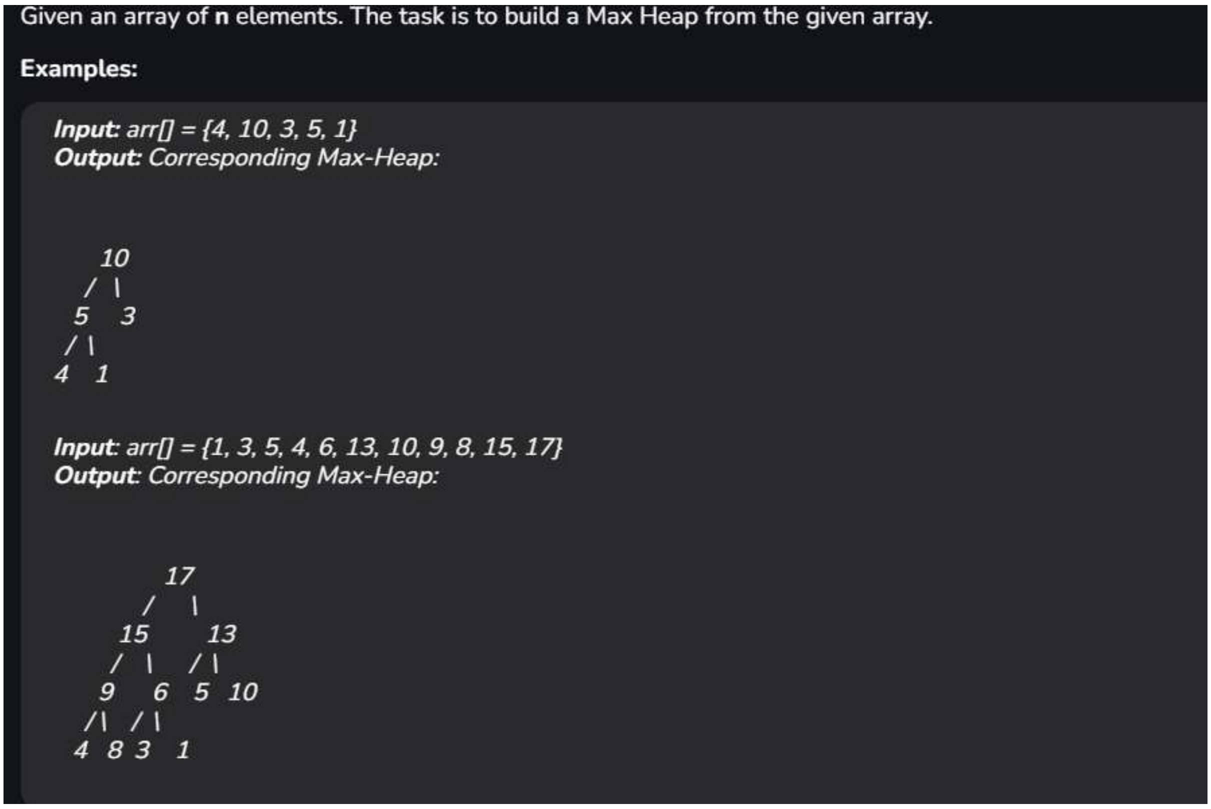
Heap

1.Building Heap From Array



Given an array of **n** elements. The task is to build a Max Heap from the given array.

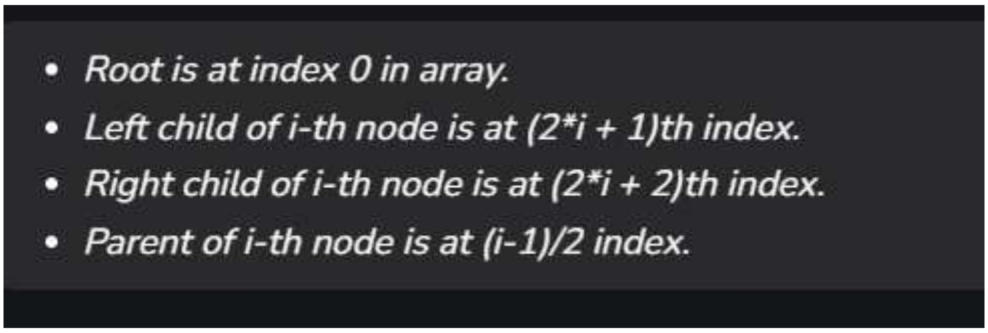
Examples:

Input: *arr[] = {4, 10, 3, 5, 1}*
Output: *Corresponding Max-Heap:*

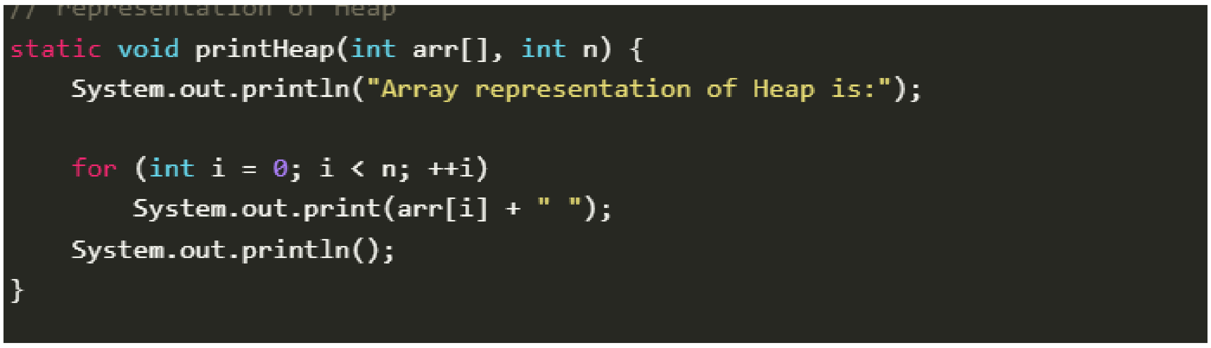
```
     10
    / \
   5   3
  / \
 4   1
```

Input: *arr[] = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}*
Output: *Corresponding Max-Heap:*

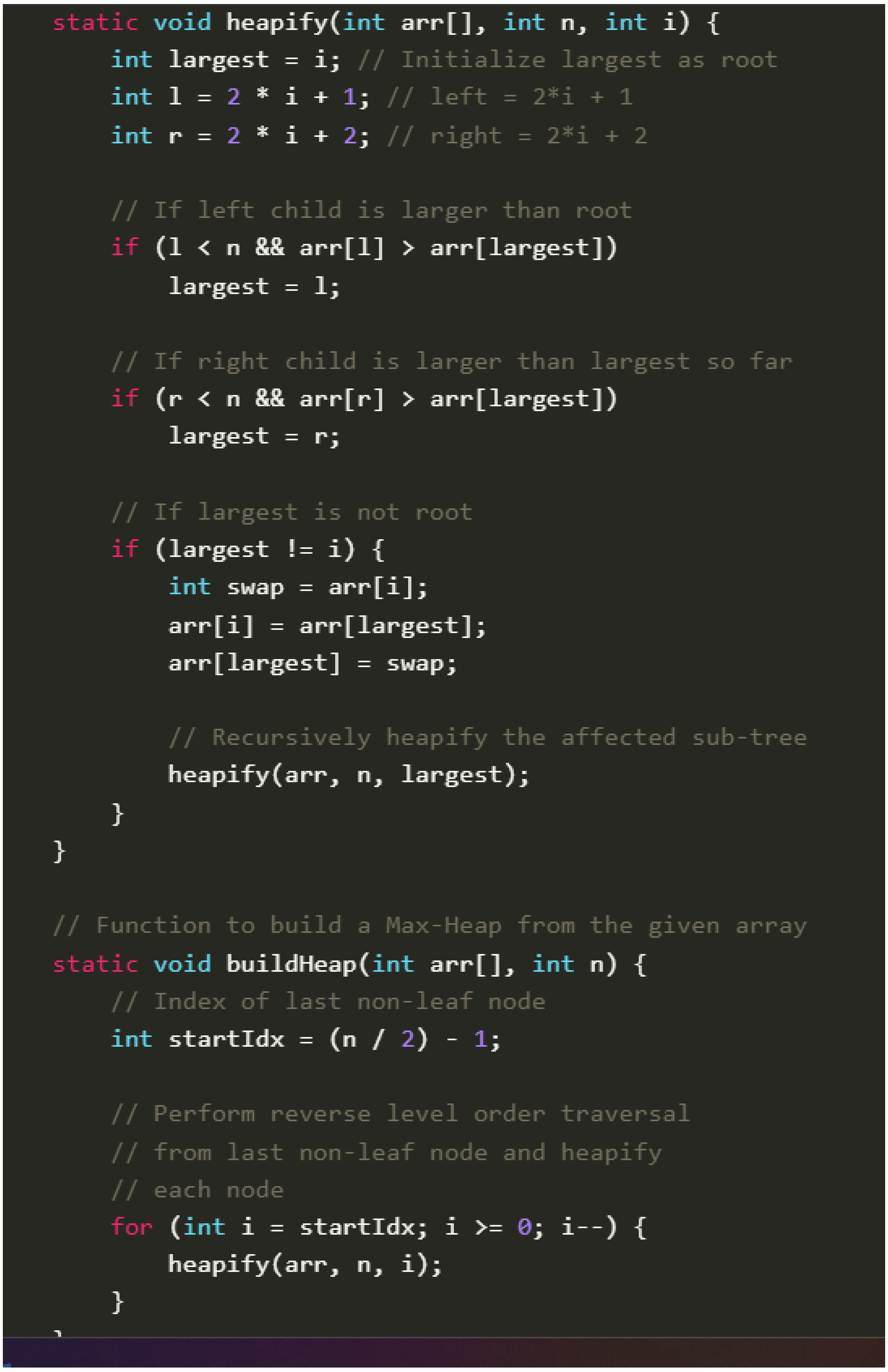
```
          17
        /    \
      15      13
     /  \    / \
    9    6  5  10
   / \  / \
  4  8 3   1
```



- *Root is at index 0 in array.*
- *Left child of i-th node is at (2*i + 1)th index.*
- *Right child of i-th node is at (2*i + 2)th index.*
- *Parent of i-th node is at (i-1)/2 index.*

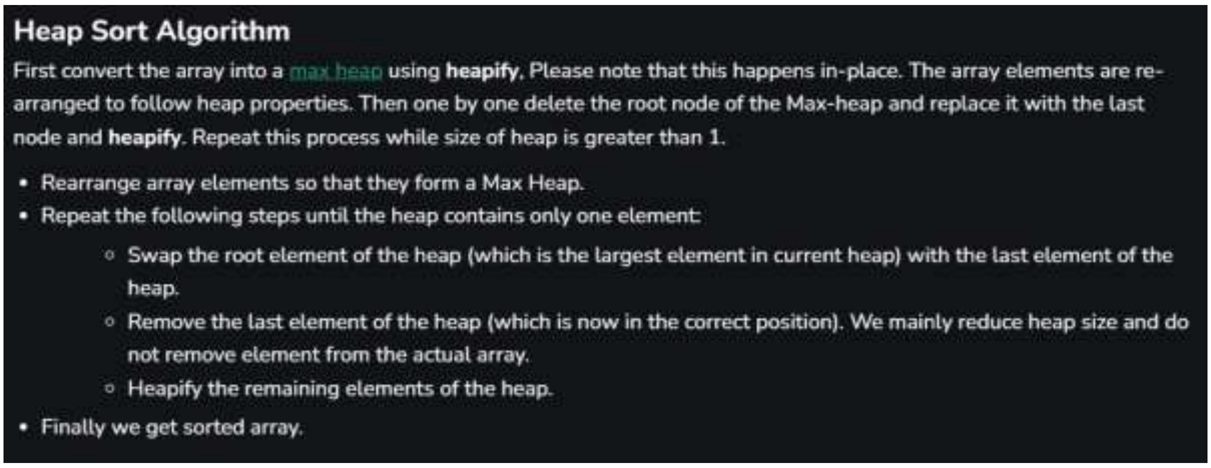


```java
// representation of heap
static void printHeap(int arr[], int n) {
    System.out.println("Array representation of Heap is:");

    for (int i = 0; i < n; ++i)
        System.out.print(arr[i] + " ");
    System.out.println();
}
```



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

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

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

    // 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);
    }
}

// Function to build a Max-Heap from the given array
static void buildHeap(int arr[], int n) {
    // Index of last non-leaf node
    int startIdx = (n / 2) - 1;

