PW SKILLS

Lesson Plan

Priority Queue-1

Java

PW SKILLS

Today's Checklist:

- Introduction
- Types of Priority Queue
- Operation on Priority Queue
- Perfect Binary Tree
- Implementation of Priority Queue
- Time Complexity based on given implementation
- Practice Questions

Introduction

A priority queue is a data structure that manages elements with specific priorities. Unlike a regular queue, where the first element added is the first to be removed (FIFO), a priority queue ensures that the element with the highest priority is processed first.

Differences from a Regular Queue:

- In a regular queue, elements are processed in the order they are added, while in a priority queue, elements are processed based on their priority level.
- Priority queues do not follow a strict FIFO (First In, First Out) approach like regular queues.

Purpose and Importance:

- Priority queues find extensive use in various domains, including:
- Scheduling Algorithms: Managing tasks or jobs based on their priority levels.
- Graph Algorithms: Implementing algorithms like Dijkstra's shortest path algorithm where nodes with the shortest distance are processed first.
- Operating Systems: Handling processes based on their importance and resource requirements

Types of Priority Queue

Title: Exploring Priority Queue Variants

- **Min Heap:**
  - The smallest element has the highest priority.
  - Ideal for applications where the smallest value needs to be processed first.

```
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
```

- **Max Heap:**
  - The largest element has the highest priority.
  - Useful when the largest value is of utmost significance.

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

- **Priority Queue with Comparator:**
  - Allows custom priority rules, providing flexibility in defining how elements are prioritized.
  - Example: Sorting based on different criteria (e.g., custom objects with specific comparison rules).

```
PriorityQueue<Student> customPriorityQueue = new
PriorityQueue<>(Comparator.comparing(Student::getGPA));
```

Operations on Priority Queue

Title: Key Operations in Priority Queue

- **Insertion (offer/enqueue):**
  - Adding an element to the queue while maintaining the priority order.
  - Ensures the element is placed at the correct position according to its priority.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(5);
```

- **Extraction (poll/dequeue):**
  - Removing and returning the highest priority element from the queue.
  - The element with the highest priority is dequeued and removed.

```
int highestPriority = pq.poll();
```

- **Peek:**
  - Viewing the highest priority element without removing it from the queue.
  - Helps in observing the next element to be dequeued without altering the queue.

```
int peekValue = pq.peek();
```

Slide 4: Perfect Binary Tree

Title: Role of Perfect Binary Trees in Priority Queues

- **Definition:**
  - Perfect binary trees have every level completely filled, except possibly the last level, which is filled from left to right.
- **Utilization in Priority Queues:**
  - Perfect binary trees are commonly used to represent heaps, the underlying structure of priority queues.
  - Illustration depicting the structure of a perfect binary tree within a priority queue context.

Slide 5: Implementation of Priority Queue

Title: Implementing Priority Queues

- **Using Heaps:**
  - Priority queues can be efficiently implemented using heaps, often through array-based representations
  - Demonstration of how elements are stored and organized in the perfect binary tree structure.
  - Example of a basic priority queue implementation using heaps in .

Code:

```
  import java.util.ArrayList;
import java.util.List;
import java.util.Collections;

class PriorityQueue<T extends Comparable<T>> {
    private List<T> heap;

    public PriorityQueue() {
        heap = new ArrayList<>();
    }

    public void offer(T element) {
        heap.add(element);
        int index = heap.size() - 1;
        bubbleUp(index);
    }

    public T poll() {
        if (heap.isEmpty()) {
            return null;
        }

        Collections.swap(heap, 0, heap.size() - 1);
        T removed = heap.remove(heap.size() - 1);
        bubbleDown(0);
        return removed;
    }

    public T peek() {
        return heap.isEmpty() ? null : heap.get(0);
    }
    private void bubbleUp(int index) {
        int parent = (index - 1) / 2;
        while (index > 0 && heap.get(index).compareTo(heap.get(parent)) < 0) {
            Collections.swap(heap, index, parent);
            index = parent;
            parent = (index - 1) / 2;
        }
    }
    private void bubbleDown(int index) {
        int leftChild, rightChild, minIndex;
        while (true) {
            leftChild = 2 * index + 1;
            rightChild = 2 * index + 2;
            minIndex = index;
```

```
            if (leftChild < heap.size() &&
heap.get(leftChild).compareTo(heap.get(minIndex)) < 0) {
                minIndex = leftChild;
            }

            if (rightChild < heap.size() &&
heap.get(rightChild).compareTo(heap.get(minIndex)) < 0) {
                minIndex = rightChild;
            }

            if (minIndex != index) {
                Collections.swap(heap, index, minIndex);
                index = minIndex;
            } else {
                break;
            }
        }
    }
}
```

Time Complexity based on implementation:

Implementation	push	pop	peek
Array	O(1)	O(n)	O(n)
Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

STL of Priority Queue

The Java Collections Framework provides the `PriorityQueue` class, offering an implementation of the priority queue data structure. This class uses a priority heap to maintain the elements in a sorted order based on their natural ordering or a specified comparator. Elements are inserted according to their priority and retrieved accordingly.

Example:

```
import java.util.*;

public class PriorityQueueExample {
    public static void main(String[] args) {
        // Creating a priority queue
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        // Adding elements to the priority queue
        pq.offer(5);
        pq.offer(1);
        pq.offer(3);

        // Polling elements in priority order
        while (!pq.isEmpty()) {
            System.out.print(pq.poll() + " ");
        }
    }
}
```

STL Based on Given Priority

In Java's `PriorityQueue`, elements can have a natural ordering (if they implement the `Comparable` interface) or can follow a custom priority defined by a comparator. This allows flexibility in sorting elements based on different criteria, making it versatile for various use cases.

Example:

```
import java.util.*;

public class CustomPriorityQueue {
    public static void main(String[] args) {
        // Creating a priority queue with a custom comparator
        PriorityQueue<Integer> customPQ = new PriorityQueue<>((a, b)
→ Integer.compare(b, a));

        // Adding elements to the priority queue with custom
priority
        customPQ.offer(5);
        customPQ.offer(1);
        customPQ.offer(3);

        // Polling elements with custom priority order
        while (!customPQ.isEmpty()) {
            System.out.print(customPQ.poll() + " ");
        }
    }
}
```

Ques : What is the functionality of the following piece of code?

