Heaps Questions

Basic Level:
Insert and Extract Min/Max in a Min/Max Heap: Implement a Min/Max Heap with insert and extract operations.
Heapify an Array: Convert a given array into a Min/Max Heap.
Find the Kth Largest Element: Given an unsorted array, find the Kth largest element using a heap.
Find the Kth Smallest Element: Given an unsorted array, find the Kth smallest element using a Min-Heap.
Merge K Sorted Arrays: Given K sorted arrays, merge them into one sorted array using a heap.
Check if Binary Tree is a Min/Max Heap: Given a binary tree, check if it follows the properties of a Min/Max Heap.
Intermediate Level:
Sort an Array using Heapsort: Implement Heapsort to sort an array.
K Closest Points to Origin: Given a list of points in the plane, find the K closest points to the origin using a Max-Heap.
Find the Median in a Stream of Integers: Use two heaps (Min-Heap and Max-Heap) to find the median of a stream of integers efficiently.
Top K Frequent Elements: Find the K most frequent elements in an array using a heap.
Rearrange String such that no Adjacent Characters are the Same: Rearrange the string using a Max-Heap to ensure no two adjacent characters are the same.
Minimize Cost of Connecting Ropes: Given N ropes of different lengths, find the minimum cost to connect all ropes into one rope (variation of the "Connect N Ropes" problem).
explain problem with example with input and output in simple way and give optimized and simple java code with its time complexity

Memory updated

Let's break down some of the problems one by one with examples and optimized Java code.

1. Insert and Extract Min/Max in a Min Heap

Problem Explanation:

- You need to implement a Min Heap where the smallest element is at the top.
- Operations:
  - **Insert:** Add an element to the heap.
  - **Extract Min:** Remove and return the smallest element from the heap.

Example:

- Input:

```mathematica
Insert: 5, 3, 8, 1
Extract Min: 1
Extract Min: 3
```

- Output:

```yaml
Heap after insertions: [1, 3, 8, 5]
Extract Min: 1
Heap after extraction: [3, 5, 8]
Extract Min: 3
Heap after extraction: [5, 8]
```

Java Code:

```java
import java.util.PriorityQueue;

public class MinHeap {
    PriorityQueue<Integer> minHeap;

    public MinHeap() {
        minHeap = new PriorityQueue<>();  // Min-Heap by default
    }

    public void insert(int value) {
        minHeap.add(value);
    }

    public int extractMin() {
        if (minHeap.isEmpty()) {
            throw new IllegalStateException("Heap is empty");
        }
        return minHeap.poll();  // Remove and return the min element
    }

    public static void main(String[] args) {
        MinHeap heap = new MinHeap();
        heap.insert(5);
        heap.insert(3);
        heap.insert(8);
        heap.insert(1);

        System.out.println("Extract Min: " + heap.extractMin());  // 1
        System.out.println("Extract Min: " + heap.extractMin());  // 3
    }
}
```

Time Complexity:

- **Insert:** O(log N)
- **Extract Min:** O(log N)

---

2. Find the Kth Largest Element

Problem Explanation:

- Given an unsorted array, you need to find the Kth largest element using a heap.

Example:

- Input:

```makefile
Array: [3, 2, 1, 5, 6, 4]
K: 2
```

- Output:

```mathematica
2nd Largest Element: 5
```

Java Code:

```java
import java.util.PriorityQueue;

public class KthLargestElement {
    public static int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums) {
            minHeap.add(num);
            if (minHeap.size() > k) {
                minHeap.poll();  // Remove the smallest element to keep only k largest
elements
            }
        }
        return minHeap.peek();  // Kth largest element will be at the top
    }

    public static void main(String[] args) {
        int[] nums = {3, 2, 1, 5, 6, 4};
        int k = 2;
        System.out.println(findKthLargest(nums, k));  // Output: 5
    }
}
```

Time Complexity:

- **Time:** O(N log K), where N is the number of elements in the array and K is the number of largest elements tracked.

---

3. Merge K Sorted Arrays

Problem Explanation:

- Given K sorted arrays, merge them into one sorted array using a heap.