    // Perform reverse level order traversal
    // from last non-leaf node and heapify
    // each node
    for (int i = startIdx; i >= 0; i--) {
        heapify(arr, n, i);
    }
}
```

2.Heap Sort

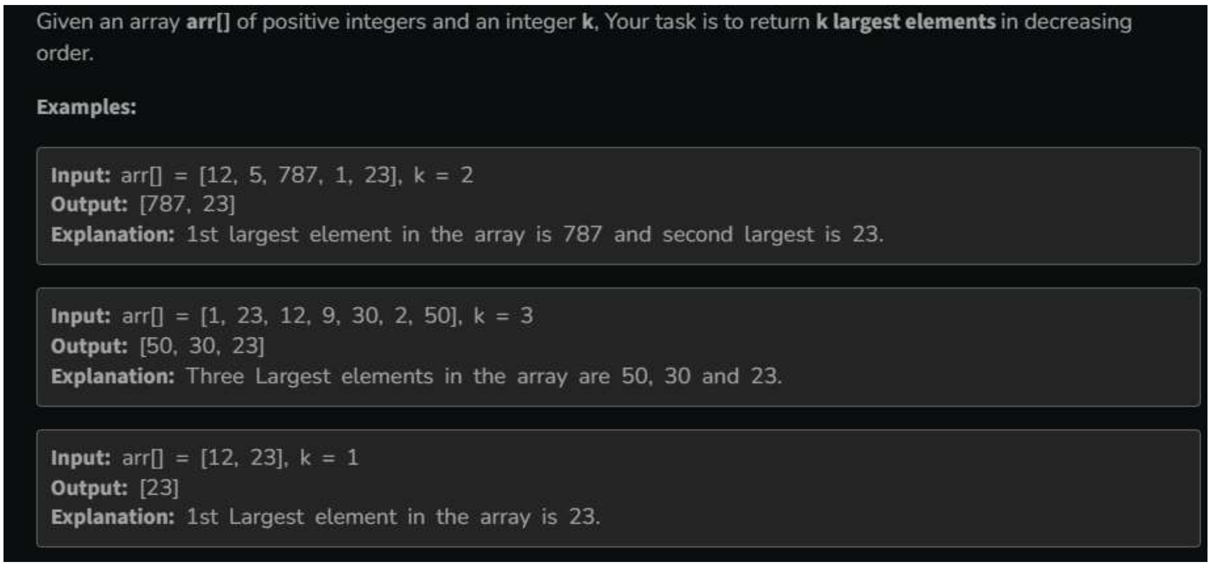


Heap Sort Algorithm

First convert the array into a max heap using **heapify**, Please note that this happens in-place. The array elements are re-arranged to follow heap properties. Then one by one delete the root node of the Max-heap and replace it with the last node and **heapify**. Repeat this process while size of heap is greater than 1.

- Rearrange array elements so that they form a Max Heap.
- Repeat the following steps until the heap contains only one element:
  - Swap the root element of the heap (which is the largest element in current heap) with the last element of the heap.
  - Remove the last element of the heap (which is now in the correct position). We mainly reduce heap size and do not remove element from the actual array.
  - Heapify the remaining elements of the heap.
- Finally we get sorted array.

3.Kth Largest Elements

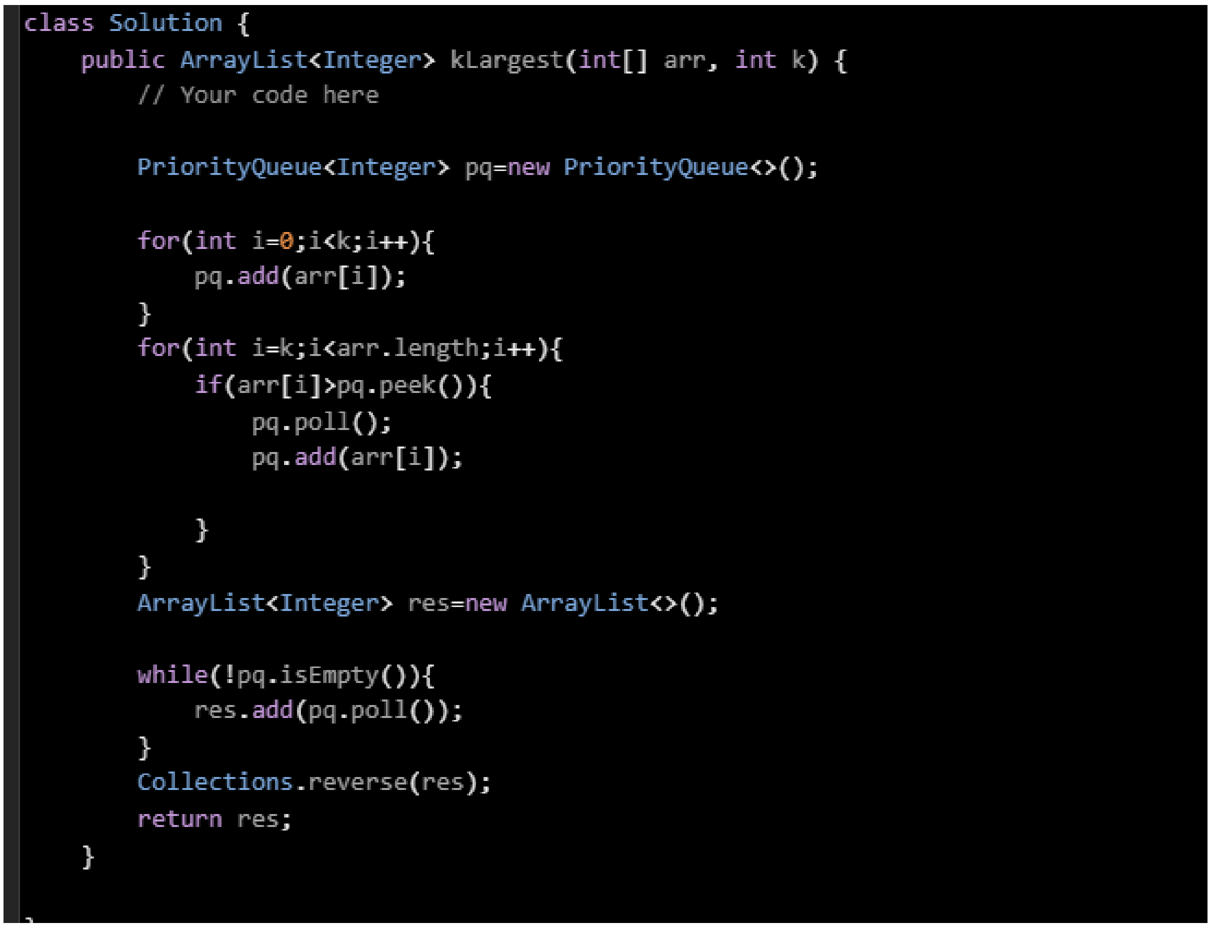

Given an array **arr[]** of positive integers and an integer **k**, Your task is to return **k largest elements** in decreasing order.

Examples:

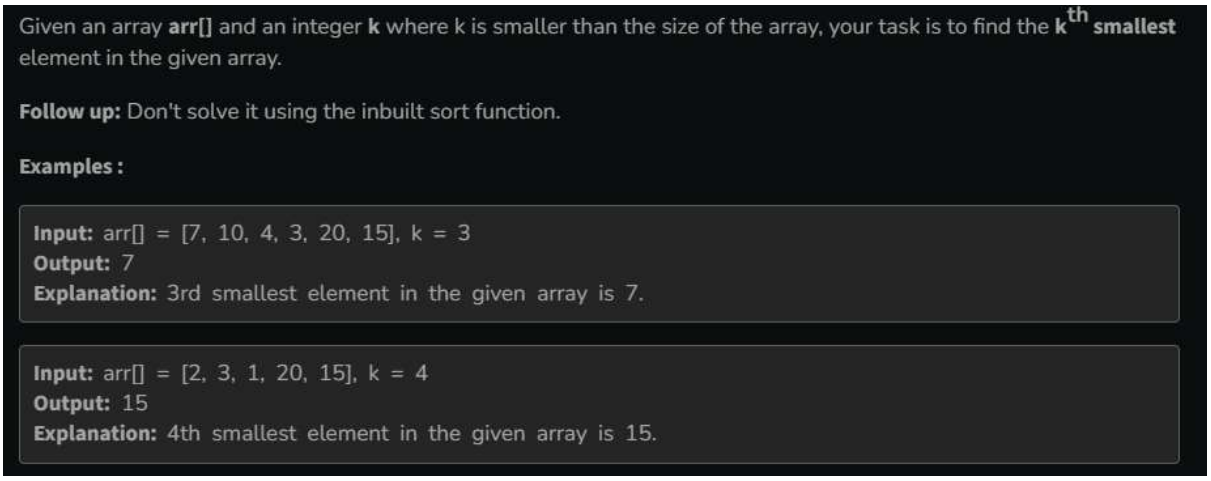
```
Input: arr[] = [12, 5, 787, 1, 23], k = 2
Output: [787, 23]
Explanation: 1st largest element in the array is 787 and second largest is 23.
```

```
Input: arr[] = [1, 23, 12, 9, 30, 2, 50], k = 3
Output: [50, 30, 23]
Explanation: Three Largest elements in the array are 50, 30 and 23.
```

```
Input: arr[] = [12, 23], k = 1
Output: [23]
Explanation: 1st Largest element in the array is 23.
```


```java
class Solution {
    public ArrayList<Integer> kLargest(int[] arr, int k) {
        // Your code here

        PriorityQueue<Integer> pq=new PriorityQueue<>();

        for(int i=0;i<k;i++){
            pq.add(arr[i]);
        }
        for(int i=k;i<arr.length;i++){
            if(arr[i]>pq.peek()){
                pq.poll();
                pq.add(arr[i]);

            }
        }
        ArrayList<Integer> res=new ArrayList<>();

        while(!pq.isEmpty()){
            res.add(pq.poll());
        }
        Collections.reverse(res);
        return res;
    }

}
```

4.Kth Smallest



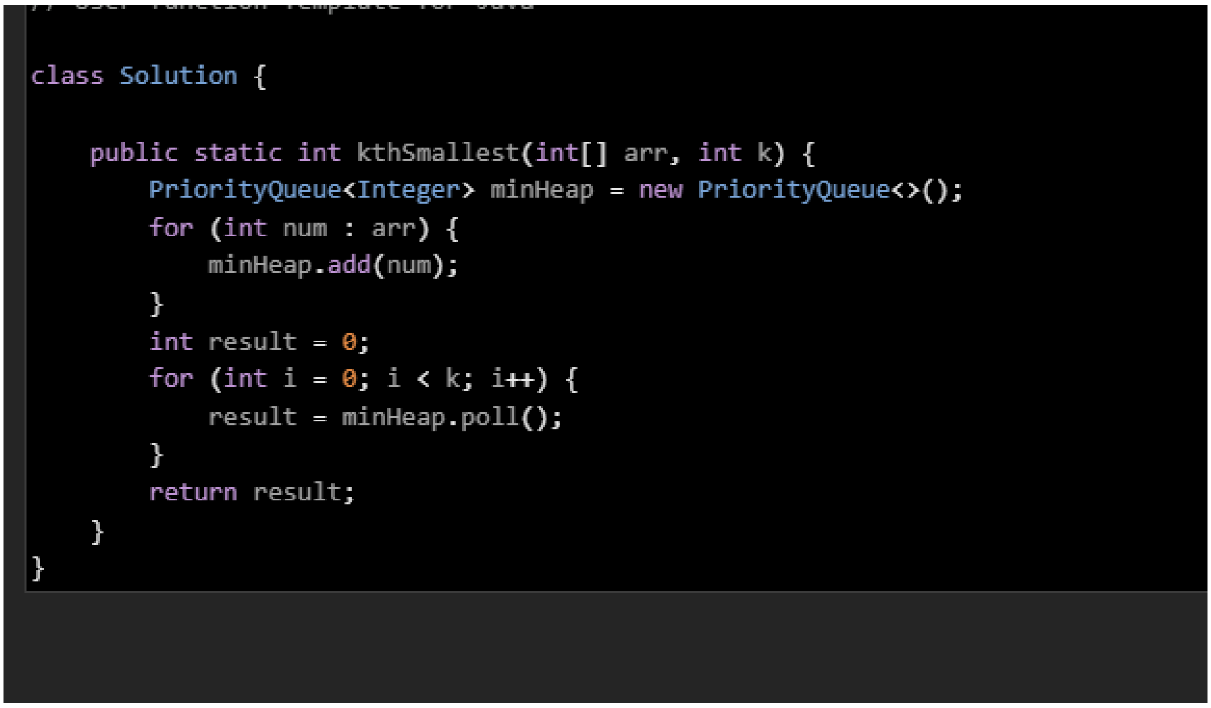
Given an array **arr[]** and an integer **k** where k is smaller than the size of the array, your task is to find the **kth smallest** element in the given array.

Follow up: Don't solve it using the inbuilt sort function.

Examples :

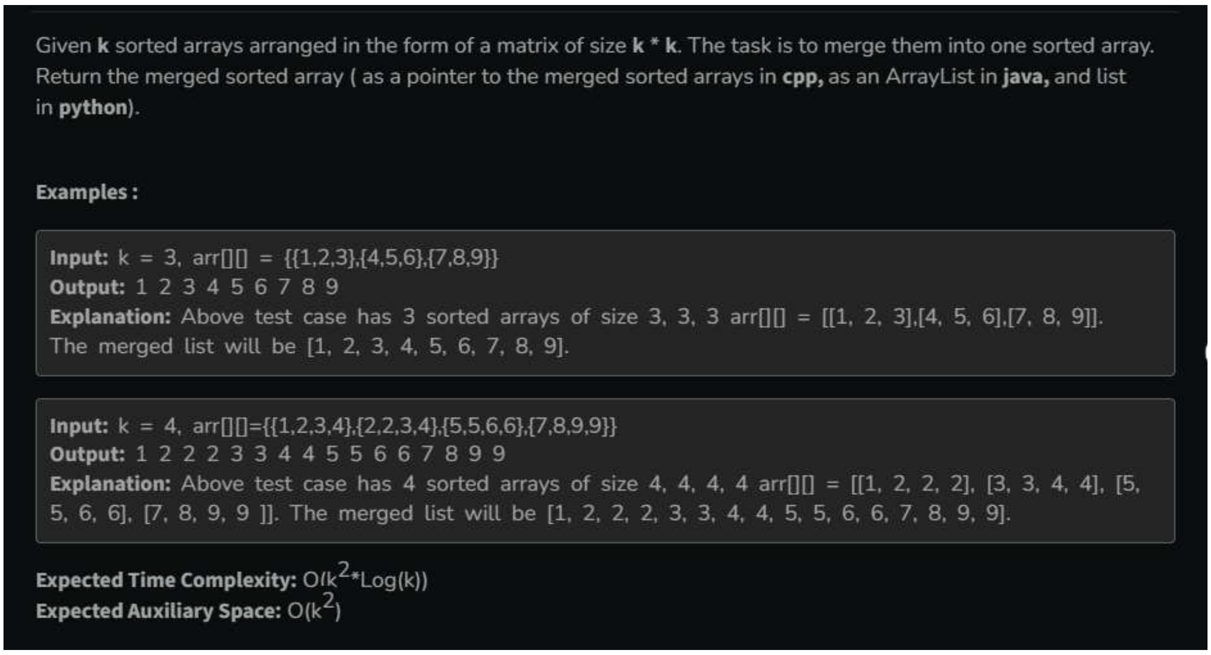
```
Input: arr[] = [7, 10, 4, 3, 20, 15], k = 3
Output: 7
Explanation: 3rd smallest element in the given array is 7.
```

