nk)$, but each prune permanently discards k elements and an element is discarded only once, so at most n/k prunes occur in total, for $(n/k) \cdot O(k) = O(n)$. With the final sort the total is $O(n + k \log k)$ in $O(k)$ space.

This is optimal up to the sorting term, since every element must be read. The buffer capacity has to exceed k by a constant factor: at capacity exactly k, every arrival after the first k triggers an $O(k)$ prune that discards a single element, and the cost is $\Theta(nk)$. The slack is what the amortization spends.

(c) Merging k sorted lists

```
H := min-heap of (value, list_id, position)
for each nonempty list L_t: Insert(H, (L_t[1], t, 1))
while H nonempty:
    (v, t, p) := ExtractMin(H)
    output v
    if L_t[p+1] exists: Insert(H, (L_t[p+1], t, p+1))
```

Correctness. At the top of each iteration H holds the smallest not-yet-output element of every list that still has one. This holds initially and is restored after each iteration, since only L_t's frontier

advances. Given it, $\min(H)$ is the smallest not-yet-output element overall: any such element lies at or after its own list's frontier, and its list is sorted, so it is at least that list's representative in H. Each position is inserted once and removed once, so every element is emitted exactly once. $\square$

Time. $|H| \le k$, with n ExtractMins and at most n inserts at $O(\log k)$ each, plus $O(k)$ to build H: total $O(n \log k)$.

Merging left to right instead re-scans the accumulated prefix each time. With k lists of length n/k, step t merges a prefix of length tn/k against n/k new elements, giving $\sum_{t=1}^{k} \Theta(tn/k) = \Theta(nk)$. Balanced pairwise merging avoids the heap entirely and still runs in $O(n \log k)$: $\lceil \log k \rceil$ tournament rounds, each touching every element once.

(d) Lower bound

With k sorted lists of length n/k, the output is determined by the interleaving, and all $n!/((n/k)!)^k$ interleavings are realizable. A comparison-based algorithm is a binary decision tree needing at least that many leaves, so its worst-case depth is at least

$$\log \frac{n!}{((n/k)!)^k} = \Theta(n \log n) - k \cdot \Theta\Big(\frac{n}{k} \log \frac{n}{k}\Big) = \Theta(n \log k),$$

using $\log(m!) = \Theta(m \log m)$. So (c) is optimal. $\square$