```
public Object delete_key()
{
 if(count == 0)
 {
  System.out.println("Q is empty");
  System.exit(0);
 }
 else
 {
  Node cur = head.getNext();
  Node dup = cur.getNext();
  Object e = cur.getEle();
  head.setNext(dup);
  count--;
  return e;
 }
}
```

a) Delete the second element in the list
b) Return but not delete the second element in the list
c) Delete the first element in the list
d) Return but not delete the first element in the list

Ans: c) Delete the first element in the list
It deletes the first element from the queue and returns the element, reducing the count of elements in the queue by 1.

Ques : kth largest element in a stream [Leetcode 703]
Design a class to find the kth largest element in a stream. Note that it is the kth largest element in the sorted order, not the kth distinct element.

Implement Kth Largest class:

KthLargest(int k, int[] nums) Initializes the object with the integer k and the stream of integers nums.
int add(int val) Appends the integer val to the stream and returns the element representing the kth largest element in the stream.

Example 1:
Input
["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]
Output
[null, 4, 5, 5, 8, 8]

Explanation
KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
kthLargest.add(3); // return 4
kthLargest.add(5); // return 5
kthLargest.add(10); // return 5
kthLargest.add(9); // return 8
kthLargest.add(4); // return 8

Code:

```
import java.util.PriorityQueue;

class KthLargest {
    private PriorityQueue<Integer> p;
    private int k;

    public KthLargest(int K, int[] nums) {
        this.k = K;
        p = new PriorityQueue<>();

        for (int num : nums) {
            p.offer(-num);
            if (p.size() > k) {
                p.poll();
            }
        }
    }

    public int add(int val) {
        p.offer(-val);
        if (p.size() > k) {
            p.poll();
        }
        return -p.peek();
    }
}
```

Explanation: We will be using priority queue and will add the K elements in our queue
Since priority queue in C++ follows the max-heap implementation soo we will add negative of every element to make it follow the min-heap implementation.

Time Complexity: O(N·log(N)+M·log(k)), For the constructor, adding N elements initially takes O(N * log(N)) time due to heap operations, and for each subsequent addition, maintaining the k elements takes O(M * log(k)), where M is the number of new elements added.

Space Complexity: O(N),to store the initial N elements in the priority queue.

Ques : Last stone weight [Leetcode 1046]
You are given an array of integers stones where stones[i] is the weight of the ith stone.

We are playing a game with the stones. On each turn, we choose the heaviest two stones and smash them together. Suppose the heaviest two stones have weights x and y with x <= y. The result of this smash is:

If x == y, both stones are destroyed, and
If x != y, the stone of weight x is destroyed, and the stone of weight y has new weight y - x.
At the end of the game, there is at most one stone left.

Return the weight of the last remaining stone. If there are no stones left, return 0.

Example 1:
Input: stones = [2,7,4,1,8,1]
Output: 1
Explanation:
We combine 7 and 8 to get 1 so the array converts to [2,4,1,1,1] then,
we combine 2 and 4 to get 2 so the array converts to [2,1,1,1] then,
we combine 2 and 1 to get 1 so the array converts to [1,1,1] then,
we combine 1 and 1 to get 0 so the array converts to [1] then that's the value of the last stone.

Example 2:
Input: stones = [1]
Output: 1

Code:

```
import java.util.PriorityQueue;

class Solution {
    public int lastStoneWeight(int[] a) {
        PriorityQueue<Integer> pq = new PriorityQueue<>((x, y) -> Integer.compare(y, x));

        for (int num : a) {
            pq.offer(num);
        }

        while (pq.size() > 1) {
            int aVal = pq.poll();
            int bVal = pq.poll();

            if (aVal != bVal) {
                pq.offer(Math.abs(aVal - bVal));
            }
        }

        return pq.isEmpty() ? 0 : pq.peek();
    }
}
```

Explanation: To solve this problem, we can use a priority queue to keep track of the heaviest stones.
At each turn, we can pop the two heaviest stones from the heap, smash them together according to the given rules, and then push the resulting stone (if any) back onto the heap.
We repeat this process until there is at most one stone left in the heap.

Time Complexity: O(N * log N) for initialization and log N for each iteration in the while loop.

Space Complexity: O(N) for storing the elements in the priority queue.

Ques : K closest points to origin [Leetcode 973]
Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).

The distance between two points on the X-Y plane is the Euclidean distance (i.e., √(x1 - x2)2 + (y1 - y2)2).
You may return the answer in any order. The answer is guaranteed to be unique (except for the order that it is in).

Example 1:
Input: points = [[1,3],[-2,2]], k = 1
Output: [[-2,2]]
Explanation:
The distance between (1, 3) and the origin is sqrt(10).
The distance between (-2, 2) and the origin is sqrt(8).
Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.
We only want the closest k = 1 points from the origin, so the answer is just [[-2,2]].

Example 2:
Input: points = [[3,3],[5,-1],[-2,4]], k = 2
Output: [[3,3],[-2,4]]
Explanation: The answer [[-2,4],[3,3]] would also be accepted.

Code:

```
class Solution {
    public int[][] kClosest(int[][] points, int k) {
        PriorityQueue<Pair<Double, int[]>> pq = new PriorityQueue<>(
            Comparator.comparingDouble(Pair::getKey)
        );
        int[][] ans = new int[k][2];

        for (int[] point : points) {
            double distance = Math.pow(point[0], 2) + Math.pow(point[1], 2);
            pq.offer(new Pair<>(distance, point));
            if (pq.size() > k) {
                pq.poll();
            }
        }

        int index = 0;
        while (!pq.isEmpty()) {
            ans[index++] = pq.poll().getValue();
        }

        return ans;
    }

    static class Pair<K, V> {
        private final K key;
        private final V value;
```