```
Input: arr[] = [2, 3, 1, 20, 15], k = 4
Output: 15
Explanation: 4th smallest element in the given array is 15.
```



```java
class Solution {

    public static int kthSmallest(int[] arr, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int num : arr) {
            minHeap.add(num);
        }
        int result = 0;
        for (int i = 0; i < k; i++) {
            result = minHeap.poll();
        }
        return result;
    }
}
```

5.Merge K Sortedd array



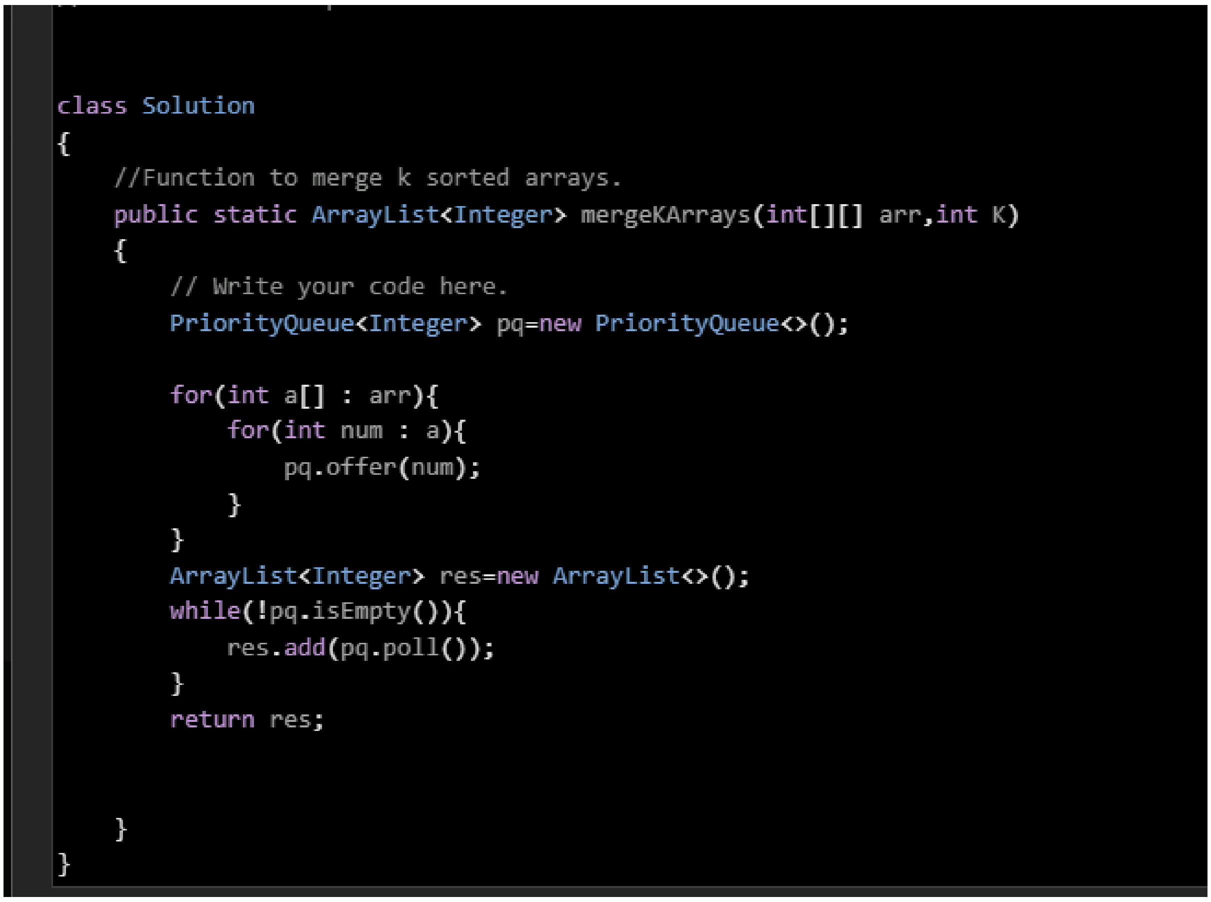
Given **k** sorted arrays arranged in the form of a matrix of size **k * k**. The task is to merge them into one sorted array. Return the merged sorted array (as a pointer to the merged sorted arrays in **cpp,** as an ArrayList in **java,** and list in **python**).

Examples :

Input: k = 3, arr[][] = {{1,2,3},{4,5,6},{7,8,9}}
Output: 1 2 3 4 5 6 7 8 9
Explanation: Above test case has 3 sorted arrays of size 3, 3, 3 arr[][] = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]. The merged list will be [1, 2, 3, 4, 5, 6, 7, 8, 9].

Input: k = 4, arr[][]={{1,2,3,4},{2,2,3,4},{5,5,6,6},{7,8,9,9}}
Output: 1 2 2 2 3 3 4 4 5 5 6 6 7 8 9 9
Explanation: Above test case has 4 sorted arrays of size 4, 4, 4, 4 arr[][] = [[1, 2, 2, 2], [3, 3, 4, 4], [5, 5, 6, 6], [7, 8, 9, 9]]. The merged list will be [1, 2, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 8, 9, 9].

Expected Time Complexity: O(k^2*Log(k))
Expected Auxiliary Space: O(k^2)

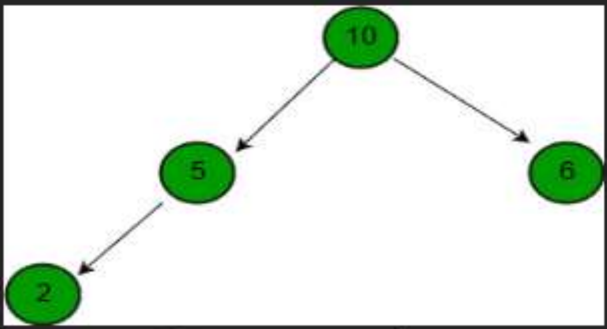


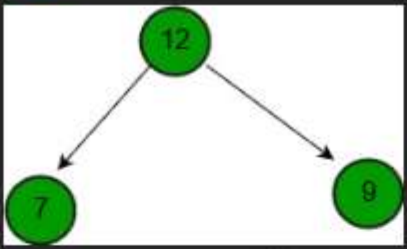
```java
class Solution
{
    //Function to merge k sorted arrays.
    public static ArrayList<Integer> mergeKArrays(int[][] arr,int K)
    {
        // Write your code here.
        PriorityQueue<Integer> pq=new PriorityQueue<>();

        for(int a[] : arr){
            for(int num : a){
                pq.offer(num);
            }
        }
        ArrayList<Integer> res=new ArrayList<>();
        while(!pq.isEmpty()){
            res.add(pq.poll());
        }
        return res;


    }
}
```

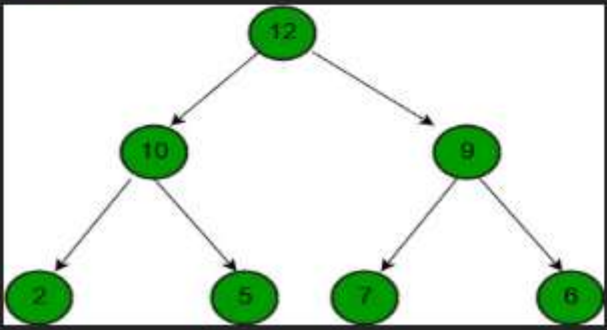
6.Merge Two Binary max Heaps

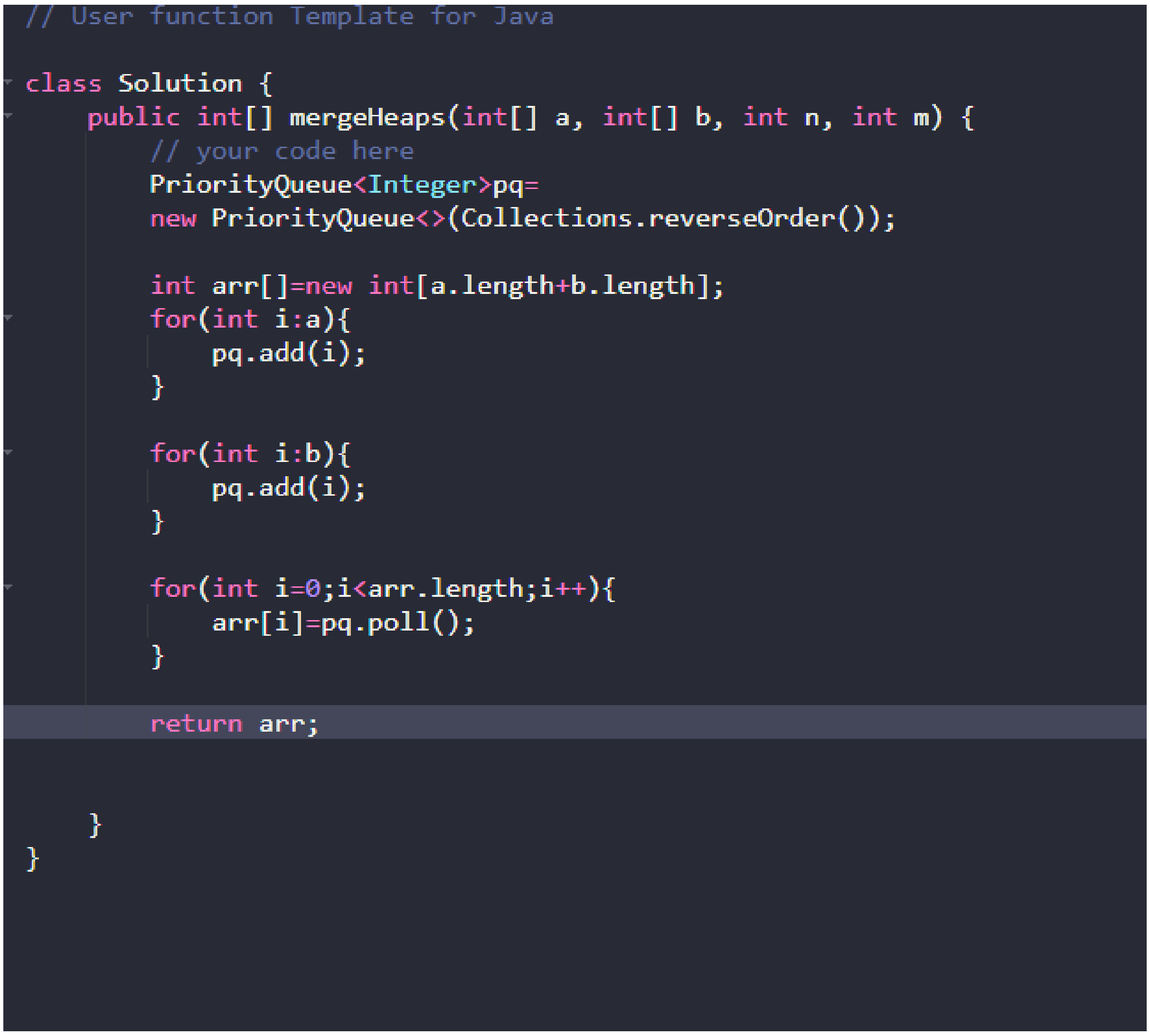
Given two binary max heaps as arrays, merge the given heaps to form a new max heap.

