Heaps

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

**Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.

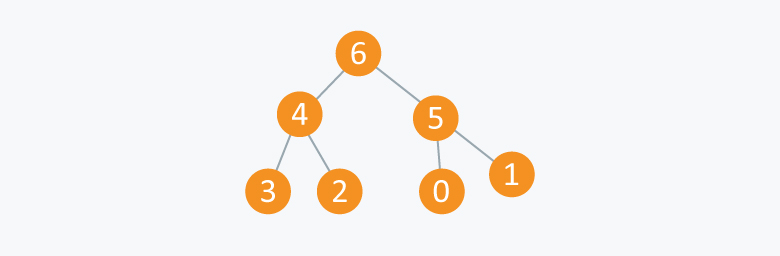
**Min-Heap:** In a Min-Heap the key present at the root node must be minimum among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.

A heap is a tree-based data structure in which all the nodes of the tree are in a specific order.

For example, if X is the parent node of Y , then the value of X follows a specific order with respect to the value of Y and the same order will be followed across the tree.

The maximum number of children of a node in a heap depends on the type of heap. However, in the more commonly used heap type, there are at most 2 children of a node and it's known as a Binary heap.

In binary heap, if the heap is a complete binary tree with N nodes, then it has smallest possible height which is logN.



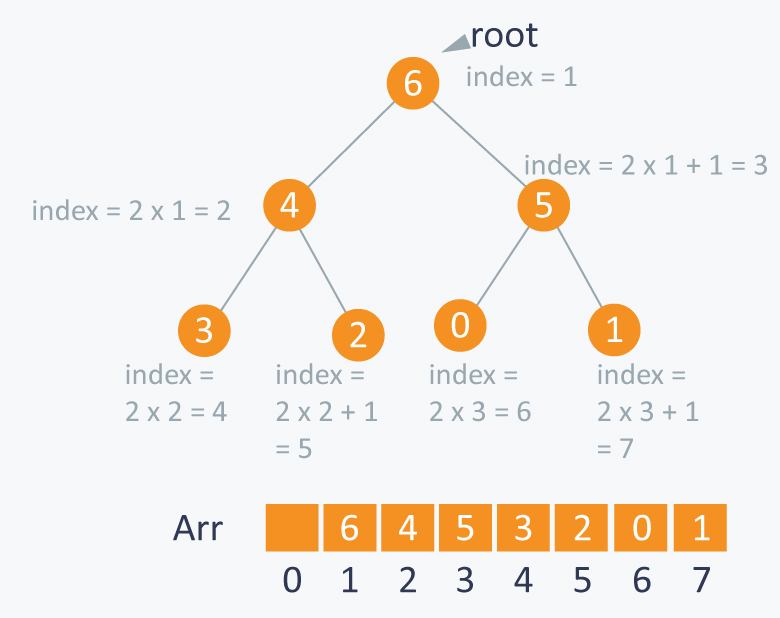
In the diagram above, you can observe a particular sequence, i.e each node has greater value than any of its children.

Suppose there are N Jobs in a queue to be done, and each job has its own priority. The job with maximum priority will get completed first than others. At each instant, we are completing a job with maximum priority and at the same time we are also interested in inserting a new job in the queue with its own priority.

So at each instant we have to check for the job with maximum priority to complete it and also insert if there is a new job. This task can be very easily executed using a heap by considering N jobs as N nodes of the tree.

As you can see in the diagram below, we can use an array to store the nodes of the tree. Let’s say we have 7 elements with values {6, 4, 5, 3, 2, 0, 1}.

Note: An array can be used to simulate a tree in the following way. If we are storing one element at index i in array Arr , then its parent will be stored at index i/2 (unless its a root, as root has no parent) and can be accessed by Arr[i/2], and its left child can be accessed by Arr[2\*i] and its right child can be accessed by Arr[2\*i+1]. Index of root will be 1 in an array.



There can be two types of heap:

Max Heap: In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.