```
        public Pair(K key, V value) {
            this.key = key;
            this.value = value;
        }

        public K getKey() {
            return key;
        }

        public V getValue() {
            return value;
        }
    }
}
```

Explanation: To get the k smallest distance, we use a priority_queue to store all the distances
Whenever we push a new distance into it, we check if the size of it is bigger than k or not
If yes, we pop() the top element which is the biggest
After pushing all elements and follow the above rule, we can get the k smallest distance
Notice that we just need x1^2+x2^2 for comparing distance, don't really have to sqrt() it

Time Complexity: O(N * log K) for iterating through the points and pushing them into the priority queue, where N is the number of points.

Space Complexity: O(K) for the priority queue to store the K closest points.

Ques: Find K pairs with smallest sum [Leetcode 373]
You are given two integer arrays nums1 and nums2 sorted in non-decreasing order and an integer k.

Define a pair (u, v) which consists of one element from the first array and one element from the second array.

Return the k pairs (u1, v1), (u2, v2), ..., (uk, vk) with the smallest sums.

Example 1:
Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3
Output: [[1,2],[1,4],[1,6]]
Explanation: The first 3 pairs are returned from the sequence: [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]

Example 2:
Input: nums1 = [1,1,2], nums2 = [1,2,3], k = 2
Output: [[1,1],[1,1]]
Explanation: The first 2 pairs are returned from the sequence: [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]

Code:

```
class Solution {
    public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        List<List<Integer>> ans = new ArrayList<>();
        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));

        for (int x : nums1) {
            for (int y : nums2) {
                if (pq.size() < k) {
                    pq.offer(new int[]{x + y, x, y});
                } else {
                    if (x + y < pq.peek()[0]) {
                        pq.poll();
                        pq.offer(new int[]{x + y, x, y});
                    }
                }
            }
        }
```

```
        while (!pq.isEmpty() && k-- > 0) {
            int[] pair = pq.poll();
            ans.add(Arrays.asList(pair[1], pair[2]));
        }

        return ans;
    }
}
```

Explanation: Make a max priority queue of vectors {sum,x,y} where sum = x+y
Traverse over two arrays in nested loops
Insert in priority queue till size is less than k
Now size of priority queue is k, we will maintain this size and will only include that sum which is less than top.
After complete traversal, priority queue will be having just k largest pairs but in order of decreasing sum. Top will be the max sum present.
Now insert them in the reverse order in vector and return.

Time Complexity: O(M * N * log K) where M and N are the sizes of nums1 and nums2 respectively, for iterating through both arrays and maintaining the priority queue of size K.

Space Complexity: O(K) for the priority queue to store the K smallest pairs.

Ques: Reorganize string [Leetcode 767]
Given a string s, rearrange the characters of s so that any two adjacent characters are not the same.

Return any possible rearrangement of s or return "" if not possible.

Example 1:
Input: s = "aab"
Output: "aba"

Example 2:
Input: s = "aaab"
Output: ""

Code:

```
class Solution {
    public String reorganizeString(String S) {
        StringBuilder res = new StringBuilder();
        int[] freq = new int[26];

        for (char c : S.toCharArray()) {
            ++freq[c - 'a'];
        }

        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> b[1] - a[1]);

        for (int i = 0; i < 26; ++i) {
            if (freq[i] > 0) {
                pq.offer(new int[]{i, freq[i]});
            }
        }

        while (!pq.isEmpty()) {
            int[] current = pq.poll();
```

```
            if (res.length() == 0 || res.charAt(res.length() - 1) != (char)('a' +
current[0])) {
                res.append((char)('a' + current[0]));
                if (--current[1] > 0) {
                    pq.offer(current);
                }
            } else {
                int[] next = pq.poll();
                if (next == null) {
                    return "";
                }
                res.append((char)('a' + next[0]));
                if (--next[1] > 0) {
                    pq.offer(next);
                }
                pq.offer(current);
            }
        }

        return res.toString();
    }
}
```

Explanation: Everytime we need to add the character with most frequency. This is the optimal solution. So we maintain the frequencies for each character, and find the character with most frequency at each time by iterating the array. Notice the character should not be the last character added into the string. Total time complexity is O(n) where n is the size of the string (ignore extra time of string concatenation).

Time Complexity: O(N * 26) = O(N) where N is the size of the string, as it iterates through the string once and checks frequencies of each character in constant time.

Space Complexity: O(26) = O(1) for the frequency array of characters. The space is constant because the array size is fixed regardless of the input string size.

Ques: Sort characters by frequency [Leetcode 451]
Given a string s, sort it in decreasing order based on the frequency of the characters. The frequency of a character is the number of times it appears in the string.

Return the sorted string. If there are multiple answers, return any of them.

Example 1:
Input: s = "tree"
Output: "eert"
Explanation: 'e' appears twice while 'r' and 't' both appear once.
So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.

Example 2:
Input: s = "cccaaa"
Output: "aaaccc"
Explanation: Both 'c' and 'a' appear three times, so both "cccaaa" and "aaaccc" are valid answers.
Note that "cacaca" is incorrect, as the same characters must be together.

Example 3:
Input: s = "Aabb"
Output: "bbAa"
Explanation: "bbaA" is also a valid answer, but "Aabb" is incorrect.
Note that 'A' and 'a' are treated as two different characters.

Code:

```
class Solution {
    public String frequencySort(String s) {
        StringBuilder res = new StringBuilder();
        Map<Character, Integer> map = new HashMap<>();

        for (char c : s.toCharArray()) {
            map.put(c, map.getOrDefault(c, 0) + 1);
        }

        List<Map.Entry<Character, Integer>> list = new ArrayList<>(map.entrySet());

        Collections.sort(list, new Comparator<Map.Entry<Character, Integer>>() {
            public int compare(Map.Entry<Character, Integer> a, Map.Entry<Character,
Integer> b) {
                return b.getValue() - a.getValue();
            }
        });

        for (Map.Entry<Character, Integer> entry : list) {
            char c = entry.getKey();
            int freq = entry.getValue();
            res.append(String.valueOf(c).repeat(freq));
        }

        return res.toString();
    }
}
```

Explanation: Use a map to count character frequencies, sorts characters based on their frequencies, and reconstructs the string accordingly, resulting in characters arranged by frequency in descending order.

Time Complexity: O(N log N) for sorting, where N is the number of unique characters.

Space Complexity: O(N) for the map and auxiliary data structures used for sorting.

Example 1:
Input: words = ["i","love","leetcode","i","love","coding"], k = 2
Output: ["i","love"]
Explanation: "i" and "love" are the two most frequent words.
Note that "i" comes before "love" due to a lower alphabetical order.

Example 2:
Input: words = ["the","day","is","sunny","the","the","the","sunny","is","is"], k = 4
Output: ["the","is","sunny","day"]
Explanation: "the", "is", "sunny" and "day" are the four most frequent words, with the number of occurrence being 4, 3, 2 and 1 respectively.

Code:

```
class Solution {
    public List<String> topKFrequent(String[] word, int k) {
        Map<String, Integer> freq = new HashMap<>();
        for (String w : word) {
            freq.put(w, freq.getOrDefault(w, 0) + 1);
        }

        PriorityQueue<Map.Entry<String, Integer>> pq = new PriorityQueue<>(
            (a, b) -> a.getValue() == b.getValue() ?
a.getKey().compareTo(b.getKey()) : b.getValue() - a.getValue()
        );
```

```
        for (Map.Entry<String, Integer> entry : freq.entrySet()) {
            pq.offer(entry);
            if (pq.size() > k) {
                pq.poll();
            }
        }

        List<String> ans = new ArrayList<>();
        while (!pq.isEmpty()) {
            ans.add(pq.poll().getKey());
        }
        Collections.reverse(ans);

        return ans;
    }
}
```

Explanation: here we will make our own priority queue , in which first element store freq and second element store string in lexicographical order if freq is eqal ,,, for this we will use comparator, in which we will return true iffreq is higher , and in case same freq we return which have lexicographically greater

Time Complexity: O(N log K) for iterating through the words and maintaining the priority queue of size K, where N is the number of words.

Space Complexity: O(N + K) for storing word frequencies and the top K frequent words.

Ques: Longest happy string [Leetcode 1405]

A string s is called happy if it satisfies the following conditions:

s only contains the letters 'a', 'b', and 'c'.
s does not contain any of "aaa", "bbb", or "ccc" as a substring.
s contains at most a occurrences of the letter 'a'.
s contains at most b occurrences of the letter 'b'.
s contains at most c occurrences of the letter 'c'.
Given three integers a, b, and c, return the longest possible happy string. If there are multiple longest happy strings, return any of them. If there is no such string, return the empty string "".

A substring is a contiguous sequence of characters within a string.

Example 1:
Input: a = 1, b = 1, c = 7
Output: "ccaccbcc"
Explanation: "ccbccacc" would also be a correct answer.

Example 2:
Input: a = 7, b = 1, c = 0
Output: "aabaa"
Explanation: It is the only correct answer in this case.

Code:

```
class Solution {
    public String longestDiverseString(int a, int b, int c) {
        StringBuilder ans = new StringBuilder();
        PriorityQueue<Pair<Integer, Character>> pq = new PriorityQueue<>(
            (x, y) -> y.getKey() - x.getKey()
        );

        if (a > 0) {
            pq.offer(new Pair<>(a, 'a'));
        }
        if (b > 0) {
            pq.offer(new Pair<>(b, 'b'));
        }
        if (c > 0) {
            pq.offer(new Pair<>(c, 'c'));
        }

        while (!pq.isEmpty()) {
            int x = pq.peek().getKey();
            char ch = pq.poll().getValue();

            if (ans.length() >= 2) {
                int n = ans.length();
                if (ans.charAt(n - 1) == ans.charAt(n - 2) && ans.charAt(n - 1) == ch) {
                    if (pq.isEmpty()) {
                        return ans.toString();
                    }
                    int y = pq.peek().getKey();
                    char chy = pq.poll().getValue();
                    ans.append(chy);
                    y--;
                    if (y > 0) {
                        pq.offer(new Pair<>(y, chy));
                    }
                }
            }
            ans.append(ch);
            x--;
            if (x > 0) {
                pq.offer(new Pair<>(x, ch));
            }
        }

        return ans.toString();
    }
}
```

Explanation: Here approach is that the most frequent element will be added twice, rest will be inserted based upon the condition. Thus here we use priority queue here, which will give us the max element each time we pop and element.
Here we are taking priority queue of pair , because we want to know which elelment is the maximum one.
Now, just dry run the entire code you will get the answer.

Time Complexity: O(N log N) where N is the sum of a, b, and c, for maintaining the priority queue.

Space Complexity: O(N) for the priority queue storing characters and their frequencies.

Ques: Furthest Building you can reach [Leetcode 1642]
You are given an integer array heights representing the heights of buildings, some bricks, and some ladders.

You start your journey from building 0 and move to the next building by possibly using bricks or ladders.

While moving from building i to building i+1 (0-indexed),

If the current building's height is greater than or equal to the next building's height, you do not need a ladder or bricks.
If the current building's height is less than the next building's height, you can either use one ladder or (h[i+1] - h[i]) bricks.
Return the furthest building index (0-indexed) you can reach if you use the given ladders and bricks optimally.

Example 1:

Input: heights = [4,2,7,6,9,14,12], bricks = 5, ladders = 1

Output: 4
Explanation: Starting at building 0, you can follow these steps:
- Go to building 1 without using ladders nor bricks since 4 >= 2.
- Go to building 2 using 5 bricks. You must use either bricks or ladders because 2 < 7.
- Go to building 3 without using ladders nor bricks since 7 >= 6.
- Go to building 4 using your only ladder. You must use either bricks or ladders because 6 < 9.
It is impossible to go beyond building 4 because you do not have any more bricks or ladders.

Example 2:
Input: heights = [4,12,2,7,3,18,20,3,19], bricks = 10, ladders = 2
Output: 7

Example 3:
Input: heights = [14,3,19,3], bricks = 17, ladders = 0
Output: 3

Code:

```
class Solution {
    public int furthestBuilding(int[] A, int bricks, int ladders) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        for (int i = 0; i < A.length - 1; i++) {
            int d = A[i + 1] - A[i];
            if (d > 0) {
                pq.offer(-d);
            }
            if (pq.size() > ladders) {
                bricks += pq.poll();
            }
            if (bricks < 0) {
                return i;
            }
        }

        return A.length - 1;
    }
}
```

Explanation: Heap heap store k height difference that we need to use ladders.
Each move, if the height difference d > 0,
we push d into the priority queue pq.
If the size of queue exceed ladders,
it means we have to use bricks for one move.
Then we pop out the smallest difference, and reduce bricks.
If bricks < 0, we can't make this move, then we return current index i.
If we can reach the last building, we return A.length - 1.

Time Complexity: O(NlogK), for maintaining the priority queue.

Space Complexity: O(K) for the priority queue.

Ques : Distant barcodes [Leetcode 1054]
In a warehouse, there is a row of barcodes, where the ith barcode is barcodes[i].

Rearrange the barcodes so that no two adjacent barcodes are equal. You may return any answer, and it is guaranteed an answer exists.

Example 1:
Input: barcodes = [1,1,1,2,2,2]
Output: [2,1,2,1,2,1]

Example 2:
Input: barcodes = [1,1,1,1,2,2,3,3]
Output: [1,3,1,3,1,2,1,2]

Code:

```
class Solution {
    public int[] rearrangeBarcodes(int[] barcodes) {
        int[] hash = new int[10001];
        for (int i = 0; i < barcodes.length; i++) {
            hash[barcodes[i]]++;
        }

        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> b[0] - a[0]);
        for (int i = 0; i <= 10000; i++) {
            if (hash[i] != 0) {
                pq.offer(new int[]{hash[i], i});
            }
        }

        int[] ans = new int[barcodes.length];
        int i = 0;
        while (!pq.isEmpty()) {
            int[] top = pq.poll();
            int count = top[0];
            int node = top[1];
            while (count > 0) {
                ans[i] = node;
                i += 2;
                if (i >= barcodes.length) {
                    i = 1;
                }
                count--;
            }
        }
        return ans;
    }
}
```

Explanation: In this method I have gone through even odd rule :Travelling from the higher freq to lower freq priority_queue for storing the frequencies As it is always valid firstly store all the higher freq at even position As it goes beyond size turn it to odd positions and as it is always valid We are not going to have the problem.

Time Complexity: O(N log N) for iterating through barcodes and populating the priority queue, where N is the size of barcodes.

Space Complexity: O(N) for the hash array & priority queues

Ques : Construct String With Repeat Limit [Leetcode 2182]
You are given a string s and an integer repeatLimit. Construct a new string repeatLimitedString using the characters of s such that no letter appears more than repeatLimit times in a row. You do not have to use all characters from s.

Return the lexicographically largest repeatLimitedString possible.

A string a is lexicographically larger than a string b if in the first position where a and b differ, string a has a letter that appears later in the alphabet than the corresponding letter in b. If the first min(a.length, b.length) characters do not differ, then the longer string is the lexicographically larger one.

Example 1:
Input: s = "cczazcc", repeatLimit = 3
Output: "zzcccac"
Explanation: We use all of the characters from s to construct the repeatLimitedString "zzcccac".
The letter 'a' appears at most 1 time in a row.
The letter 'c' appears at most 3 times in a row.
The letter 'z' appears at most 2 times in a row.
Hence, no letter appears more than repeatLimit times in a row and the string is a valid repeatLimitedString.
The string is the lexicographically largest repeatLimitedString possible so we return "zzcccac".
Note that the string "zzcccca" is lexicographically larger but the letter 'c' appears more than 3 times in a row, so it is not a valid repeatLimitedString.

Example 2:
Input: s = "aababab", repeatLimit = 2
Output: "bbabaa"
Explanation: We use only some of the characters from s to construct the repeatLimitedString "bbabaa".
The letter 'a' appears at most 2 times in a row.
The letter 'b' appears at most 2 times in a row.
Hence, no letter appears more than repeatLimit times in a row and the string is a valid repeatLimitedString.
The string is the lexicographically largest repeatLimitedString possible so we return "bbabaa".
Note that the string "bbabaaa" is lexicographically larger but the letter 'a' appears more than 2 times in a row, so it is not a valid repeatLimitedString.

Code:

```
class Solution {
    public String repeatLimitedString(String str, int k) {
        StringBuilder ans = new StringBuilder();
        Map<Character, Integer> myMap = new HashMap<>();

        for (char s : str.toCharArray()) {
            myMap.put(s, myMap.getOrDefault(s, 0) + 1);
        }

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

```
        pq.addAll(myMap.entrySet());

        while (!pq.isEmpty()) {
            Map.Entry<Character, Integer> entry = pq.poll();
            int i = 0;
            char ch = entry.getKey();
            int count = entry.getValue();

            while (i < k && i < count) {
                ans.append(ch);
                i++;
            }

            if (count > k && !pq.isEmpty()) {
                Map.Entry<Character, Integer> nextEntry = pq.poll();
                char nextCh = nextEntry.getKey();

                ans.append(nextCh);
                if (nextEntry.getValue() > 1) {
                    pq.offer(new AbstractMap.SimpleEntry<>(nextCh,
nextEntry.getValue() - 1));
                }
                pq.offer(new AbstractMap.SimpleEntry<>(ch, count - k));
            }
        }

        return ans.toString();
    }
}
```

Explanation:
Calculate character frequencies using a hashmap or array.
Add character-frequency pairs to a Priority Queue.
Extract the top character from the Priority Queue, adding it to the result until reaching the minimum of its frequency or a repeat limit.
If the frequency exceeds the limit, check for the next character in the queue.
If the next character's frequency allows, add it to the result and update its frequency in the queue.
Push back the first character with reduced frequency, considering it for the next iteration.
Repeat until the queue is empty, ensuring not to push elements with frequencies exceeding the limit and no subsequent characters available.

Time Complexity: O(N log N) for inserting elements into the priority queue.

Space Complexity: O(N) for the priority queue storing character-frequency pairs.

Ques : Minimum Operations to Halve Array Sum [Leetcode 2208]

You are given an array nums of positive integers. In one operation, you can choose any number from nums and reduce it to exactly half the number. (Note that you may choose this reduced number in future operations.)