```
Input  :
n = 4 m = 3
a[] = {10, 5, 6, 2},
b[] = {12, 7, 9}
Output :
{12, 10, 9, 2, 5, 7, 6}
Explanation :
```



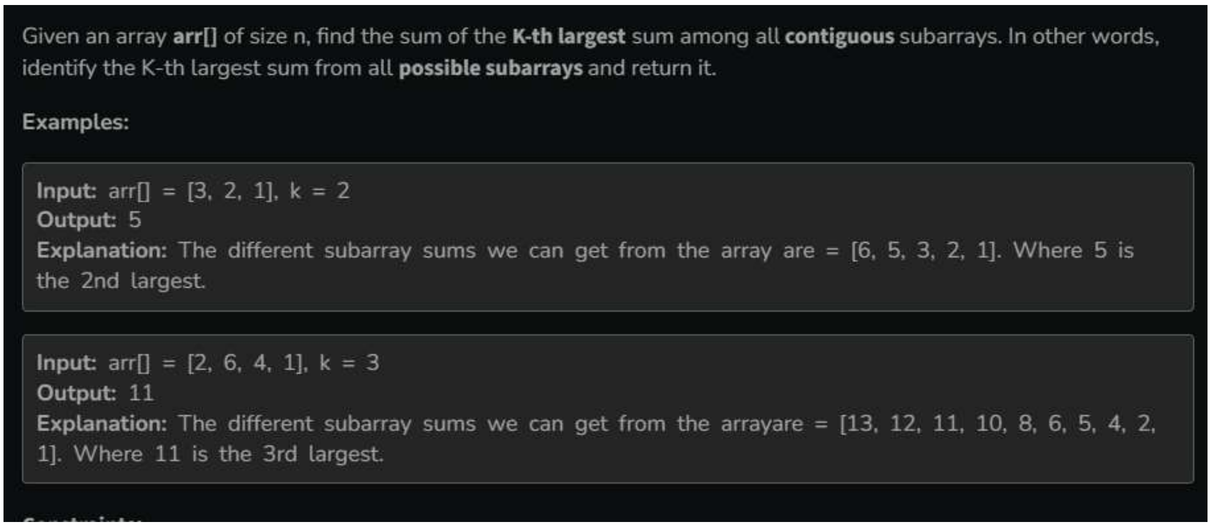






```java
// User function Template for Java

class Solution {
    public int[] mergeHeaps(int[] a, int[] b, int n, int m) {
        // your code here
        PriorityQueue<Integer>pq=
        new PriorityQueue<>(Collections.reverseOrder());

        int arr[]=new int[a.length+b.length];
        for(int i:a){
            pq.add(i);
        }

        for(int i:b){
            pq.add(i);
        }

        for(int i=0;i<arr.length;i++){
            arr[i]=pq.poll();
        }

        return arr;


    }
}
```

7.K-th Largest Sum Contiguous Subarray

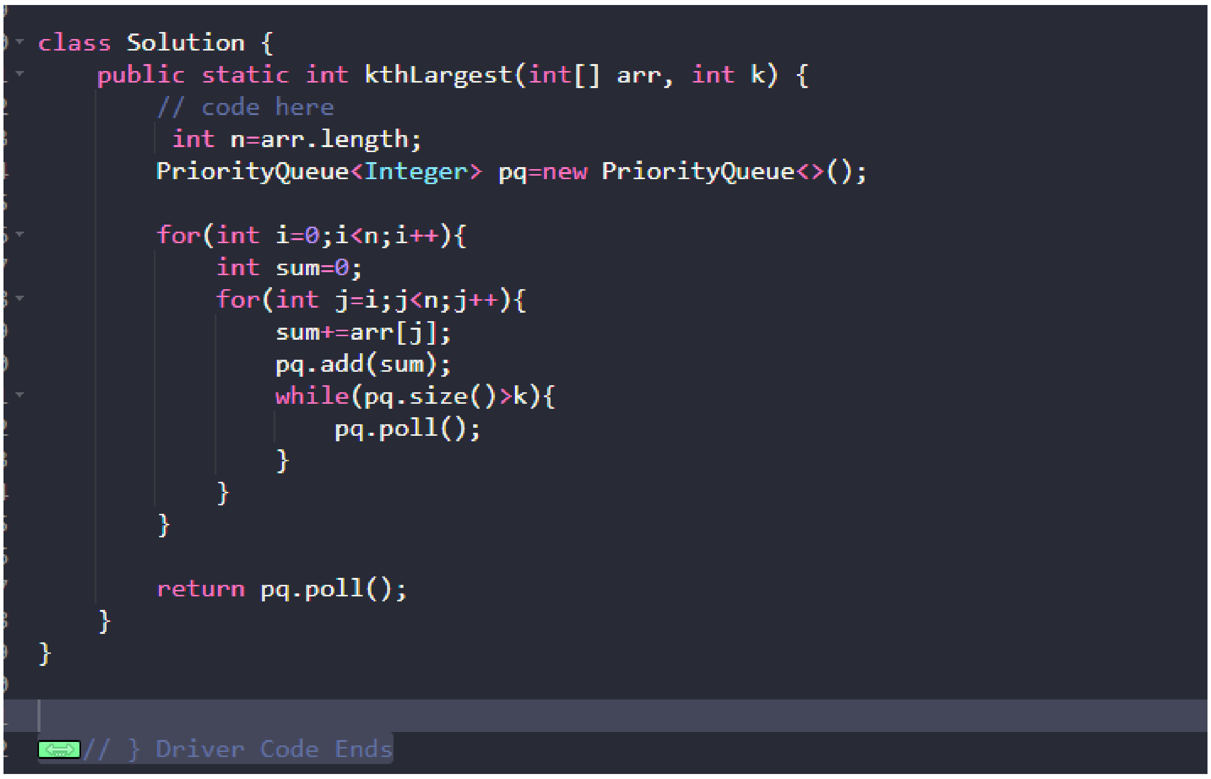


Given an array **arr[]** of size n, find the sum of the **K-th largest** sum among all **contiguous** subarrays. In other words, identify the K-th largest sum from all **possible subarrays** and return it.

Examples:

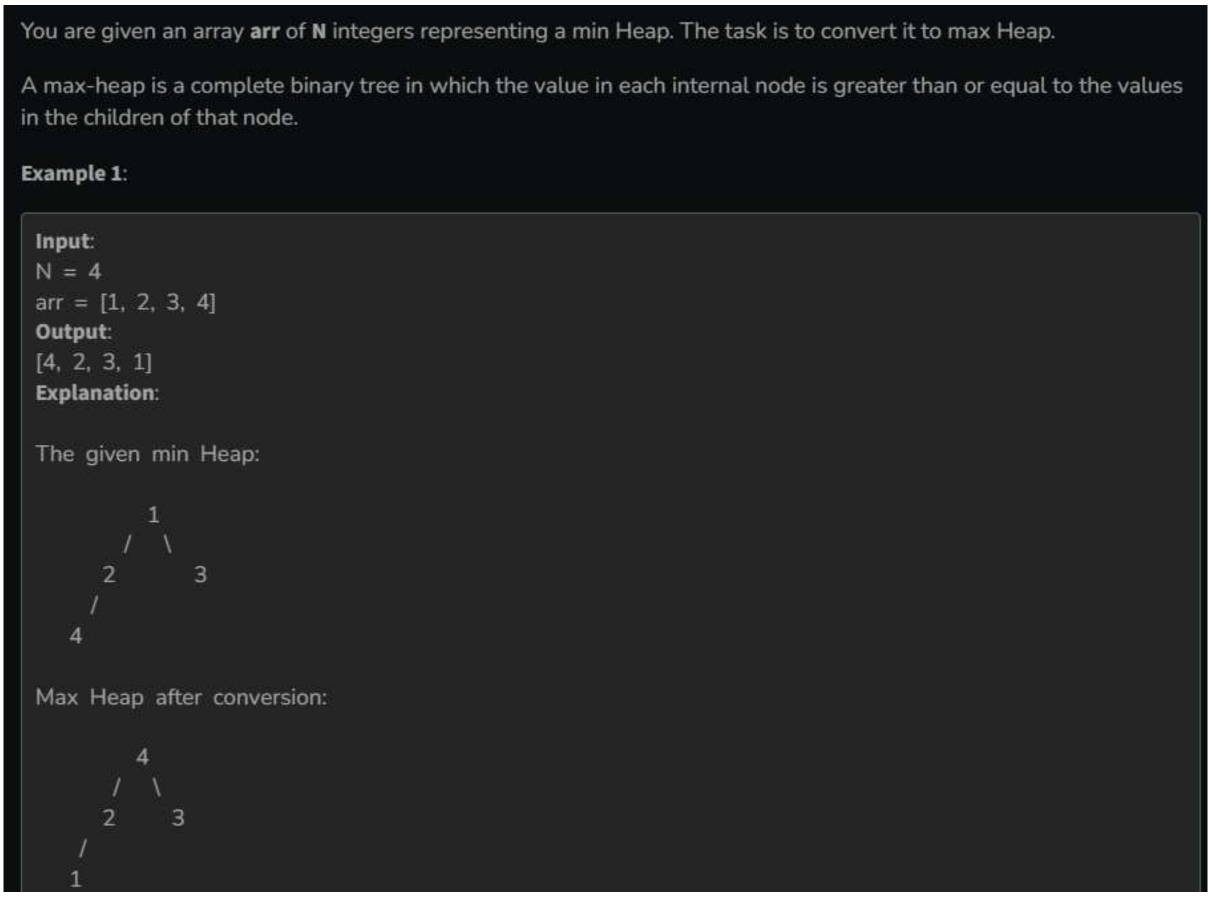
Input: arr[] = [3, 2, 1], k = 2
Output: 5
Explanation: The different subarray sums we can get from the array are = [6, 5, 3, 2, 1]. Where 5 is the 2nd largest.

Input: arr[] = [2, 6, 4, 1], k = 3
Output: 11
Explanation: The different subarray sums we can get from the arrayare = [13, 12, 11, 10, 8, 6, 5, 4, 2, 1]. Where 11 is the 3rd largest.



```
class Solution {
    public static int kthLargest(int[] arr, int k) {
        // code here
         int n=arr.length;
        PriorityQueue<Integer> pq=new PriorityQueue<>();

        for(int i=0;i<n;i++){
            int sum=0;
            for(int j=i;j<n;j++){
                sum+=arr[j];
                pq.add(sum);
                while(pq.size()>k){
                    pq.poll();
                }
            }
        }