Implementation:

Let’s assume that we have a heap having some elements which are stored in array Arr. The way to convert this array into a heap structure is the following. We pick a node in the array, check if the left sub-tree and the right sub-tree are max heaps, in themselves and the node itself is a max heap (it’s value should be greater than all the child nodes)

To do this we will implement a function that can maintain the property of max heap (i.e each element value should be greater than or equal to any of its child and smaller than or equal to its parent).

void max\_heapify (int Arr[ ], int i, int N)

{

        int left = 2\*i                   //left child

        int right = 2\*i +1           //right child

        if(left<= N and Arr[left] > Arr[i] )

              largest = left;

        else

             largest = i;

        if(right <= N and Arr[right] > Arr[largest] )

            largest = right;

        if(largest != i )

        {

            swap (Arr[i] , Arr[largest]);

            max\_heapify (Arr, largest,N);

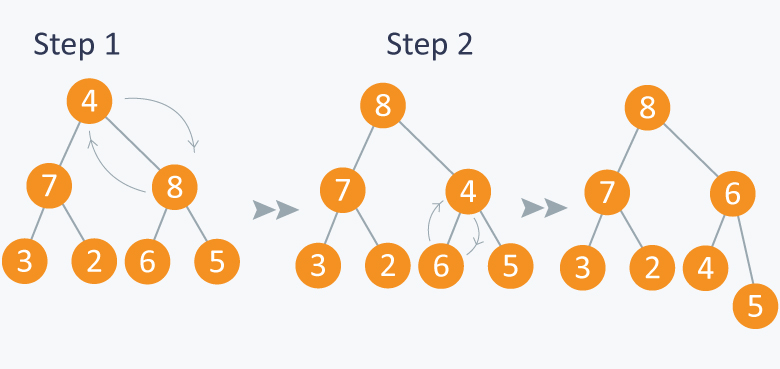
        }

}

Complexity: O(log N)

Example:

In the diagram below,initially 1st node (root node) is violating property of max-heap as it has smaller value than its children, so we are performing max\_heapify function on this node having value 4.



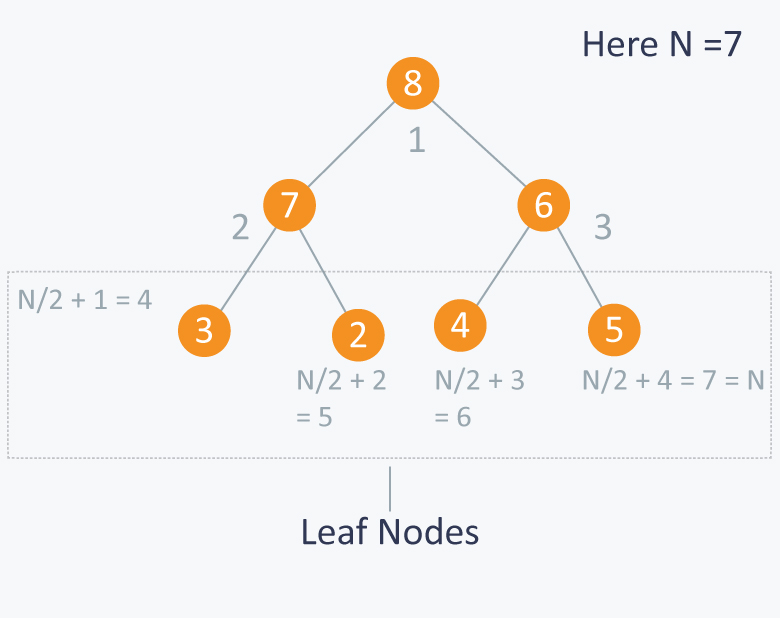
As 8 is greater than 4, so 8 is swapped with 4 and max\_heapify is performed again on 4, but on different position. Now in step 2, 6 is greater than 4, so 4 is swapped with 6 and we will get a max heap, as now 4 is a leaf node, so further call to max\_heapify will not create any effect on heap.

Now as we can see that we can maintain max- heap by using **max\_heapify** function.

Before moving ahead, lets observe a property which states: A N element heap stored in an array has leaves indexed by N/2+1, N/2+2 , N/2+3 …. upto N.

Let’s observe this with an example:

Lets take above example of 7 elements having values {8, 7, 6, 3, 2, 4, 5}.



So you can see that elements 3, 2, 4, 5 are indexed by N/2+1 (i.e 4), N/2+2 (i.e 5 ) and N/2+3 (i.e 6) and N/2+4 (i.e 7) respectively.

**Building MAX HEAP:**

Now let’s say we have N elements stored in the array Arr indexed from 1 to N. They are currently not following the property of max heap. So we can use max-heapify function to make a max heap out of the array.