Return the minimum number of operations to reduce the sum of nums by at least half.

Example 1:
Input: nums = [5,19,8,1]
Output: 3
Explanation: The initial sum of nums is equal to 5 + 19 + 8 + 1 = 33.
The following is one of the ways to reduce the sum by at least half:
Pick the number 19 and reduce it to 9.5.
Pick the number 9.5 and reduce it to 4.75.
Pick the number 8 and reduce it to 4.
The final array is [5, 4.75, 4, 1] with a total sum of 5 + 4.75 + 4 + 1 = 14.75.
The sum of nums has been reduced by 33 - 14.75 = 18.25, which is at least half of the initial sum, 18.25 >= 33/2 = 16.5.
Overall, 3 operations were used so we return 3.
It can be shown that we cannot reduce the sum by at least half in less than 3 operations.

Example 2:
Input: nums = [3,8,20]
Output: 3
Explanation: The initial sum of nums is equal to 3 + 8 + 20 = 31.
The following is one of the ways to reduce the sum by at least half:
Pick the number 20 and reduce it to 10.
Pick the number 10 and reduce it to 5.
Pick the number 3 and reduce it to 1.5.
The final array is [1.5, 8, 5] with a total sum of 1.5 + 8 + 5 = 14.5.
The sum of nums has been reduced by 31 - 14.5 = 16.5, which is at least half of the initial sum, 16.5 >= 31/2 = 15.5.
Overall, 3 operations were used so we return 3.
It can be shown that we cannot reduce the sum by at least half in less than 3 operations.

Code:

```
class Solution {
    public int halveArray(List<Integer> nums) {
        PriorityQueue<Double> pq = new
PriorityQueue<>(Collections.reverseOrder());
        double tot = 0, runningSum;
        for (int i : nums) {
            double temp = i;
            tot += temp;
            pq.offer(temp);
        }
        runningSum = tot;
        int ans = 0;
        while (runningSum * 2 > tot) {
            double t = pq.poll();
            runningSum -= t / 2;
            pq.offer(t / 2);
            ans++;
        }
        return ans;
    }
}
```

Explanation:- Put all the elements in maxHeap.
Decrease the topmost element by half and then push the decreased element back into maxHeap.
Why are we decreasing greatest element?? :- because it's half is greatest among all elements half.Greater the value greater the half.

Time Complexity: O(N log N) for populating and managing the priority queue where N is the size of the nums vector.
Space Complexity: O(N) for the priority queue storing the elements, proportional to the input size.

Ques : Maximum performance of a team [Leetcode 1383]
You are given two integers n and k and two integer arrays speed and efficiency both of length n. There are n engineers numbered from 1 to n. speed[i] and efficiency[i] represent the speed and efficiency of the ith engineer respectively.

Choose at most k different engineers out of the n engineers to form a team with the maximum performance.

The performance of a team is the sum of its engineers' speeds multiplied by the minimum efficiency among its engineers.

Return the maximum performance of this team. Since the answer can be a huge number, return it modulo 109 + 7.

Example 1:
Input: n = 6, speed = [2,10,3,1,5,8], efficiency = [5,4,3,9,7,2], k = 2
Output: 60
Explanation: We have the maximum performance of the team by selecting engineer 2 (with speed=10 and efficiency=4) and engineer 5 (with speed=5 and efficiency=7). That is, performance = (10 + 5) * min(4, 7) = 60.

Example 2:
Input: n = 6, speed = [2,10,3,1,5,8], efficiency = [5,4,3,9,7,2], k = 3
Output: 68
Explanation: This is the same example as the first but k = 3. We can select engineer 1, engineer 2 and engineer 5 to get the maximum performance of the team. That is, performance = (2 + 10 + 5) * min(5, 4, 7) = 68.

Example 3:
Input: n = 6, speed = [2,10,3,1,5,8], efficiency = [5,4,3,9,7,2], k = 4
Output: 72

Code:

```
class Solution {
    public int maxPerformance(int n, int[] speed, int[] efficiency, int k) {
        int mod = (int) 1e9 + 7;
        List<int[]> pairs = new ArrayList<>();

        for (int i = 0; i < n; i++) {
            pairs.add(new int[]{efficiency[i], speed[i]});
        }

        pairs.sort((a, b) -> b[0] - a[0]);
```

```
        PriorityQueue<Integer> pq = new PriorityQueue<>(k, Comparator.naturalOrder());
        long totalSpeed = 0;
        long maxPerformance = 0;

        for (int[] pair : pairs) {
            int e = pair[0];
            int s = pair[1];

            totalSpeed += s;
            pq.offer(s);

            if (pq.size() > k) {
                totalSpeed -= pq.poll();
            }

            maxPerformance = Math.max(maxPerformance, totalSpeed * e);
        }

        return (int) (maxPerformance % mod);
    }
}
```

Explananation: For maximum performance of a team we take the engineers by their efficiency i.e those have maximum efficiency we take them first.

but every engineer have their speed corresponding to their efficiency. So we make a pair vector tmp in which tmp[i]=[efficiency[i],speed[i]];

now sort this pair vector in reverse order as we have to take first which having the higher efficiency
here we use a min priority queue to tackle the speed of engineer. we only hire atmost k egineers to maximise the performance.
so if the size of priority queue greater than k then we remove the engineers with minimum speed from priority queue

Time Complexity: O(N log N) for sorting the vector 'tmp' using sort() and iterating through 'tmp' once. N is the number of elements.

Space Complexity: O(N) for the priority queue and the temporary vector 'tmp', both scaling linearly with the input size.