        return pq.poll();
    }
}

// } Driver Code Ends
```

8.Convert Min Heap to Max Heap

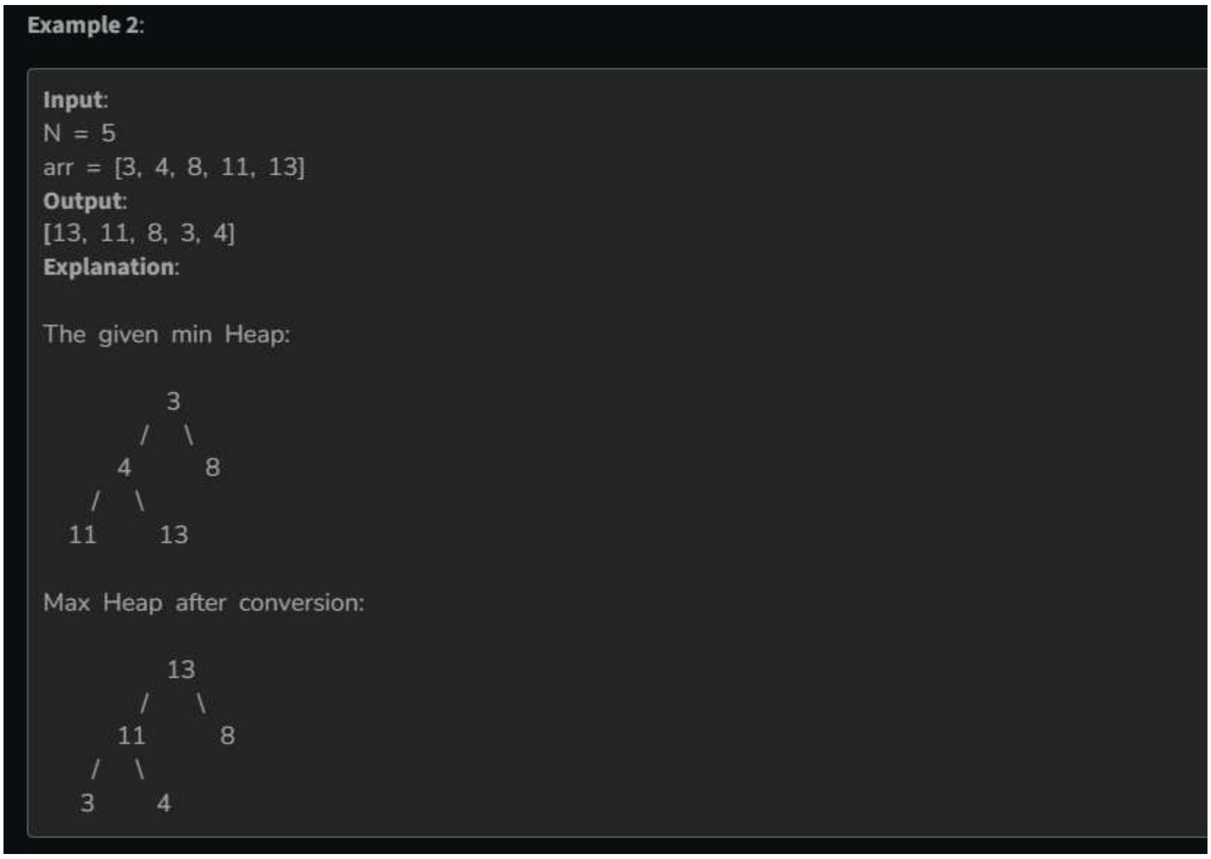


You are given an array **arr** of **N** integers representing a min Heap. The task is to convert it to max Heap.

A max-heap is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node.

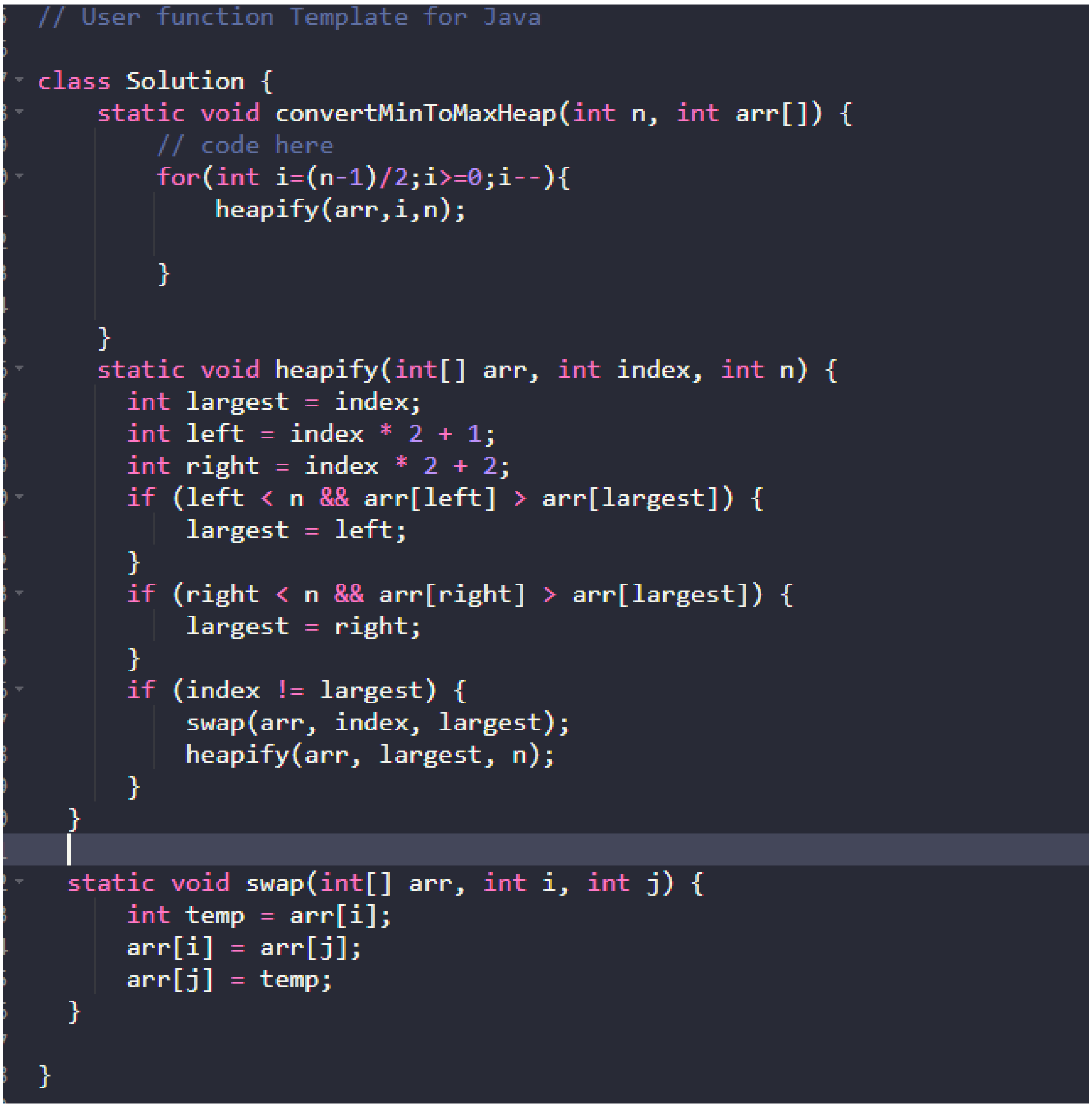
Example 1:

Input:
```
N = 4
arr = [1, 2, 3, 4]
```
Output:
```
[4, 2, 3, 1]
```
Explanation:
```
The given min Heap:

          1
        /   \
      2       3
     /
   4

Max Heap after conversion:

         4
       /   \
      2     3
     /
   1
```



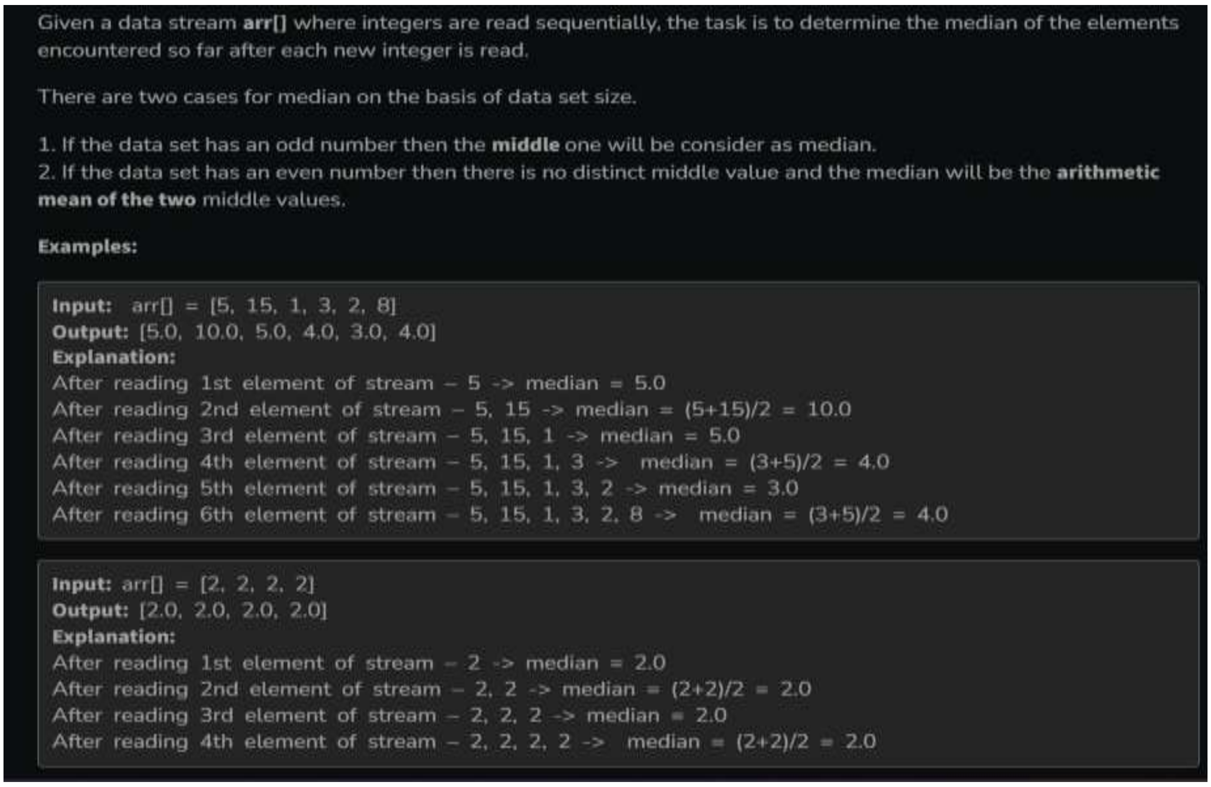
Example 2:

Input:
```
N = 5
arr = [3, 4, 8, 11, 13]
```
Output:
```
[13, 11, 8, 3, 4]
```
Explanation:
```
The given min Heap:

          3
        /   \
      4       8
    /   \
  11      13

Max Heap after conversion:

          13
        /    \
      11       8
    /   \
   3      4
```



```java
// User function Template for Java

class Solution {
    static void convertMinToMaxHeap(int n, int arr[]) {
        // code here
        for(int i=(n-1)/2;i>=0;i--){
            heapify(arr,i,n);

        }

    }
    static void heapify(int[] arr, int index, int n) {
      int largest = index;
      int left = index * 2 + 1;
      int right = index * 2 + 2;
      if (left < n && arr[left] > arr[largest]) {
        largest = left;
      }
      if (right < n && arr[right] > arr[largest]) {
        largest = right;
      }
      if (index != largest) {
        swap(arr, index, largest);
        heapify(arr, largest, n);
      }
  }

  static void swap(int[] arr, int i, int j) {
      int temp = arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
  }

}
```

9.Find Median in a stream



Given a data stream **arr[]** where integers are read sequentially, the task is to determine the median of the elements encountered so far after each new integer is read.

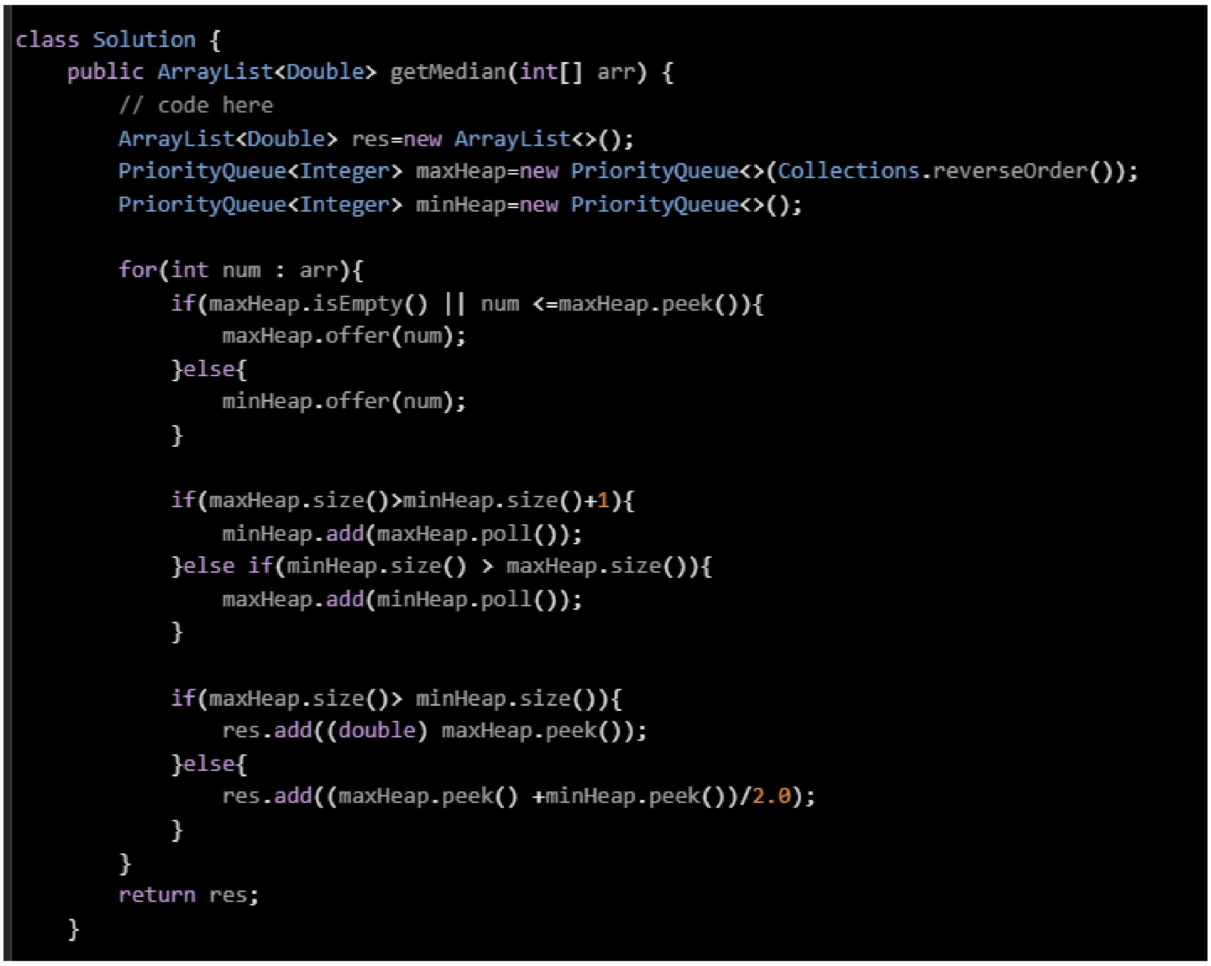
There are two cases for median on the basis of data set size.

1. If the data set has an odd number then the **middle** one will be consider as median.
2. If the data set has an even number then there is no distinct middle value and the median will be the **arithmetic mean of the two** middle values.

Examples:

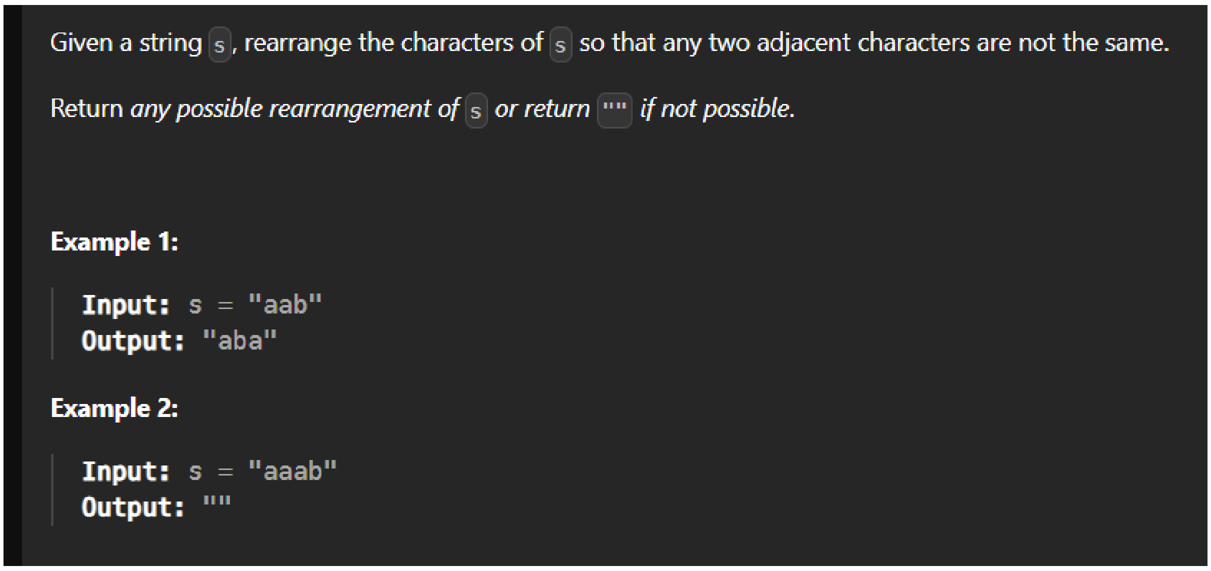
```
Input:  arr[] = [5, 15, 1, 3, 2, 8]
Output: [5.0, 10.0, 5.0, 4.0, 3.0, 4.0]
Explanation:
After reading 1st element of stream – 5 -> median = 5.0
After reading 2nd element of stream – 5, 15 -> median = (5+15)/2 = 10.0
After reading 3rd element of stream – 5, 15, 1 -> median = 5.0
After reading 4th element of stream – 5, 15, 1, 3 ->  median = (3+5)/2 = 4.0
After reading 5th element of stream – 5, 15, 1, 3, 2 -> median = 3.0
After reading 6th element of stream – 5, 15, 1, 3, 2, 8 ->  median = (3+5)/2 = 4.0
```