Example:

- Input:

```csharp
Arrays:
[
  [1, 4, 5],
  [1, 3, 4],
  [2, 6]
]
```

- Output:

```javascript
Merged Array: [1, 1, 2, 3, 4, 4, 5, 6]
```

Java Code:

```java
import java.util.PriorityQueue;

class Node {
    int value, arrayIndex, elementIndex;

    public Node(int value, int arrayIndex, int elementIndex) {
        this.value = value;
        this.arrayIndex = arrayIndex;
        this.elementIndex = elementIndex;
    }
}

public class MergeKSortedArrays {
    public static int[] mergeKSortedArrays(int[][] arrays) {
        PriorityQueue<Node> minHeap = new PriorityQueue<>((a, b) -> a.value - b.value);
        int totalElements = 0;

        for (int i = 0; i < arrays.length; i++) {
            if (arrays[i].length > 0) {
                minHeap.add(new Node(arrays[i][0], i, 0));
                totalElements += arrays[i].length;
            }
        }

        int[] result = new int[totalElements];
        int index = 0;

        while (!minHeap.isEmpty()) {
            Node node = minHeap.poll();
            result[index++] = node.value;

            if (node.elementIndex + 1 < arrays[node.arrayIndex].length) {
                minHeap.add(new Node(arrays[node.arrayIndex][node.elementIndex + 1],
node.arrayIndex, node.elementIndex + 1));
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[][] arrays = {
            {1, 4, 5},
            {1, 3, 4},
            {2, 6}
        };
        int[] result = mergeKSortedArrays(arrays);

        for (int num : result) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity:

- **Time:** O(N log K), where N is the total number of elements and K is the number of arrays.

---

4. Minimize Cost of Connecting Ropes

Problem Explanation:

- Given N ropes of different lengths, the goal is to connect them into one rope with minimum cost. The cost to connect two ropes is the sum of their lengths.

Example:

- Input:

  ```makefile
  Ropes: [4, 3, 2, 6]
  ```

- Output:

  ```yaml
  Minimum Cost: 29
  ```

Java Code:

```java
import java.util.PriorityQueue;

public class ConnectRopes {
    public static int minimizeCost(int[] ropes) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int rope : ropes) {
            minHeap.add(rope);
        }

        int totalCost = 0;
        while (minHeap.size() > 1) {
```

```
                int first = minHeap.poll();
                int second = minHeap.poll();
                int cost = first + second;
                totalCost += cost;
                minHeap.add(cost);
            }

            return totalCost;
        }

        public static void main(String[] args) {
            int[] ropes = {4, 3, 2, 6};
            System.out.println("Minimum Cost: " + minimizeCost(ropes));  // Output: 29
        }
    }
```

Time Complexity:

- **Time:** O(N log N)

give remaining

Let's continue with the remaining problems.

---

5. Check if Binary Tree is a Min/Max Heap

Problem Explanation:

- You are given a binary tree. The task is to check whether it satisfies the properties of a Min-Heap or Max-Heap.
  - **Min-Heap:** Every node should be smaller than its children.
  - **Max-Heap:** Every node should be larger than its children.

Example:

- Input:

```markdown
Binary Tree:
    10
   /  \
  20   30
```

- Output:

```csharp
The tree is not a Min-Heap.
```

Java Code:

```java
class TreeNode {
    int value;
    TreeNode left, right;

    TreeNode(int value) {
        this.value = value;
        left = right = null;
    }
}

public class BinaryTreeHeapChecker {
    public static boolean isMinHeap(TreeNode root) {
        if (root == null) return true;

        if ((root.left != null && root.value > root.left.value) ||
            (root.right != null && root.value > root.right.value)) {
            return false;
        }

        return isMinHeap(root.left) && isMinHeap(root.right);
    }

    public static boolean isMaxHeap(TreeNode root) {
        if (root == null) return true;

        if ((root.left != null && root.value < root.left.value) ||
            (root.right != null && root.value < root.right.value)) {
            return false;
        }

        return isMaxHeap(root.left) && isMaxHeap(root.right);
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(20);
        root.right = new TreeNode(30);

        System.out.println("Is Min-Heap: " + isMinHeap(root));  // Output: false
        System.out.println("Is Max-Heap: " + isMaxHeap(root));  // Output: true
    }
}
```

Time Complexity:

- **Time:** O(N), where N is the number of nodes in the tree (since we visit every node once).

---

6. Sort an Array using Heapsort

Problem Explanation:

- Implement Heapsort to sort an array. This sorting algorithm uses a heap to sort elements in O(N log N) time.

Example:

- Input:

```javascript
Array: [3, 5, 1, 10, 2]
```

- Output:

```javascript
Sorted Array: [1, 2, 3, 5, 10]
```

Java Code:

```java
public class HeapSort {
    public static void heapsort(int[] arr) {
        int n = arr.length;

        // Build max heap
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }

        // Extract elements from heap one by one
        for (int i = n - 1; i >= 0; i--) {
            // Move current root to the end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // Heapify the reduced heap
            heapify(arr, i, 0);
        }
    }

    private static void heapify(int[] arr, int n, int i) {
        int largest = i;  // Initialize largest as root
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        // If left child is larger than root
        if (left < n && arr[left] > arr[largest]) {
            largest = left;
        }

        // If right child is larger than largest so far
        if (right < n && arr[right] > arr[largest]) {
            largest = right;
        }

        // If largest is not root
        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
        }
    }

    public static void main(String[] args) {
        int[] arr = {3, 5, 1, 10, 2};
        heapsort(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
        // Output: 1 2 3 5 10
    }
}
```

Time Complexity:

- **Time:** O(N log N)

---

7. K Closest Points to the Origin

Problem Explanation:

- You are given a list of points on a 2D plane. Find the K closest points to the origin (0, 0) using a Max-Heap.

Example:

- Input:

```lua
Points: [[1, 3], [-2, 2], [5, 8], [0, 1]]
K: 2
```

- Output:

```lua
Closest Points: [[-2, 2], [0, 1]]
```

Java Code:

```java
import java.util.PriorityQueue;

public class KClosestPoints {
    public static int[][] kClosest(int[][] points, int K) {
        PriorityQueue<int[]> maxHeap = new PriorityQueue<>((a, b) ->
            (b[0] * b[0] + b[1] * b[1]) - (a[0] * a[0] + a[1] * a[1])
        );

        for (int[] point : points) {
            maxHeap.add(point);
            if (maxHeap.size() > K) {
                maxHeap.poll();
            }
        }

        int[][] result = new int[K][2];
        for (int i = 0; i < K; i++) {
            result[i] = maxHeap.poll();
        }
        return result;
    }

    public static void main(String[] args) {
        int[][] points = {{1, 3}, {-2, 2}, {5, 8}, {0, 1}};
        int K = 2;
        int[][] result = kClosest(points, K);
        for (int[] point : result) {
            System.out.println("[" + point[0] + ", " + point[1] + "]");
        }
        // Output: [-2, 2], [0, 1]
    }
}
```

Time Complexity:

- **Time:** O(N log K), where N is the number of points.

---

8. Find the Median in a Stream of Integers

Problem Explanation:

- Given a stream of integers, use two heaps (Min-Heap and Max-Heap) to efficiently find the median at any point.

Example:

- Input:

```arduino
Stream: [5, 10, 15]
```

- Output:

```makefile
Median: 10
```

Java Code:

```java
import java.util.Collections;
import java.util.PriorityQueue;

public class MedianFinder {
    private PriorityQueue<Integer> maxHeap;
    private PriorityQueue<Integer> minHeap;

    public MedianFinder() {
        maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        minHeap = new PriorityQueue<>();
    }

    public void addNum(int num) {
        maxHeap.add(num);
```

```
            minHeap.add(maxHeap.poll());

            if (maxHeap.size() < minHeap.size()) {
                maxHeap.add(minHeap.poll());
            }
        }

        public double findMedian() {
            if (maxHeap.size() > minHeap.size()) {
                return maxHeap.peek();
            } else {
                return (maxHeap.peek() + minHeap.peek()) / 2.0;
            }
        }

        public static void main(String[] args) {
            MedianFinder medianFinder = new MedianFinder();
            medianFinder.addNum(5);
            medianFinder.addNum(10);
            System.out.println(medianFinder.findMedian());  // Output: 7.5
            medianFinder.addNum(15);
            System.out.println(medianFinder.findMedian());  // Output: 10
        }
    }
```

Time Complexity:

- **Add Number:** O(log N)
- **Find Median:** O(1)

---

9. Top K Frequent Elements

Problem Explanation:

- Given an array, find the K most frequent elements using a heap.

Example:

- Input:

```makefile
Array: [1, 1, 1, 2, 2, 3]
K: 2
```

- Output:

```css
Top 2 Frequent Elements: [1, 2]
```

Java Code:

```java
import java.util.*;

public class TopKFrequentElements {
    public static int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> countMap = new HashMap<>();
        for (int num : nums) {
            countMap.put(num, countMap.getOrDefault(num, 0) + 1);
        }

        PriorityQueue<Integer> minHeap = new PriorityQueue<>((a, b) -> countMap.get(a) -
countMap.get(b));

        for (int num : countMap.keySet()) {
            minHeap.add(num);
            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }

        int[] result = new int[k];
        for (int i = 0; i < k; i++) {
            result[i] = minHeap.poll();
        }
        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        int[] result = topKFrequent(nums, k);
        System.out.println(Arrays.toString(result));  // Output: [1, 2]
    }
}
```

Time Complexity:

- **Time:** O(N log K) where N is the number of elements in the array.

These are the remaining problems with their explanations and Java solutions. Let me know if you need further clarification!

give remaining

Let's continue with the remaining problems.

---

10. Rearrange String Such That No Adjacent Characters Are the Same

Problem Explanation:

- Given a string, rearrange it such that no two adjacent characters are the same. Use a Max-Heap to solve this problem.

Example:

- Input:

```arduino
String: "aaabbc"
```

- Output:

```arduino
Rearranged String: "ababac"
```

Explanation: We have a string `"aaabbc"`. The goal is to rearrange the characters such that no two adjacent characters are the same. Using a Max-Heap, we can always pick the character with the highest frequency, place it, and then pick the next character with the second-highest frequency to avoid adjacent duplicates.

Java Code:

```java
import java.util.HashMap;
import java.util.Map;
import java.util.PriorityQueue;

public class RearrangeString {
    public static String rearrangeString(String s) {
        Map<Character, Integer> freqMap = new HashMap<>();
        for (char c : s.toCharArray()) {
            freqMap.put(c, freqMap.getOrDefault(c, 0) + 1);
        }

        PriorityQueue<Map.Entry<Character, Integer>> maxHeap = new PriorityQueue<>(
            (a, b) -> b.getValue() - a.getValue());

        maxHeap.addAll(freqMap.entrySet());

        StringBuilder result = new StringBuilder();
        Map.Entry<Character, Integer> prev = null;

        while (!maxHeap.isEmpty()) {
            Map.Entry<Character, Integer> current = maxHeap.poll();
            result.append(current.getKey());
            current.setValue(current.getValue() - 1);

            if (prev != null && prev.getValue() > 0) {
                maxHeap.add(prev);
            }

            prev = current;
        }

        return result.length() == s.length() ? result.toString() : "";
    }

    public static void main(String[] args) {
        String input = "aaabbc";
        System.out.println(rearrangeString(input));  // Output: "ababac"
    }
}
```

Time Complexity:

- **Time:** O(N log N), where N is the length of the string.

---

11. Minimize Cost of Connecting Ropes

Problem Explanation:

- You are given `N` ropes of different lengths. The task is to connect all the ropes into one rope. The cost of connecting two ropes is equal to the sum of their lengths. You need to minimize the total cost to connect all ropes.

Example:

- Input:

```makefile
Ropes: [4, 3, 2, 6]
```

- Output:

```yaml
Minimum Cost: 29
```

Explanation: We can connect the ropes in the following way:

- First, connect ropes of lengths 2 and 3. The cost is 5.
- Then, connect the resulting rope (of length 5) with the rope of length 4. The cost is 9.
- Finally, connect the resulting rope (of length 9) with the rope of length 6. The cost is 15. The total minimum cost is `5 + 9 + 15 = 29`.

Java Code:

```java
import java.util.PriorityQueue;

public class MinCostOfRopes {
    public static int minCost(int[] ropes) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int rope : ropes) {
            minHeap.add(rope);
        }

        int totalCost = 0;
        while (minHeap.size() > 1) {
            int first = minHeap.poll();
            int second = minHeap.poll();
            int cost = first + second;
            totalCost += cost;
            minHeap.add(cost);
        }
```