Ques : Merge k sorted lists [Leetcode 23]
You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
[
 1->4->5,
 1->3->4,
 2->6
]
merging them into one sorted list:
1->1->2->3->4->4->5->6

Example 2:
Input: lists = []
Output: []

Example 3:
Input: lists = [[]]
Output: []

Code:

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

public class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        int n = lists.length;
        ListNode newHead = new ListNode(0);
        ListNode tail = newHead;

        PriorityQueue<ListNode> pq = new PriorityQueue<>((a, b) -> a.val - b.val);

        for (ListNode list : lists) {
            if (list != null) {
                pq.offer(list);
            }
        }

        while (!pq.isEmpty()) {
            ListNode node = pq.poll();
            tail.next = node;
            tail = tail.next;

            if (node.next != null) {
                pq.offer(node.next);
            }
        }

        return newHead.next;
    }
}
```

Explanation:
Method 1 (Brute Force)
A simple way of doing this is to make a big list by iterating over the vector and then sorting it using any of the sorting techniques lijke merge sort or quick sort.

Time Complexity : O(n * k) [for iterating over vector of size n and average linked list size k) + O(p * log p) [sorting the big list of size p = n * k)

Method 2 (Optimized)
Another way of doing this is to make use of the fact that linked lists are k sorted. We'll use a min priority queue. Since we know that linked list is k sorted , we first push head of all the lists in the priority queue. The minimum of these will be the first node of the list. After that we move this min node to its next (if exists) and push that next node again to the priority queue. This way we keep adding nodes to the resultant list and the moment priority queue becomes empty, we get our resulting list.
The code for same is give above.

Time Complexity : O(k* log k) [for pushing elements in pq] + O(N * log k) [logK for popping min node and pushing it's next back, and we are doing this N times]

Space Complexity : O(k) [at any momemnt pq will have max k elements]

Ques : Sliding window median [Leetcode 480]
The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle values.

For examples, if arr = [2,3,4], the median is 3.
For examples, if arr = [1,2,3,4], the median is (2 + 3) / 2 = 2.5.
You are given an integer array nums and an integer k. There is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the median array for each window in the original array. Answers within 10-5 of the actual value will be accepted.

Example 1:
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [1.00000,-1.00000,-1.00000,3.00000,5.00000,6.00000]
Explanation:

Window position	Median
[1 3 -1] -3 5 3 6 7	1
1 [3 -1 -3] 5 3 6 7	-1
1 3 [-1 -3 5] 3 6 7	-1
1 3 -1 [-3 5 3] 6 7	3
1 3 -1 -3 [5 3 6] 7	5
1 3 -1 -3 5 [3 6 7]	6

Example 2:
Input: nums = [1,2,3,4,2,3,1,4,2], k = 3
Output: [2.00000,3.00000,3.00000,3.00000,2.00000,3.00000,2.00000]

Code:

```
import java.util.*;

class Solution {
    public double[] medianSlidingWindow(int[] nums, int k) {
        List<Double> median = new ArrayList<>();
        Map<Integer, Integer> mp = new HashMap<>();
        PriorityQueue<Integer> maxHeap = new
PriorityQueue<>(Collections.reverseOrder());
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int i = 0; i < k; i++) {
            maxHeap.offer(nums[i]);
        }

        for (int i = 0; i < k / 2; i++) {
            minHeap.offer(maxHeap.poll());
        }

        for (int i = k; i < nums.length; i++) {
            if (k % 2 != 0) {
                median.add((double) maxHeap.peek());
            } else {
                median.add((maxHeap.peek() + minHeap.peek()) / 2.0);
            }
```

```
            if (p <= maxHeap.peek()) {
                balance--;
                if (p == maxHeap.peek()) {
                    maxHeap.poll();
                } else {
                    mp.put(p, mp.getOrDefault(p, 0) + 1);
                }
            } else {
                balance++;
                if (p == minHeap.peek()) {
                    minHeap.poll();
                } else {
                    mp.put(p, mp.getOrDefault(p, 0) + 1);
                }
            }

            if (!maxHeap.isEmpty() && q <= maxHeap.peek()) {
                maxHeap.offer(q);
                balance++;
            } else {
                minHeap.offer(q);
                balance--;
            }

            if (balance > 0) {
                minHeap.offer(maxHeap.poll());
            } else if (balance < 0) {
                maxHeap.offer(minHeap.poll());
            }

            while (!maxHeap.isEmpty() && mp.getOrDefault(maxHeap.peek(), 0) > 0) {
                mp.put(maxHeap.peek(), mp.get(maxHeap.peek()) - 1);
                maxHeap.poll();
            }

            while (!minHeap.isEmpty() && mp.getOrDefault(minHeap.peek(), 0) > 0) {
                mp.put(minHeap.peek(), mp.get(minHeap.peek()) - 1);
                minHeap.poll();
            }
        }

        if (k % 2 != 0) {
            median.add((double) maxHeap.peek());
        } else {
            median.add((maxHeap.peek() + minHeap.peek()) / 2.0);
        }

        return median.stream().mapToDouble(Double::doubleValue).toArray();
    }
}
```

Explanation: This code computes the median of a sliding window in an array by using two heaps (max-heap and min-heap) and a map to maintain and manage the elements within the window. Here's a step-by-step explanation:

1. Initialization:

- Two priority queues (maxHeap and minHeap) are used to maintain elements in descending and ascending order respectively, mimicking the behavior of max and min heaps.
- A map mp tracks the elements' occurrences to facilitate their removal.

2. Sliding Window Approach:
- The initial window of size k is processed to populate the heaps and establish an initial balance.
- Iterating through the remaining elements:
  - Median calculation based on the heaps.
  - Handling element removal and insertion to maintain the window and balance the heaps.
  - Adjusting the heaps by popping elements and updating the map accordingly.

3. Median Calculation:
- After processing each window, the median is computed based on the values present in the heaps.
- The calculated median is added to the median vector, and finally returned as the result.

Time Complexity: O(N log K) for iterating through the nums array and managing the heaps, where N is the size of nums and K is the window size.

Space Complexity: O(N) for the median vector storing the results.

Ques : Construct Target Array With Multiple Sums [Leetcode 1354]
You are given an array target of n integers. From a starting array arr consisting of n 1's, you may perform the following procedure :

let x be the sum of all elements currently in your array.
choose index i, such that 0 <= i < n and set the value of arr at index i to x.
You may repeat this procedure as many times as needed.
Return true if it is possible to construct the target array from arr, otherwise, return false.

Example 1:
Input: target = [9,3,5]
Output: true
Explanation: Start with arr = [1, 1, 1]
[1, 1, 1], sum = 3 choose index 1
[1, 3, 1], sum = 5 choose index 2
[1, 3, 5], sum = 9 choose index 0
[9, 3, 5] Done

Example 2:
Input: target = [1,1,1,2]
Output: false
Explanation: Impossible to create target array from [1,1,1,1].

Example 3:
Input: target = [8,5]
Output: true

Code:

```
class Solution {
    public boolean isPossible(int[] target) {
        PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());
        long sum = 0;

        for (int num : target) {
            pq.offer(num);
            sum += num;
        }
```

```
        while (pq.peek() != 1) {
            sum -= pq.peek();

            if (sum == 0 || sum ⩾ pq.peek()) {
                return false;
            }

            int old = pq.poll() % (int)sum;

            if (sum != 1 && old == 0) {
                return false;
            }

            pq.offer(old);
            sum += old;
        }

        return true;
    }
}
```

Explanation: The problem says that we need to check if we did the given operation on any element any number of times on an array with all 1 will we get the target array or not. So if we can do the reverse operations on the target array we should get the array with all 1.

Now in each step the sum of the old array is replaced with any element of the old array to get current array. So the greatest element of current array is the sum of old array.
Let old array be [a, b, c, d] and new array after doing 1 operation may be [a, b, c, (a+b+c+d)].
We have to find d from here to get the old array back. So if we thake sum of the current array and exclude the greatest element i.e. sum of previous array we get (a+b+c). Now if we minus it from the previous sum we get (a+b+c+d) - (a+b+c) = d. We are getting old element back.

Time Complexity: O(N log N) for the priority queue operations, where N is the size of the 'target' vector.

Space Complexity: O(N) for the priority queue storing the elements, proportional to the input size.

Ques: Find the Kth Smallest Sum of a Matrix With Sorted Rows [Leetcode 1439]
You are given an m x n matrix mat that has its rows sorted in non-decreasing order and an integer k.

You are allowed to choose exactly one element from each row to form an array.
Return the kth smallest array sum among all possible arrays.

Example 1:
Input: mat = [[1,3,11],[2,4,6]], k = 5
Output: 7
Explanation: Choosing one element from each row, the first k smallest sum are:
[1,2], [1,4], [3,2], [3,4], [1,6]. Where the 5th sum is 7.

Example 2:
Input: mat = [[1,3,11],[2,4,6]], k = 9
Output: 17

Example 3:
Input: mat = [[1,10,10],[1,4,5],[2,3,6]], k = 7
Output: 9
Explanation: Choosing one element from each row, the first k smallest sum are:
[1,1,2], [1,1,3], [1,4,2], [1,4,3], [1,1,6], [1,5,2], [1,5,3]. Where the 7th sum is 9.

Code:

```
class Solution {
    public int kthSmallest(List<List<Integer>> mat, int k) {
        PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());
        int res = 0;

        for (List<Integer> m : mat) {
            res += m.get(0);
            PriorityQueue<Integer> pqCopy = new PriorityQueue<>(pq);

            for (int i = 1; i < m.size() && i < k; ++i) {
                int d = m.get(i) - m.get(0);

                if (pq.size() == k - 1 && d > pq.peek()) {
                    continue;
                }

                pq.offer(d);

                if (pq.size() == k) {
                    pq.poll();
                }

                PriorityQueue<Integer> tpq = new PriorityQueue<>(pqCopy);

                while (!tpq.isEmpty()) {
                    pq.offer(d + tpq.peek());

                    if (pq.size() == k) {
                        pq.poll();
                    }

                    tpq.poll();
                }
            }
        }

        return res + pq.peek();
    }
}
```

Explanation: res is the smallest sum. Then we push in the distance between m[0] and m[i] into the priority queue We only keep at most k - 1 elements in the priority queue.

Time Complexity: O(M * N * log K) where M is the number of rows in 'mat', N is the number of elements in each row, and K is the input 'k'.

Space Complexity: O(K) for the priority queue storing the smallest differences between elements in the matrix.

Ques : Find the median of the given data stream [Leetcode 295]

Code:

```
class MedianFinder {
    PriorityQueue<Integer> firstQ; // max_heap for the first half
    PriorityQueue<Integer> secQ; // min_heap for the second half

    public MedianFinder() {
        firstQ = new PriorityQueue<>(Collections.reverseOrder());
        secQ = new PriorityQueue<>();
    }

    // Adds a number into the data structure.
    public void addNum(int num) {
        if (firstQ.isEmpty() || firstQ.peek() > num) {
            firstQ.offer(num); // if it belongs to the smaller half
        } else {
            secQ.offer(num);
        }

        // Rebalance the two halves to make sure the length difference is no larger
than 1
        if (firstQ.size() > secQ.size() + 1) {
            secQ.offer(firstQ.poll());
        } else if (firstQ.size() + 1 < secQ.size()) {
            firstQ.offer(secQ.poll());
        }
    }

    // Returns the median of the current data stream
    public double findMedian() {
        if (firstQ.size() == secQ.size()) {
            return firstQ.isEmpty() ? 0 : (firstQ.peek() + secQ.peek()) / 2.0;
        } else {
            return (firstQ.size() > secQ.size()) ? firstQ.peek() : secQ.peek();
        }
    }
}
```

Explanation: The idea is to use two heaps (one max heap, one mn heap) to save the input data. firstQ is a max_heap to save the first half of the data with smaller values, and secQ is a min_heap to save the second half of the data with bigger values. Everytime when inserting a new value, we first compare if it is smaller than the top of firstQ (the largest value of the first half), if so, insert into firstQ. Otherwise, it belongs to the second half. After inserting, we have to balance the first half and the second half to make sure either they have the same length or the length difference is only 1.
The median will be the mean of two top elements (when they have the same length) or the top element of the queue with a larger length.

Time Complexity:

- **Adding Number (addNum()):** O(log N) where N is the number of elements in the larger of the two heaps, due to heap operations.
- **Finding Median (findMedian()):** O(1) for returning the current median, as it involves accessing the top elements of the heaps.

Space Complexity: O(N) where N is the number of elements in the stream, as both priority queues store the elements, proportional to the input size.

PW SKILLS

THANK YOU !

PW SKILLS