How?

From the above property we observed that elements from Arr[N/2+1] to Arr[N] are leaf nodes, and each node is a 1 element heap. We can use max\_heapify function in a bottom up manner on remaining nodes, so that we can cover each node of tree.

void build\_maxheap (int Arr[ ])

{

        for(int i = N/2 ; i >= 1 ; i-- )

        {

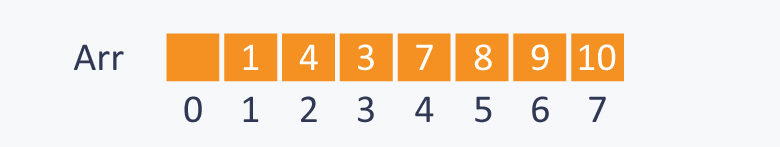
            max\_heapify (Arr, i) ;

        }

}

**Complexity:** O(N). **max\_heapify** function has complexity logN and the **build\_maxheap** functions runs only N/2 times, but the amortized complexity for this function is actually linear.  
For more details, you can refer [this](http://www.cs.umd.edu/~meesh/351/mount/lectures/lect14-heapsort-analysis-part.pdf).

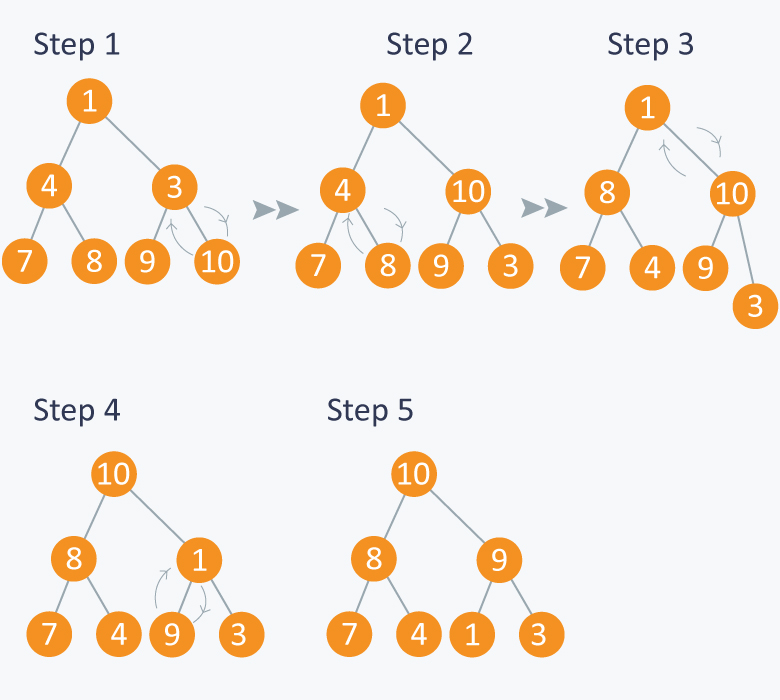
**Example:**  
Suppose you have 7 elements stored in array Arr.



Here N=7, so starting from node having index N/2=3, (also having value 3 in the above diagram), we will call max\_heapify from index N/2 to 1.

In the diagram below:

In step 1, in max\_heapify(Arr, 3), as 10 is greater than 3, 3 and 10 are swapped and further call to max\_heap(Arr, 7) will have no effect as 3 is a leaf node now.

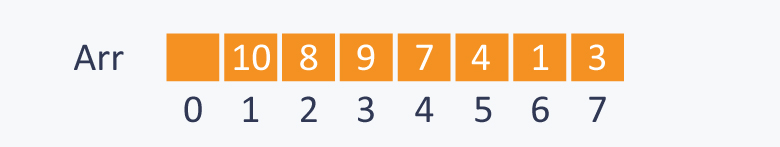


In step 2, calling max\_heapify(Arr, 2) , (node indexed with 2 has value 4) , 4 is swapped with 8 and further call to max\_heap(Arr, 5) will have no effect, as 4 is a leaf node now.