```
            return totalCost;
        }

        public static void main(String[] args) {
            int[] ropes = {4, 3, 2, 6};
            System.out.println("Minimum Cost: " + minCost(ropes));  // Output: 29
        }
    }
```

Time Complexity:

- **Time:** O(N log N), where N is the number of ropes.

These are the final two problems with explanations and their corresponding Java solutions. Let me know if you need further clarifications!

Advanced Level:
Find the Kth Largest Element in a Stream: Implement a data structure to always return the Kth largest element from a stream of integers.
Merge K Sorted Linked Lists: Given K sorted linked lists, merge them into a single sorted list using a Min-Heap.
Build a Heap from a Linked List: Convert a given linked list into a heap.
Running Median with Two Heaps: Find the median in a stream of integers using two heaps (min-heap and max-heap) with efficient time complexity.
Maximum Product of Two Elements in an Array: Use a Max-Heap to find the maximum product of any two distinct elements in the array.
Sliding Window Maximum: Find the maximum element in every sliding window of size K using a heap.
Smallest Range Covering Elements from K Lists: Given K sorted lists, find the smallest range that includes at least one number from each of the K lists.
Frequency Sort: Sort the elements of an array based on their frequency using a heap.

Memory updated
Let's go through each advanced-level problem with explanations and optimized Java code.

---

1. Find the Kth Largest Element in a Stream

Problem Explanation:

- Implement a data structure that continuously processes integers from a stream and always returns the Kth largest element. A Min-Heap can be used to maintain the K largest elements.

Example:

- Input:

```makefile
Stream: [4, 5, 8, 2]
K = 3
```

- Output:

```mathematica
Kth Largest Element: 4
```

Java Code:

```java
import java.util.PriorityQueue;

public class KthLargestInStream {
    private PriorityQueue<Integer> minHeap;
    private int k;

    public KthLargestInStream(int k, int[] nums) {
        this.k = k;
        minHeap = new PriorityQueue<>();
        for (int num : nums) {
            add(num);
        }
    }

    public int add(int num) {
        minHeap.offer(num);
        if (minHeap.size() > k) {
            minHeap.poll();
        }
        return minHeap.peek();
    }

    public static void main(String[] args) {
        int[] nums = {4, 5, 8, 2};
        KthLargestInStream kthLargest = new KthLargestInStream(3, nums);
        System.out.println(kthLargest.add(3));  // Output: 4
        System.out.println(kthLargest.add(5));  // Output: 5
    }
}
```

Time Complexity:

- **Time:** O(log K) for each insertion.

---

2. Merge K Sorted Linked Lists

Problem Explanation:

- You are given K sorted linked lists. The task is to merge them into one sorted linked list using a Min-Heap.

Example:

- Input:

```lua
K sorted lists: [[1, 4, 5], [1, 3, 4], [2, 6]]
```

- Output:

```less
Merged list: [1, 1, 2, 3, 4, 4, 5, 6]
```

Java Code:

```java
import java.util.PriorityQueue;

class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}

public class MergeKSortedLists {
    public static ListNode mergeKLists(ListNode[] lists) {
        PriorityQueue<ListNode> minHeap = new PriorityQueue<>((a, b) -> a.val - b.val);

        for (ListNode list : lists) {
            if (list != null) {
                minHeap.add(list);
            }
        }

        ListNode dummy = new ListNode(0);
        ListNode current = dummy;

        while (!minHeap.isEmpty()) {
            ListNode minNode = minHeap.poll();
            current.next = minNode;
            current = current.next;

            if (minNode.next != null) {
                minHeap.add(minNode.next);
            }
        }

        return dummy.next;
    }

    public static void main(String[] args) {
        ListNode[] lists = new ListNode[3];
        lists[0] = new ListNode(1);
        lists[0].next = new ListNode(4);
        lists[0].next.next = new ListNode(5);

        lists[1] = new ListNode(1);
        lists[1].next = new ListNode(3);
        lists[1].next.next = new ListNode(4);

        lists[2] = new ListNode(2);
        lists[2].next = new ListNode(6);

        ListNode result = mergeKLists(lists);
        while (result != null) {
            System.out.print(result.val + " ");
            result = result.next;
        }
        // Output: 1 1 2 3 4 4 5 6
    }
}
```

Time Complexity:

- **Time:** O(N log K), where N is the total number of nodes.

---

3. Build a Heap from a Linked List

Problem Explanation:

- Convert a linked list into a Min-Heap or Max-Heap.

Approach:

1. First, traverse the linked list and store elements in an array.
2. Heapify the array to convert it into a Min-Heap or Max-Heap.

Example:

- Input:

```rust
Linked List: 4 -> 1 -> 3 -> 9 -> 7
```

- Output:

```makefile
Heap: [1, 4, 3, 9, 7] (Min-Heap)
```

Java Code:

```java
import java.util.ArrayList;
import java.util.PriorityQueue;

class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}

public class LinkedListToHeap {
    public static ArrayList<Integer> buildMinHeapFromList(ListNode head) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        while (head != null) {
            minHeap.add(head.val);
            head = head.next;
        }

        ArrayList<Integer> result = new ArrayList<>(minHeap);
        return result;
    }

    public static void main(String[] args) {
        ListNode head = new ListNode(4);
        head.next = new ListNode(1);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(9);
        head.next.next.next.next = new ListNode(7);

        ArrayList<Integer> heap = buildMinHeapFromList(head);
        System.out.println(heap);  // Output: [1, 4, 3, 9, 7] (Min-Heap)
    }
}
```

Time Complexity:

- **Time:** O(N log N)

---

4. Running Median with Two Heaps

Problem Explanation:

- Find the running median in a stream of integers using two heaps (a Max-Heap for the smaller half and a Min-Heap for the larger half).

Example:

- Input:

  arduino

  ```
  Stream: [5, 10, 15]
  ```

- Output:

  makefile

  ```
  Median: 10
  ```

Java Code:

java

```
import java.util.Collections;
import java.util.PriorityQueue;

public class RunningMedian {
    private PriorityQueue<Integer> maxHeap;  // smaller half
    private PriorityQueue<Integer> minHeap;  // larger half

    public RunningMedian() {
```

```
        maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        minHeap = new PriorityQueue<>();
    }

    public void addNum(int num) {
        maxHeap.add(num);
        minHeap.add(maxHeap.poll());

        if (maxHeap.size() < minHeap.size()) {
            maxHeap.add(minHeap.poll());
        }
    }

    public double findMedian() {
        if (maxHeap.size() > minHeap.size()) {
            return maxHeap.peek();
        } else {
            return (maxHeap.peek() + minHeap.peek()) / 2.0;
        }
    }

    public static void main(String[] args) {
        RunningMedian rm = new RunningMedian();
        rm.addNum(5);
        rm.addNum(10);
        System.out.println(rm.findMedian());  // Output: 7.5
        rm.addNum(15);
        System.out.println(rm.findMedian());  // Output: 10
    }
}
```

Time Complexity:

- **Time:** O(log N) for insertion, O(1) for finding the median.

---

5. Maximum Product of Two Elements in an Array

Problem Explanation:

- Find the maximum product of any two distinct elements in an array using a Max-Heap.

Example:

- Input:

  ```javascript
  Array: [3, 4, 5, 2]
  ```

- Output:

  ```mathematica
  Maximum Product: 12
  ```

Java Code:

```java
import java.util.PriorityQueue;

public class MaxProductOfTwoElements {
    public static int maxProduct(int[] nums) {
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

        for (int num : nums) {
            maxHeap.add(num);
        }

        int first = maxHeap.poll() - 1;
        int second = maxHeap.poll() - 1;

        return first * second;
    }

    public static void main(String[] args) {
        int[] nums = {3, 4, 5, 2};
        System.out.println(maxProduct(nums));  // Output: 12
    }
}
```

Time Complexity:

- **Time:** O(N log N)

I’ll continue with the remaining problems in the next message.

give remaining

Continuing with the remaining advanced-level heap problems:

---

6. Sliding Window Maximum

Problem Explanation:

- Given an array and a sliding window of size `K`, find the maximum element in each window as it slides from left to right.

Example:

- Input:

  ```makefile
  Array: [1, 3, -1, -3, 5, 3, 6, 7]
  K = 3
  ```

- Output:

```makefile
Maximums: [3, 3, 5, 5, 6, 7]
```

Java Code:

```java
import java.util.ArrayList;
import java.util.Deque;
import java.util.LinkedList;
import java.util.List;

public class SlidingWindowMaximum {
    public static List<Integer> maxSlidingWindow(int[] nums, int k) {
        List<Integer> result = new ArrayList<>();
        Deque<Integer> deque = new LinkedList<>();

        for (int i = 0; i < nums.length; i++) {
            // Remove elements outside the window
            if (!deque.isEmpty() && deque.peekFirst() == i - k) {
                deque.pollFirst();
            }

            // Remove elements smaller than the current element
            while (!deque.isEmpty() && nums[deque.peekLast()] <= nums[i]) {
                deque.pollLast();
            }

            deque.offerLast(i);