```
Input: arr[] = [2, 2, 2, 2]
Output: [2.0, 2.0, 2.0, 2.0]
Explanation:
After reading 1st element of stream – 2 -> median = 2.0
After reading 2nd element of stream – 2, 2 -> median = (2+2)/2 = 2.0
After reading 3rd element of stream – 2, 2, 2 -> median = 2.0
After reading 4th element of stream – 2, 2, 2, 2 ->  median = (2+2)/2 = 2.0
```



```java
class Solution {
    public ArrayList<Double> getMedian(int[] arr) {
        // code here
        ArrayList<Double> res=new ArrayList<>();
        PriorityQueue<Integer> maxHeap=new PriorityQueue<>(Collections.reverseOrder());
        PriorityQueue<Integer> minHeap=new PriorityQueue<>();

        for(int num : arr){
            if(maxHeap.isEmpty() || num <=maxHeap.peek()){
                maxHeap.offer(num);
            }else{
                minHeap.offer(num);
            }

            if(maxHeap.size()>minHeap.size()+1){
                minHeap.add(maxHeap.poll());
            }else if(minHeap.size() > maxHeap.size()){
                maxHeap.add(minHeap.poll());
            }

            if(maxHeap.size()> minHeap.size()){
                res.add((double) maxHeap.peek());
            }else{
                res.add((maxHeap.peek() +minHeap.peek())/2.0);
            }
        }
        return res;
    }
```

10.Re-Organise String


Given a string `s`, rearrange the characters of `s` so that any two adjacent characters are not the same.

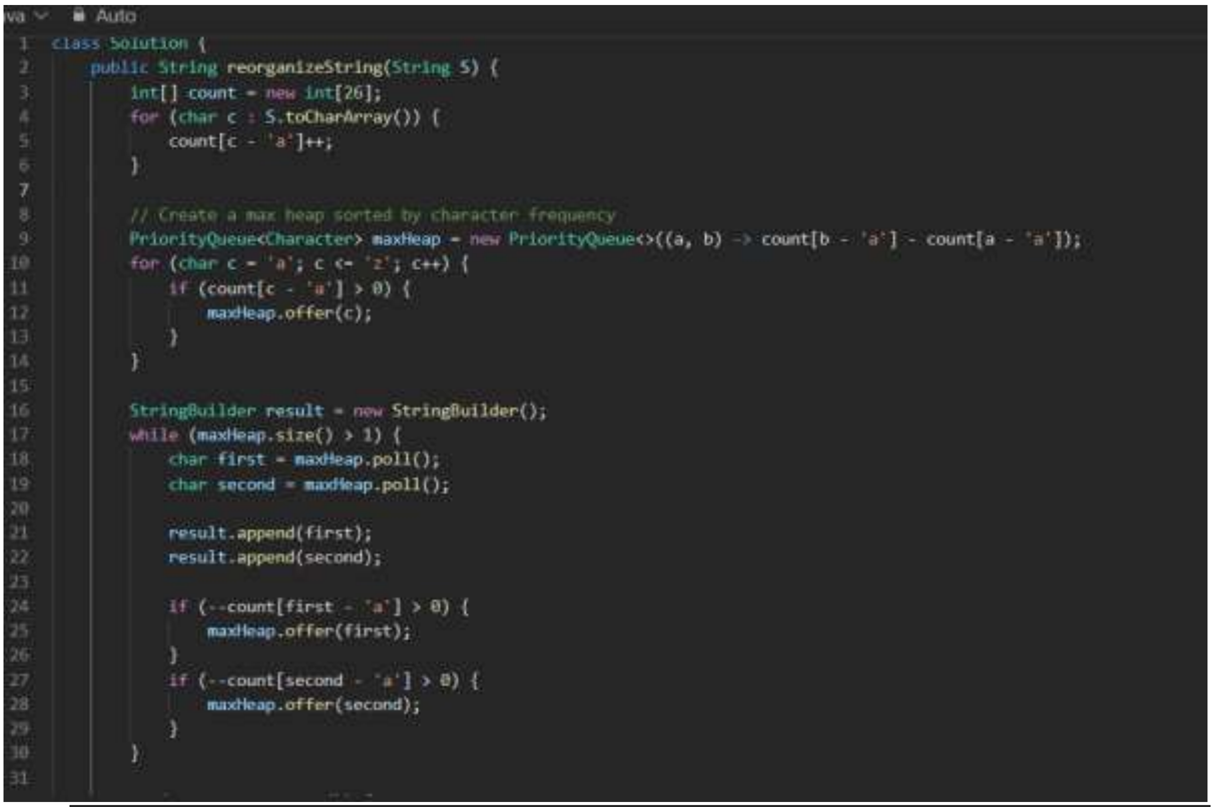
Return *any possible rearrangement of* `s` *or return* `""` *if not possible.*

Example 1:

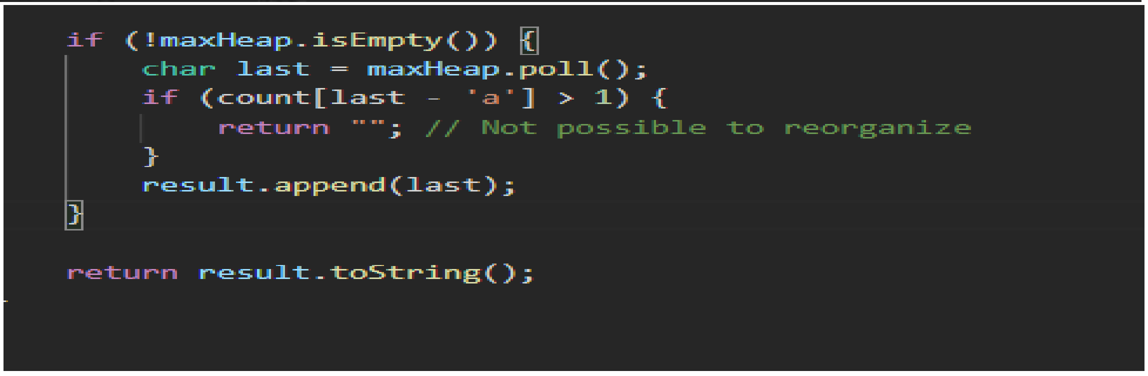
```
Input: s = "aab"
Output: "aba"
```

Example 2:

```
Input: s = "aaab"
Output: ""
```

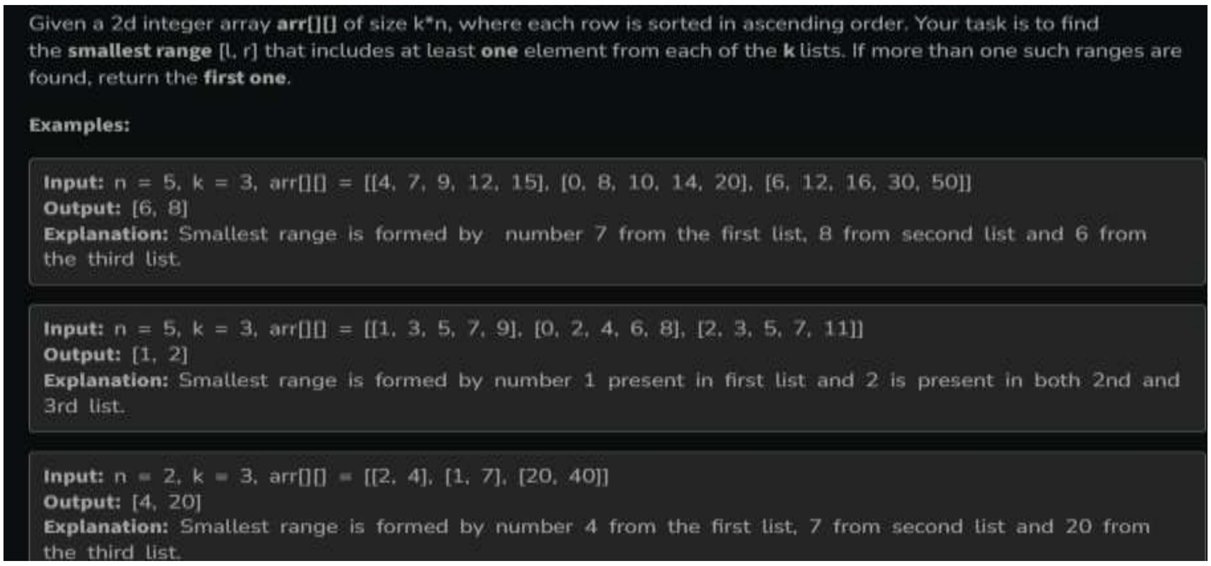


```java
class Solution {
    public String reorganizeString(String S) {
        int[] count = new int[26];
        for (char c : S.toCharArray()) {
            count[c - 'a']++;
        }

        // Create a max heap sorted by character frequency
        PriorityQueue<Character> maxHeap = new PriorityQueue<>((a, b) -> count[b - 'a'] - count[a - 'a']);
        for (char c = 'a'; c <= 'z'; c++) {
            if (count[c - 'a'] > 0) {
                maxHeap.offer(c);
            }
        }

        StringBuilder result = new StringBuilder();
        while (maxHeap.size() > 1) {
            char first = maxHeap.poll();
            char second = maxHeap.poll();

            result.append(first);
            result.append(second);

            if (--count[first - 'a'] > 0) {
                maxHeap.offer(first);
            }
            if (--count[second - 'a'] > 0) {
                maxHeap.offer(second);
            }
        }
```