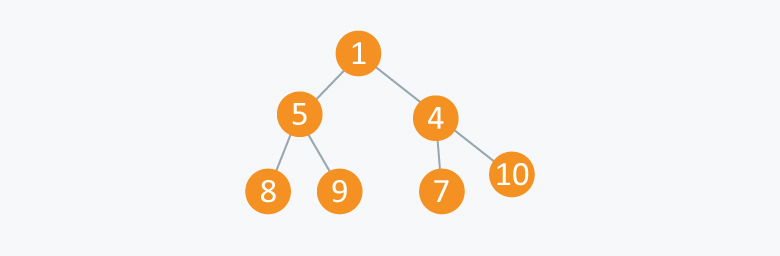
In step 3, calling max\_heapify(Arr, 1) , (node indexed with 1 has value 1 ), 1 is swapped with 10 .

Step 4 is a subpart of step 3, as after swapping 1 with 10, again a recursive call of max\_heapify(Arr, 3) will be performed , and 1 will be swapped with 9. Now further call to max\_heapify(Arr, 7) will have no effect, as 1 is a leaf node now.

In step 5, we finally get a max- heap and the elements in the array Arr will be :



**Min Heap:** In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree.



As you can see in the above diagram, each node has a value smaller than the value of their children.  
We can perform same operations as performed in building max\_heap.  
First we will make function which can maintain the min heap property, if some element is violating it.

void min\_heapify (int Arr[ ] , int i, int N)

{

    int left  = 2\*i;

    int right = 2\*i+1;

    int smallest;

    if(left <= N and Arr[left] < Arr[ i ] )

         smallest = left;

    else

        smallest = i;

    if(right <= N and Arr[right] < Arr[smallest] )

        smallest = right;

    if(smallest != i)

    {

        swap (Arr[ i ], Arr[ smallest ]);

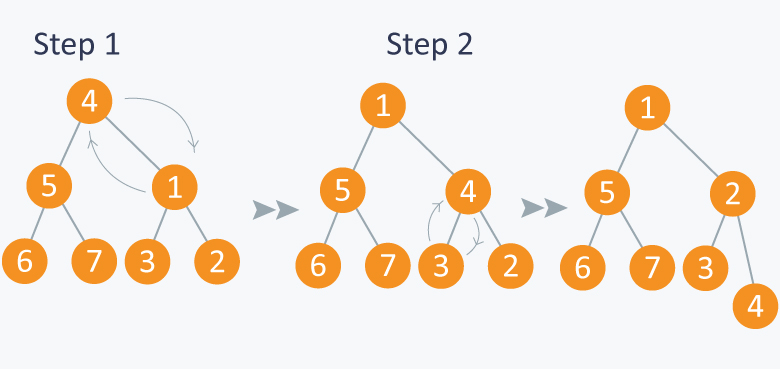
        min\_heapify (Arr, smallest,N);

    }

}

**Complexity:** O(logN) .

**Example:**  
Suppose you have elements stored in array Arr {4, 5, 1, 6, 7, 3, 2}. As you can see in the diagram below, the element at index 1 is violating the property of min -heap, so performing min\_heapify(Arr, 1) will maintain the min-heap.



Now let’s use above function in building min-heap. We will run the above function on remaining nodes other than leaves as leaf nodes are 1 element heap.

void build\_minheap (int Arr[ ])

{

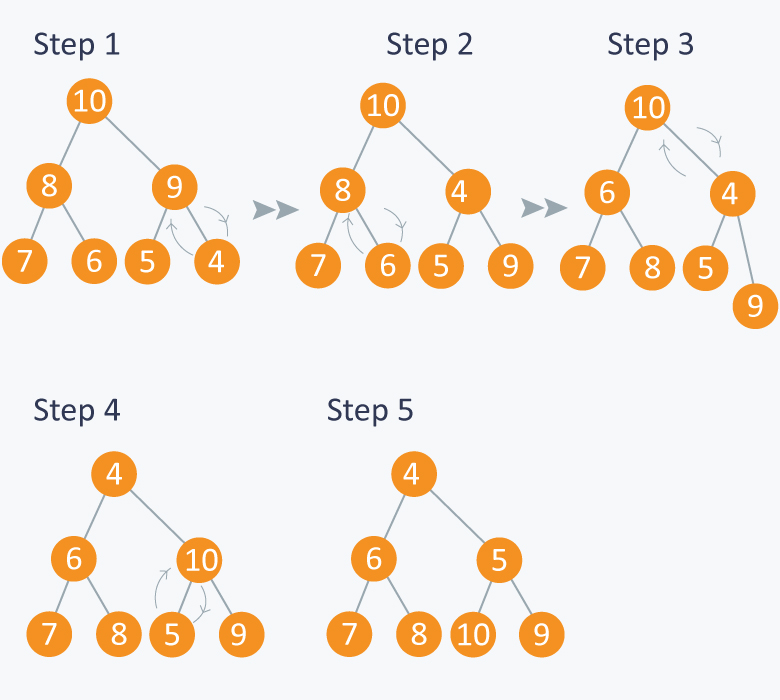
        for( int i = N/2 ; i >= 1 ; i--)

        min\_heapify (Arr, i);