            // The first k elements will not have a complete window yet
            if (i >= k - 1) {
                result.add(nums[deque.peekFirst()]);
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, -1, -3, 5, 3, 6, 7};
        int k = 3;
        System.out.println(maxSlidingWindow(nums, k));  // Output: [3, 3, 5, 5, 6, 7]
    }
}
```

Time Complexity:

- **Time:** O(N), where N is the number of elements in the array. Each element is added and removed from the deque at most once.

---

7. Smallest Range Covering Elements from K Lists

Problem Explanation:

- Given K sorted lists, find the smallest range that includes at least one number from each of the K lists.

Example:

- Input:

```lua
Lists: [[4, 10, 15], [1, 9, 14], [2, 8, 16]]
```

- Output:

```less
Smallest Range: [8, 14]
```

Java Code:

```java
import java.util.PriorityQueue;

class Element {
    int value;
    int listIndex;
    int elementIndex;

    Element(int value, int listIndex, int elementIndex) {
        this.value = value;
        this.listIndex = listIndex;
        this.elementIndex = elementIndex;
    }
}

public class SmallestRange {
    public static int[] findSmallestRange(int[][] nums) {
        PriorityQueue<Element> minHeap = new PriorityQueue<>((a, b) ->
Integer.compare(a.value, b.value));
        int max = Integer.MIN_VALUE;
        int rangeStart = 0, rangeEnd = Integer.MAX_VALUE;

        // Add the first element from each list to the heap
        for (int i = 0; i < nums.length; i++) {
            minHeap.add(new Element(nums[i][0], i, 0));
            max = Math.max(max, nums[i][0]);
        }

        while (true) {
            Element minElement = minHeap.poll();
            int minValue = minElement.value;
            int listIndex = minElement.listIndex;
            int elementIndex = minElement.elementIndex;

            if (rangeEnd - rangeStart > max - minValue) {
                rangeStart = minValue;
                rangeEnd = max;
            }

            if (elementIndex + 1 == nums[listIndex].length) {
                break;
            }

            Element nextElement = new Element(nums[listIndex][elementIndex + 1],
listIndex, elementIndex + 1);
            minHeap.add(nextElement);
            max = Math.max(max, nextElement.value);
        }
        return new int[]{rangeStart, rangeEnd};
    }

    public static void main(String[] args) {
        int[][] nums = {
            {4, 10, 15},
            {1, 9, 14},
            {2, 8, 16}
        };
        int[] range = findSmallestRange(nums);
        System.out.println("Smallest Range: [" + range[0] + ", " + range[1] + "]");  //
Output: [8, 14]
    }
}
```

Time Complexity:

- **Time:** O(N log K), where N is the total number of elements across all lists, and K is the number of lists.

---

8. Frequency Sort

Problem Explanation:

- Sort elements of an array based on their frequency. Elements with higher frequencies should appear first.

Example:

- Input:

  ```javascript
  Array: [1, 1, 2, 2, 2, 3, 3]
  ```

- Output:

  ```javascript
  Sorted Array: [2, 2, 2, 1, 1, 3, 3]
  ```

Java Code:

```java
import java.util.HashMap;
import java.util.Map;
import java.util.PriorityQueue;

public class FrequencySort {
    public static int[] frequencySort(int[] nums) {
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }
```

```
        PriorityQueue<Map.Entry<Integer, Integer>> maxHeap = new PriorityQueue<>(
            (a, b) -> Integer.compare(b.getValue(), a.getValue()));

        maxHeap.addAll(frequencyMap.entrySet());

        int[] result = new int[nums.length];
        int index = 0;
        while (!maxHeap.isEmpty()) {
            Map.Entry<Integer, Integer> entry = maxHeap.poll();
            int num = entry.getKey();
            int freq = entry.getValue();
            for (int i = 0; i < freq; i++) {
                result[index++] = num;
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 1, 2, 2, 2, 3, 3};
        int[] sorted = frequencySort(nums);
        for (int num : sorted) {
            System.out.print(num + " ");
        }
        // Output: 2 2 2 1 1 3 3
    }
}
```

Time Complexity:

- **Time:** O(N log N), where N is the number of elements in the array.

These are the remaining advanced-level heap problems with explanations and their Java solutions. If you have any questions or need further details, feel free to ask!

ChatGPT can make mistakes. Check important info.