HOMEWORK 3

Guidelines:

- Attempt all questions by yourself before discussing with peers, this is practice to strengthen your concepts.
- For coding questions, try writing clean, readable code on paper.
- Since this homework is ungraded, focus on learning rather than using LLMs to generate codes and get answers.
- If you get stuck, you are encouraged to:
  - Post your doubts on the course Slack channel.
  - Visit the TAs during office hours for guidance.

Section A: Multiple Choice Questions (MCQs)

Q1. Suppose you have an array H representing a max-heap of size n. What is the tightest upper bound on the number of leaves in the heap?

a) $\lfloor n/2 \rfloor + 1$
b) $\lceil n/2 \rceil$
c) $n - \lfloor n/2 \rfloor$
d) $\lceil n/2 \rceil + 1$
e) $\lfloor n/2 \rfloor$

Q2. What is the tightest worst-case time complexity to build a Max-Heap from an unsorted array of n elements?

a) O(logn)
b) O(nlogn)
c) O(n)
d) O(n2)

Q3. You have two valid Max-Heaps, H1 of size n and H2 of size m. What is the worst-case time complexity to **merge** the two heaps into a single Max-Heap of size n+m?

a) O(log(n+m))
b) O(n+m)
c) O((n+m)log(n+m))
d) O(min(n,m)log(n+m))

Q4. Consider a Max-Heap where you want to **increase the value** of a key at an arbitrary position i. Which heap maintenance operation is required after the increase?

a) Heapify-Down

b) Heapify-Up

c) Build-Heap

d) Level-Order Traversa

Q5. Given an array A = [10, 5, 8, 3, 2, 7] that is a valid min-heap, which of the following is a possible sequence of elements after deleting the minimum element twice?

a) `[7, 8, 10, 3, 2]`

b) `[7, 8, 10, 2]`

c) `[8, 7, 10, 3, 2]`

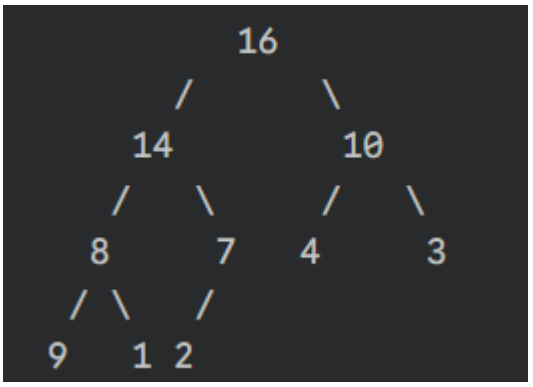
d) `[8, 7, 10]`

e) `[8, 10, 7]`

Section B: Short Questions

Q1. Starting with an empty array, insert the following numbers one by one to form a max-heap: [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]. Show the final heap as a complete binary tree and as an array.

Ans) [16, 14, 10, 8, 7, 4, 3, 9, 1, 2],



Q2. Given an array representing a binary tree, A = [10, 8, 9, 7, 5, 6, 3], determine whether it is a valid max-heap. If it is not, identify the first node (parent or child) that violates the max-heap property. Justify your answer by referencing the indices of the array.

Ans) Yes, it is a valid min heap.All parent nodes in the tree satisfy the max-heap property. Therefore, the array is a valid max-heap.

Q3. Design an algorithm(pseudo code) that merges k sorted lists into a single sorted list. Your solution should use a min-priority queue. Analyze the time and space complexity of your algorithm. Consider a scenario where the total number of elements across all lists is N and the number of lists is k.

Ans)This algorithm works by keeping a "pointer" to the smallest *available* element from each of the k lists. A min-priority queue is the perfect data structure to efficiently track which of these k elements is the *overall* minimum.

```
function mergeKLists(lists):
    // lists is an array of k sorted lists
    // N is the total number of elements
    // k is the number of lists

    // 1. Create the result list and the min-heap
    result = new empty list
    // The heap will store tuples of: (value, list_index, element_index)
    min_heap = new MinPriorityQueue()

    // 2. Initialize the heap with the first element from each non-empty list
    for i from 0 to k-1:
        if lists[i] is not empty:
            value = lists[i][0]
            // Insert (value, which list it's from, its index in that list)
            min_heap.insert( (value, i, 0) )

    // 3. Process the heap until it is empty
    while min_heap is not empty:
        // 3a. Extract the smallest element from the heap
        (value, list_idx, elem_idx) = min_heap.extractMin()

        // 3b. Add this smallest element to our result
        result.append(value)

        // 3c. Get the *next* element from the *same list*
```

```
    next_elem_idx = elem_idx + 1


    // 3d. If that list still has elements, add the next one to the heap
    if next_elem_idx < length(lists[list_idx]):
        next_value = lists[list_idx][next_elem_idx]
        min_heap.insert( (next_value, list_idx, next_elem_idx) )


// 4. Return the fully merged and sorted list
return result
```

4. Given a list of numbers [15, 30, 10, 25, 5, 20], demonstrate the step-by-step process of inserting each number into a min-priority queue and then show the order in which the elements would be extracted. You should draw the state of the priority queue (as a heap) after each insertion and deletion.Q

Final Heap: **`[5, 10, 15, 30, 25, 20]`**.