}

**Complexity:** O(N). The complexity calculation is similar to that of building max heap.

**Example:**  
Consider elements in array {10, 8, 9, 7, 6, 5, 4} . We will run min\_heapify on nodes indexed from N/2 to 1. Here node indexed at N/2 has value 9. And at last, we will get a min\_heap.



Heaps can be considered as partially ordered tree, as you can see in the above examples that the nodes of tree do not follow any order with their siblings(nodes on the same level). They can be mainly used when we give more priority to smallest or the largest node in the tree as we can extract these node very efficiently using heaps.

**APPLICATIONS:**

1) **Heap Sort:**

We can use heaps in sorting the elements in a specific order in efficient time.  
Let’s say we want to sort elements of array Arr in ascending order. We can use max heap to perform this operation.

**Idea:** We build the max heap of elements stored in Arr, and the maximum element of Arr will always be at the root of the heap.

Leveraging this idea we can sort an array in the following manner.

**Processing:**

* Initially we will build a max heap of elements in Arr.
* Now the root element that is Arr[1] contains maximum element of Arr. After that, we will exchange this element with the last element of Arr and will again build a max heap excluding the last element which is already in its correct position and will decrease the length of heap by one.
* We will repeat the step 2, until we get all the elements in their correct position.
* We will get a sorted array.

**Implementation:**

Suppose there are N elements stored in array Arr.

void heap\_sort(int Arr[ ])

{

        int heap\_size = N;

        build\_maxheap(Arr);

        for(int i = N; i>=2 ; i-- )

        {

            swap(Arr[ 1 ], Arr[ i ]);

            heap\_size = heap\_size-1;

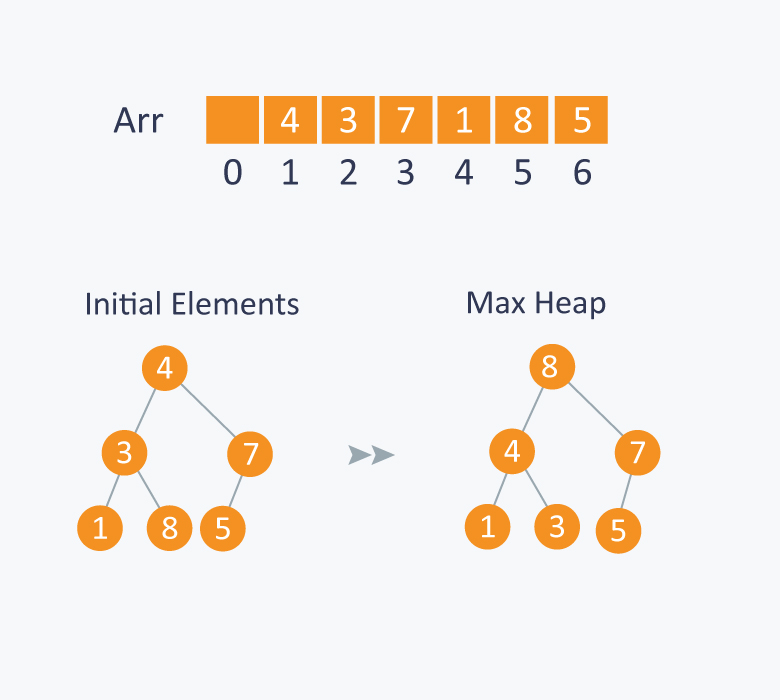
            max\_heapify(Arr, 1, heap\_size);

        }

}

**Complexity:** As we know max\_heapify has complexity O(logN), build\_maxheap has complexity O(N) and we run max\_heapify N−1 times in heap\_sort function, therefore complexity of heap\_sort function is O(NlogN).

**Example:**  
In the diagram below,initially there is an unsorted array Arr having 6 elements. We begin by building max-heap.