```java
        if (!maxHeap.isEmpty()) {
            char last = maxHeap.poll();
            if (count[last - 'a'] > 1) {
                return ""; // Not possible to reorganize
            }
            result.append(last);
        }

        return result.toString();
```

11.Smallest range in k lists



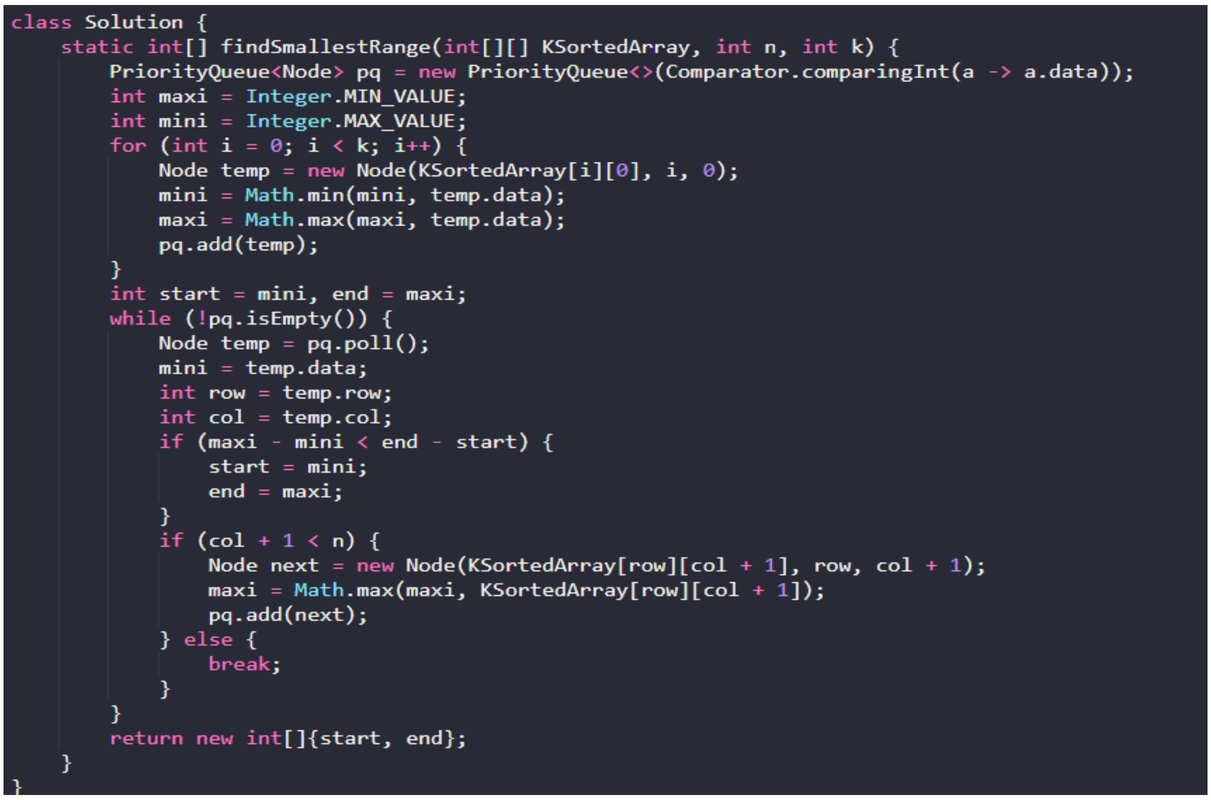
Given a 2d integer array **arr[][]** of size k*n, where each row is sorted in ascending order. Your task is to find the **smallest range** [l, r] that includes at least **one** element from each of the **k** lists. If more than one such ranges are found, return the **first one**.

Examples:

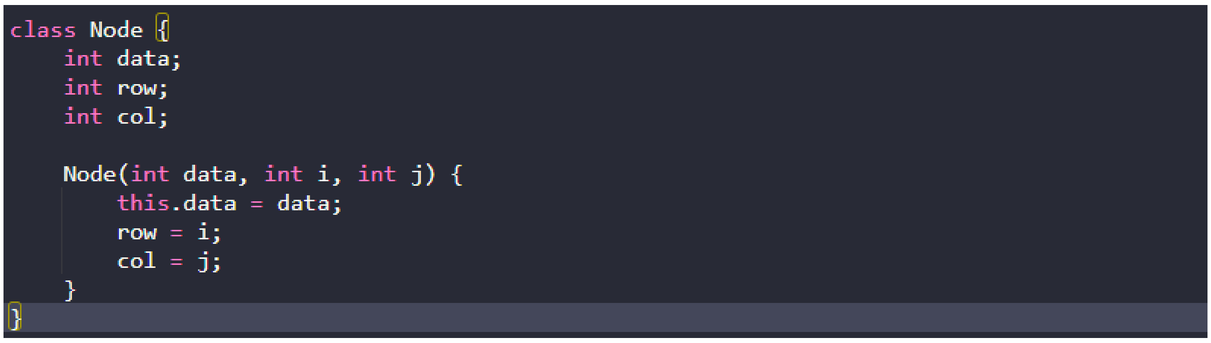
Input: n = 5, k = 3, arr[][] = [[4, 7, 9, 12, 15], [0, 8, 10, 14, 20], [6, 12, 16, 30, 50]]
Output: [6, 8]
Explanation: Smallest range is formed by number 7 from the first list, 8 from second list and 6 from the third list.

Input: n = 5, k = 3, arr[][] = [[1, 3, 5, 7, 9], [0, 2, 4, 6, 8], [2, 3, 5, 7, 11]]
Output: [1, 2]
Explanation: Smallest range is formed by number 1 present in first list and 2 is present in both 2nd and 3rd list.

Input: n = 2, k = 3, arr[][] = [[2, 4], [1, 7], [20, 40]]
Output: [4, 20]
Explanation: Smallest range is formed by number 4 from the first list, 7 from second list and 20 from the third list.

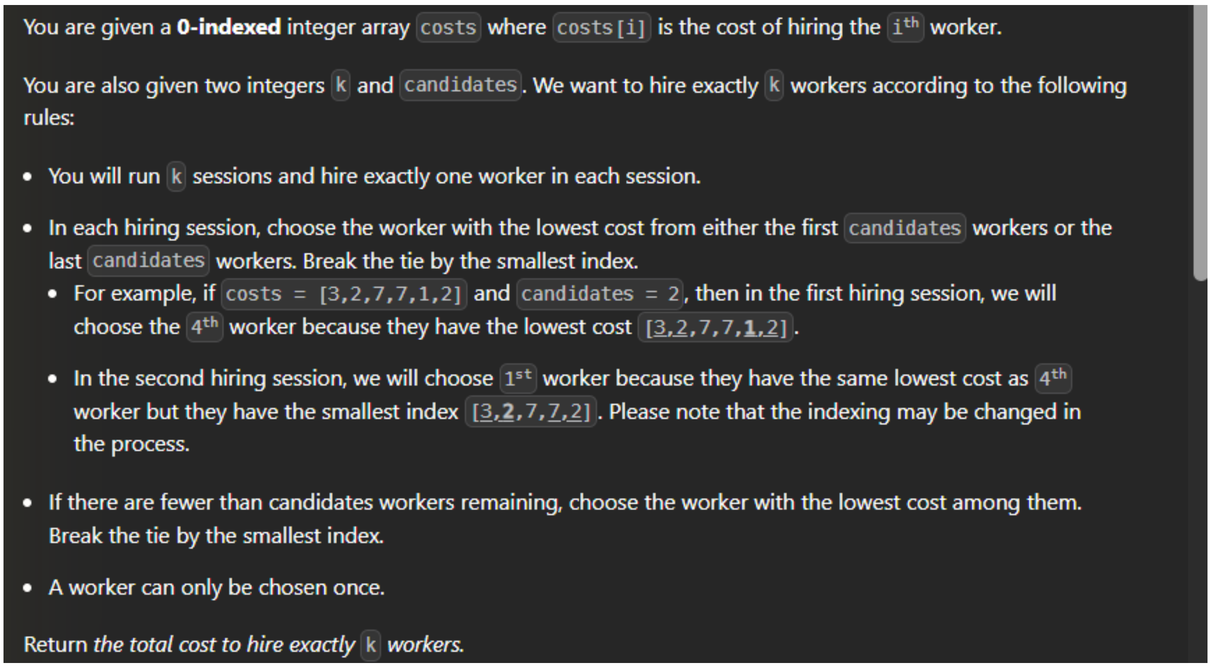


```java
class Solution {
    static int[] findSmallestRange(int[][] KSortedArray, int n, int k) {
        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a.data));
        int maxi = Integer.MIN_VALUE;
        int mini = Integer.MAX_VALUE;
        for (int i = 0; i < k; i++) {
            Node temp = new Node(KSortedArray[i][0], i, 0);
            mini = Math.min(mini, temp.data);
            maxi = Math.max(maxi, temp.data);
            pq.add(temp);
        }
        int start = mini, end = maxi;
        while (!pq.isEmpty()) {
            Node temp = pq.poll();
            mini = temp.data;
            int row = temp.row;
            int col = temp.col;
            if (maxi - mini < end - start) {
                start = mini;
                end = maxi;
            }
            if (col + 1 < n) {
                Node next = new Node(KSortedArray[row][col + 1], row, col + 1);
                maxi = Math.max(maxi, KSortedArray[row][col + 1]);
                pq.add(next);
            } else {
                break;
            }
        }
        return new int[]{start, end};
    }
}
```



```java
class Node {
    int data;
    int row;
    int col;

    Node(int data, int i, int j) {
        this.data = data;
        row = i;
        col = j;
    }
}
```

12.Total Cost to hire K Worker

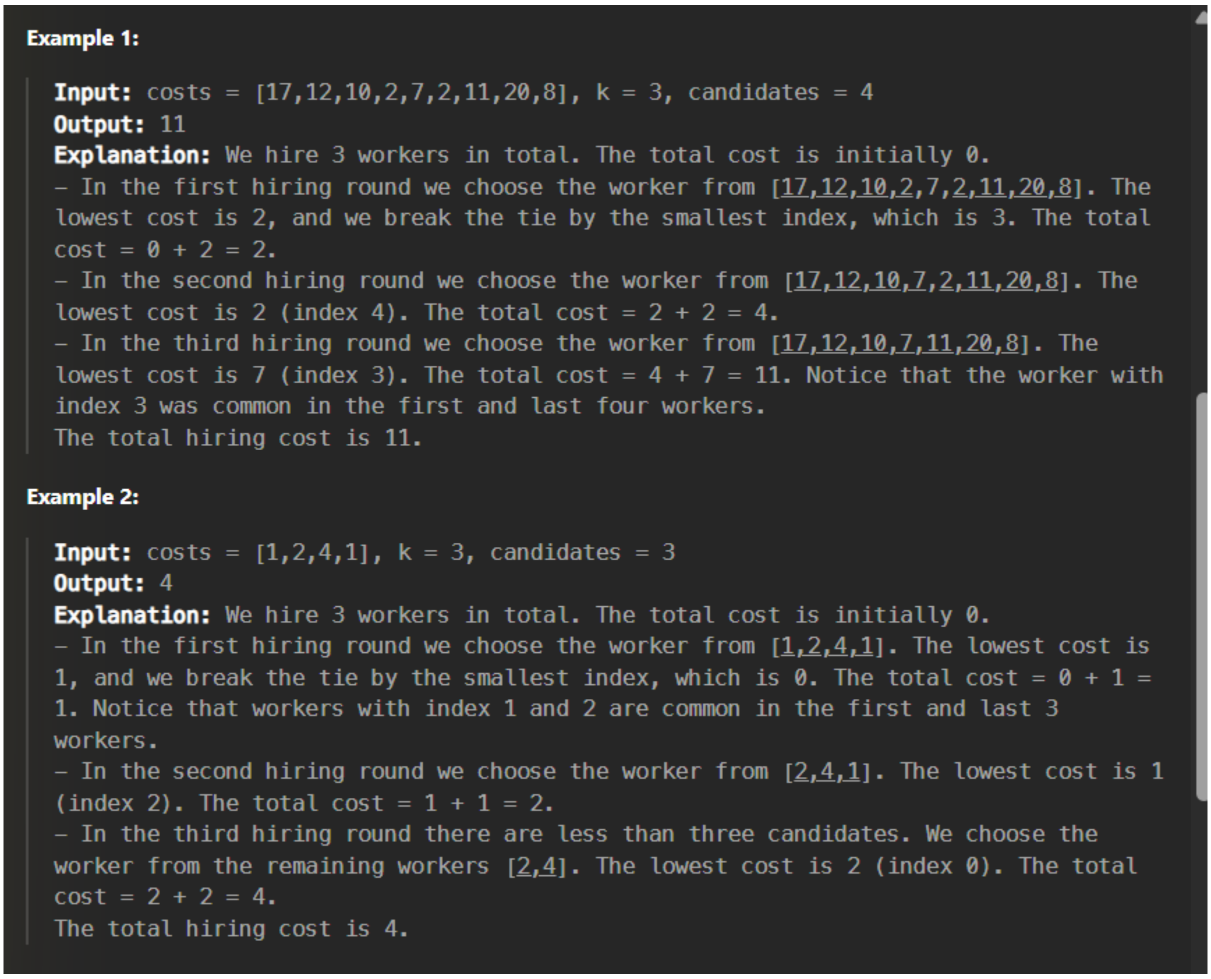


You are given a **0-indexed** integer array `costs` where `costs[i]` is the cost of hiring the `i`th worker.

You are also given two integers `k` and `candidates`. We want to hire exactly `k` workers according to the following rules:

- You will run `k` sessions and hire exactly one worker in each session.
- In each hiring session, choose the worker with the lowest cost from either the first `candidates` workers or the last `candidates` workers. Break the tie by the smallest index.
  - For example, if `costs = [3,2,7,7,1,2]` and `candidates = 2`, then in the first hiring session, we will choose the `4`th worker because they have the lowest cost `[3,2,7,7,1,2]`.
  - In the second hiring session, we will choose `1`st worker because they have the same lowest cost as `4`th worker but they have the smallest index `[3,2,7,7,2]`. Please note that the indexing may be changed in the process.
- If there are fewer than candidates workers remaining, choose the worker with the lowest cost among them. Break the tie by the smallest index.
- A worker can only be chosen once.

Return *the total cost to hire exactly* `k` *workers.*

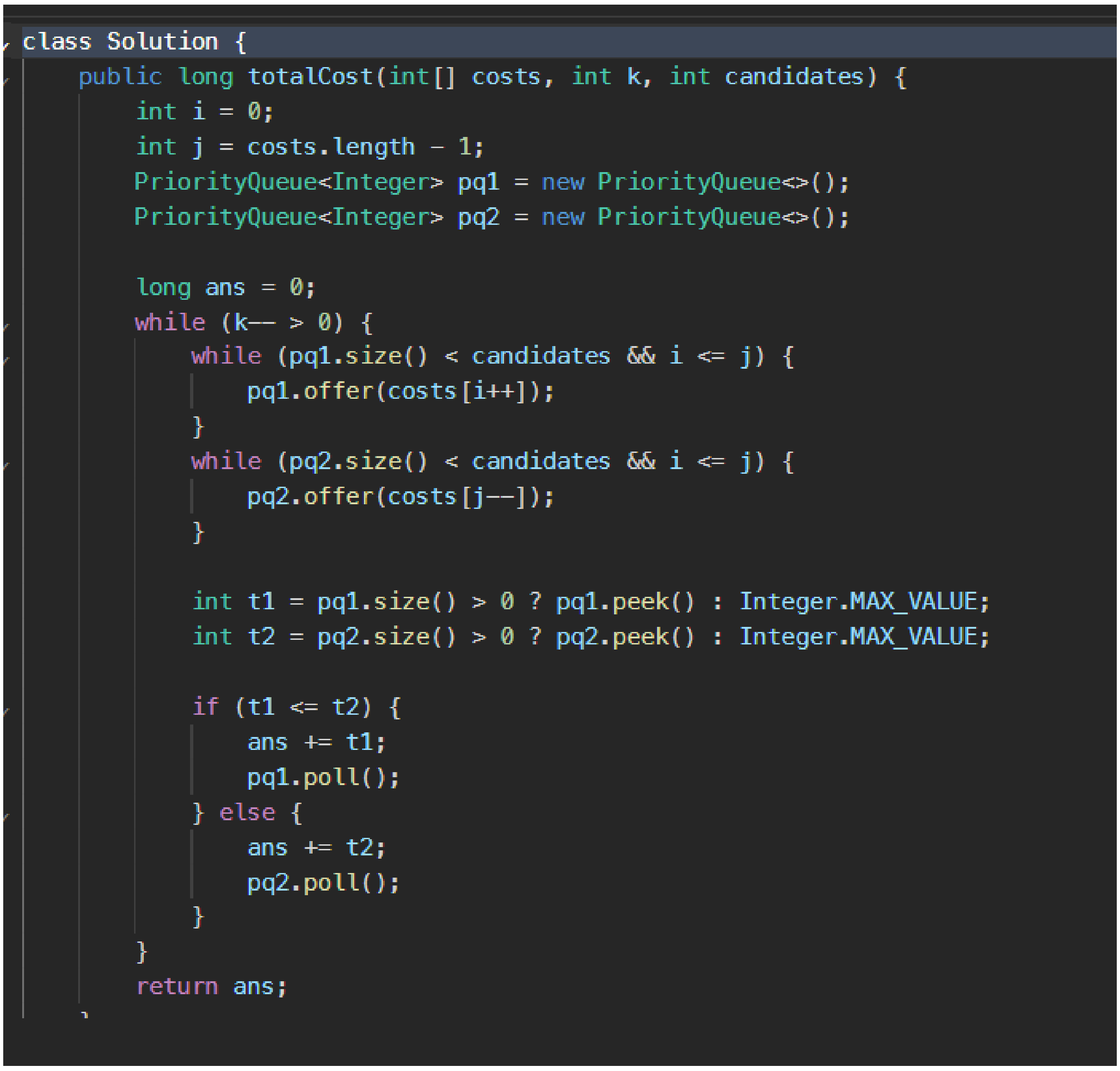


Example 1:

```
Input: costs = [17,12,10,2,7,2,11,20,8], k = 3, candidates = 4
Output: 11
Explanation: We hire 3 workers in total. The total cost is initially 0.
- In the first hiring round we choose the worker from [17,12,10,2,7,2,11,20,8]. The
lowest cost is 2, and we break the tie by the smallest index, which is 3. The total
cost = 0 + 2 = 2.
- In the second hiring round we choose the worker from [17,12,10,7,2,11,20,8]. The
lowest cost is 2 (index 4). The total cost = 2 + 2 = 4.
- In the third hiring round we choose the worker from [17,12,10,7,11,20,8]. The
lowest cost is 7 (index 3). The total cost = 4 + 7 = 11. Notice that the worker with
index 3 was common in the first and last four workers.
The total hiring cost is 11.
```

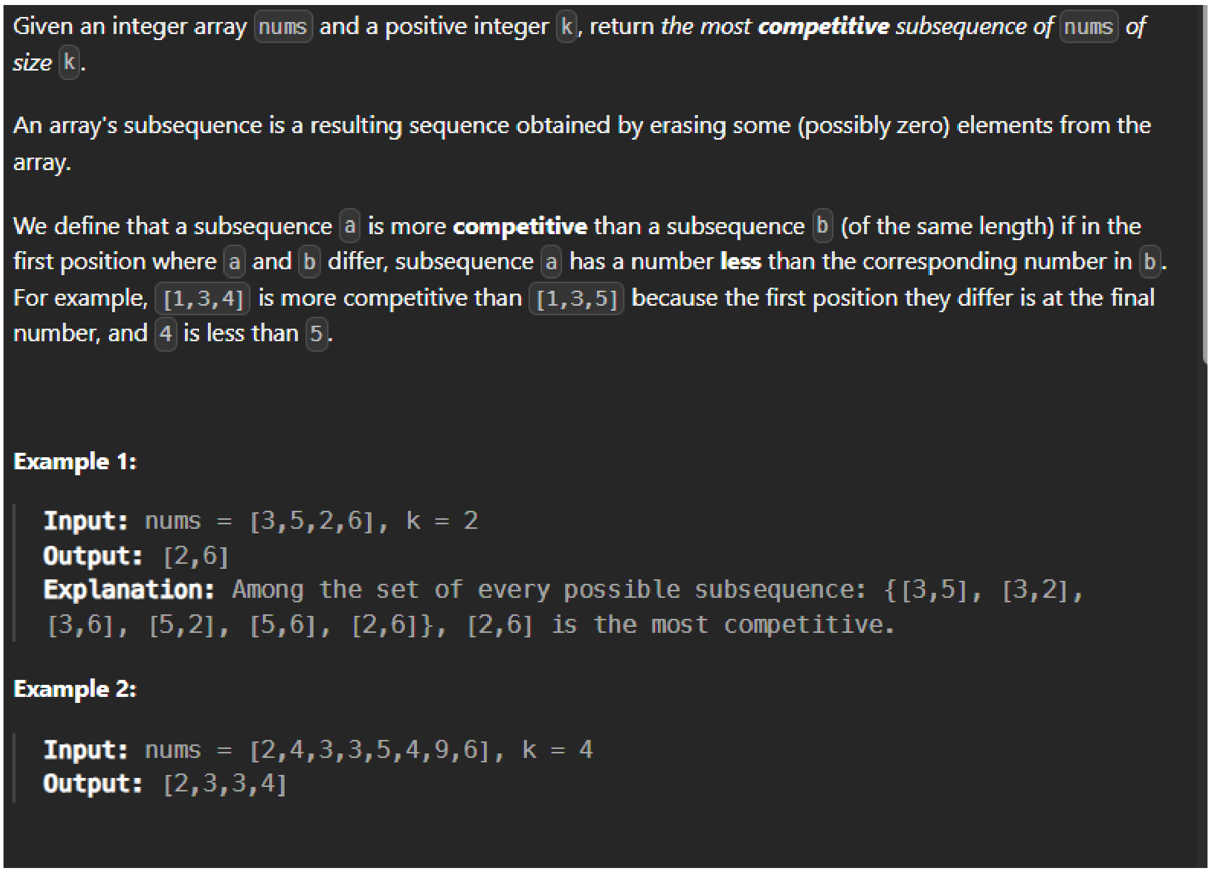
Example 2:

```
Input: costs = [1,2,4,1], k = 3, candidates = 3
Output: 4
Explanation: We hire 3 workers in total. The total cost is initially 0.
- In the first hiring round we choose the worker from [1,2,4,1]. The lowest cost is
1, and we break the tie by the smallest index, which is 0. The total cost = 0 + 1 =
1. Notice that workers with index 1 and 2 are common in the first and last 3
workers.
- In the second hiring round we choose the worker from [2,4,1]. The lowest cost is 1
(index 2). The total cost = 1 + 1 = 2.
- In the third hiring round there are less than three candidates. We choose the
worker from the remaining workers [2,4]. The lowest cost is 2 (index 0). The total
cost = 2 + 2 = 4.
The total hiring cost is 4.
```



```java
class Solution {
    public long totalCost(int[] costs, int k, int candidates) {
        int i = 0;
        int j = costs.length - 1;
        PriorityQueue<Integer> pq1 = new PriorityQueue<>();
        PriorityQueue<Integer> pq2 = new PriorityQueue<>();

        long ans = 0;
        while (k-- > 0) {
            while (pq1.size() < candidates && i <= j) {
                pq1.offer(costs[i++]);
            }
            while (pq2.size() < candidates && i <= j) {
                pq2.offer(costs[j--]);
            }

            int t1 = pq1.size() > 0 ? pq1.peek() : Integer.MAX_VALUE;
            int t2 = pq2.size() > 0 ? pq2.peek() : Integer.MAX_VALUE;

            if (t1 <= t2) {
                ans += t1;
                pq1.poll();
            } else {
                ans += t2;
                pq2.poll();
            }
        }
        return ans;
```

13.Find the most Competitive subsequence



Given an integer array `nums` and a positive integer `k`, return *the most* ***competitive*** *subsequence of* `nums` *of size* `k`.

An array's subsequence is a resulting sequence obtained by erasing some (possibly zero) elements from the array.

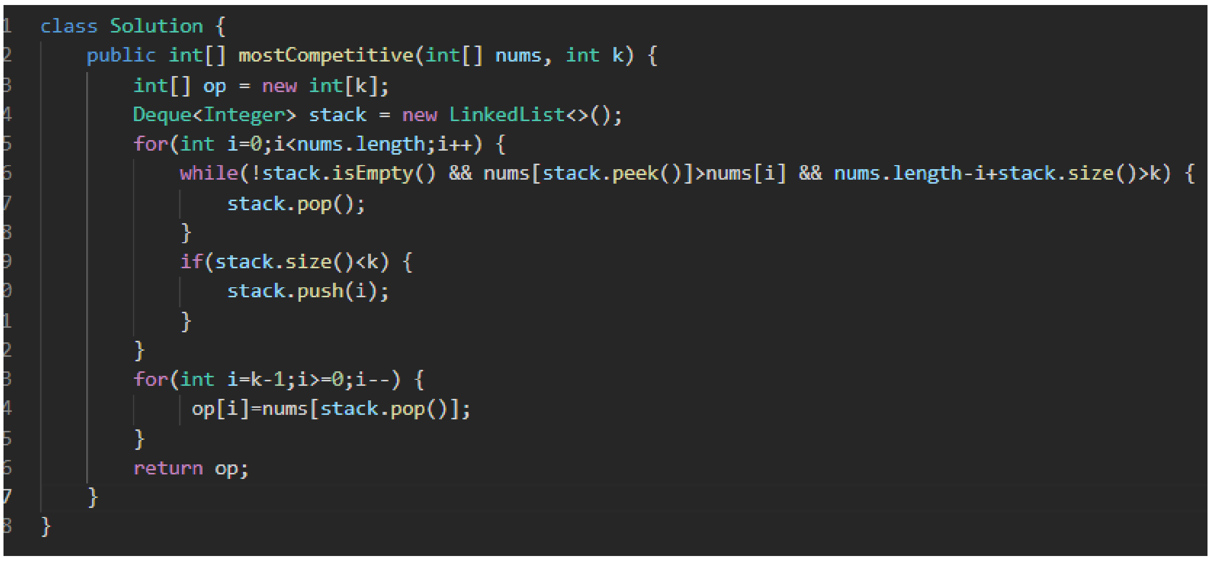
We define that a subsequence `a` is more **competitive** than a subsequence `b` (of the same length) if in the first position where `a` and `b` differ, subsequence `a` has a number **less** than the corresponding number in `b`. For example, `[1,3,4]` is more competitive than `[1,3,5]` because the first position they differ is at the final number, and `4` is less than `5`.

Example 1:

```
Input: nums = [3,5,2,6], k = 2
Output: [2,6]
Explanation: Among the set of every possible subsequence: {[3,5], [3,2],
[3,6], [5,2], [5,6], [2,6]}, [2,6] is the most competitive.
```

Example 2:

```
Input: nums = [2,4,3,3,5,4,9,6], k = 4
Output: [2,3,3,4]
```



```java
class Solution {
    public int[] mostCompetitive(int[] nums, int k) {
        int[] op = new int[k];
        Deque<Integer> stack = new LinkedList<>();
        for(int i=0;i<nums.length;i++) {
            while(!stack.isEmpty() && nums[stack.peek()]>nums[i] && nums.length-i+stack.size()>k) {
                stack.pop();
            }
            if(stack.size()<k) {
                stack.push(i);
            }
        }
        for(int i=k-1;i>=0;i--) {
             op[i]=nums[stack.pop()];
        }
        return op;
    }
}
```

Approach: https://leetcode.com/problems/find-the-most-competitive-subsequence/solutions/1027677/java-with-explaination-2-approaches-99-faster/