Extraction order: **`5, 10, 15, 20, 25, 30`**.

Q5.You have a stream of integers and need to find the **k-th smallest element** at any given time. Explain how you would use a priority queue to solve this problem and write the pseudo code for it. What type of priority queue (min or max) would you use, and what is the time complexity of the operation?

Ans) **Initialization:** Create a Max-Priority Queue that will be constrained to a maximum size of k.

Adding a new number (num):

- **Case 1 (Heap is not full):** If the heap has fewer than k elements (heap.size() < k), simply add num to the heap.
- **Case 2 (Heap is full):** If the heap is full (heap.size() == k), we compare num to the root of the heap (heap.peek()).
    - The root is the current k-th smallest element.
    - If num is **larger** than the root, it means num is the $(k+1)$-th smallest (or even larger) and does not belong in our set of k smallest. We discard num.
    - If num is **smaller** than the root, it means num *does* belong in the set, and the old root (the old k-th smallest) should be "kicked out." We call heap.pop() to remove the root and then heap.push(num) to add the new, smaller number.

Finding the k-th smallest: At any time, the k-th smallest element is simply the root of the max-heap (heap.peek()).

`add(num)`: The complexity is dominated by the heap `push` and `pop` operations. Since the heap's size is capped at k, these operations take **O(logk)** time.

`getKthSmallest()`: This operation is just a `peek()` on the heap, which takes **O(1)** time.

Q6. A priority queue is often implemented using a binary heap. While this is efficient, what is a potential drawback of using a standard binary heap to implement a priority queue if you frequently need to update the priority of an existing element? Describe a scenario where this limitation is significant and suggest an alternative or a modification to the standard heap structure that could improve performance for this specific operation.

Ans)The primary drawback of a standard binary heap for this operation is **finding the element you want to update.**

A binary heap (typically stored as an array) maintains the heap property (parent $\ge$ child in a max-heap), but it does not maintain a sorted order or provide a fast way to search for a *specific* element or value.

- To update an element's priority, you must first locate it within the heap array.
- Without any auxiliary data structure, the only way to guarantee you find it is to **perform a linear scan** of the array.
- This "find" operation takes **O(n) time** in the worst case.

Once you find the element (in $O(n)$ time), changing its value and performing the "heapify-up" or "heapify-down" to restore the heap property is fast (O(\log n)). However, the total operation is dominated by the search, making the update operation O(n)

Modification: Using a Hash Map (or Index Array)
You can maintain a **hash map** (or a simple array if your elements are integers from 0 to n-1) that maps an element's ID to its current index in the heap array.

- **Structure:** positionMap = { 'element_ID': heap_index }
- **updatePriority(element_ID, new_priority):**
    1. **Find ($O(1)$):** Look up element_ID in the positionMap to get its index i in O(1) time.
    2. **Update:** Change the priority of the element at heap_array[i].
    3. **Restore ($O(\log n)$):** Perform a heapify-up or heapify-down from index i.
- **Maintenance:** This is the key part. Whenever you **swap** two elements in the heap array (during push, pop, heapify-up, or heapify-down), you *must* also update their indices in the positionMap. This adds only an O(1) overhead to each swap.

By using this map, the find operation becomes O(1), and the total updatePriority operation becomes **O(\log n)**.

Q7.Suppose we have a max heap represented as an array:

[90, 85, 80, 70, 60, 75, 65].

If we insert 95 into this heap, what will be the final heap structure after performing the necessary swaps?

[95, 90, 80, 85, 60, 75, 65, 70]

Q8.A max heap is stored in an array H. If H[i] > H[j] for indices i < j, does this imply H[i] is the parent of H[j]? Why or why not?

Ans) **The Parent-Child Index Relationship is Specific:** In an array-based heap (0-indexed), the parent of `H[j]` is *always* at index `floor((j-1)/2)`. The condition `i < j` is true for many indices `i`, not just the parent's.

No Sibling Ordering: The max-heap property does *not* guarantee any relationship between siblings. A node can be larger than another node that appears later in the array without being its parent.