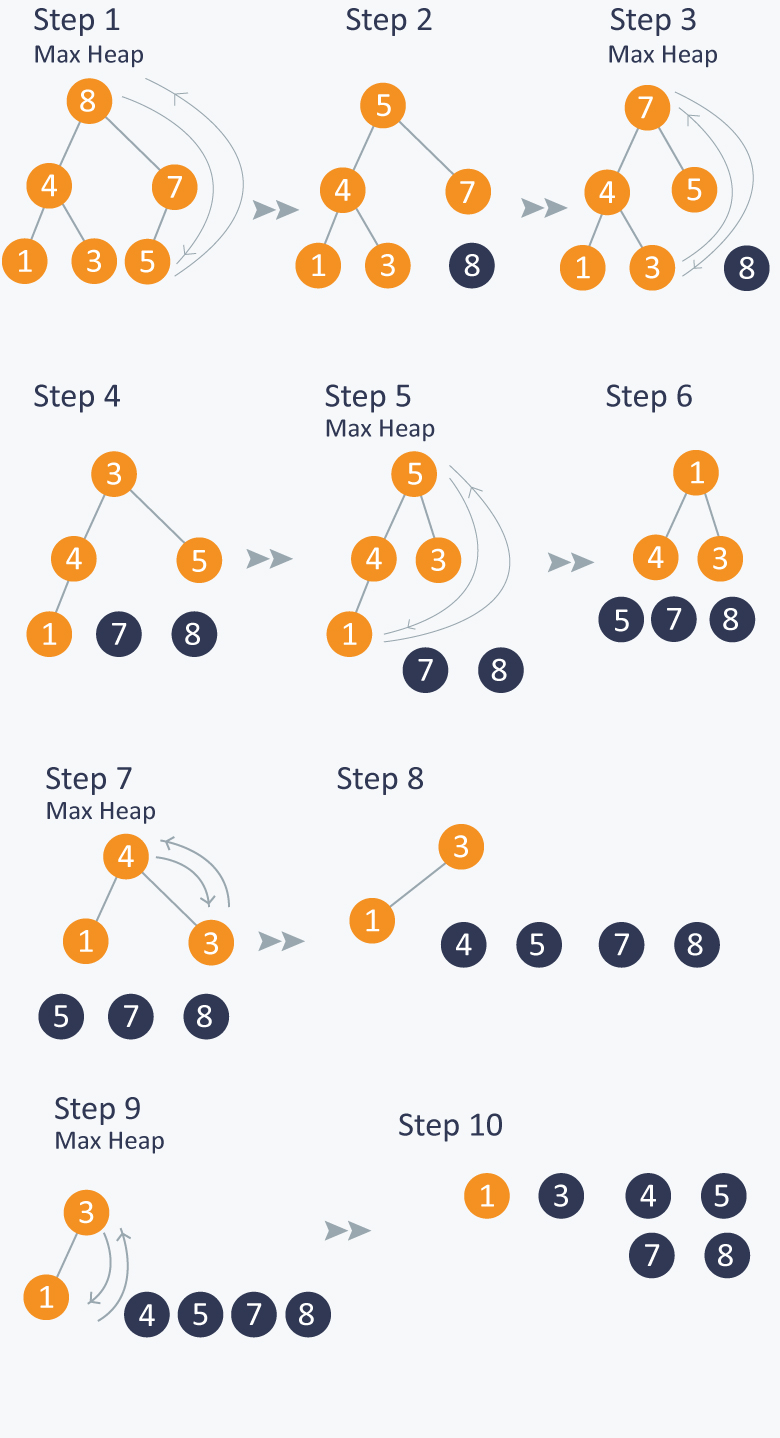


After building max-heap, the elements in the array Arr will be:



**Processing:**

Step 1: 8 is swapped with 5.  
Step 2: 8 is disconnected from heap as 8 is in correct position now.  
Step 3: Max-heap is created and 7 is swapped with 3.  
Step 4: 7 is disconnected from heap.  
Step 5: Max heap is created and 5 is swapped with 1.  
Step 6: 5 is disconnected from heap.  
Step 7: Max heap is created and 4 is swapped with 3.  
Step 8: 4 is disconnected from heap.  
Step 9: Max heap is created and 3 is swapped with 1.  
Step 10: 3 is disconnected.



After all the steps, we will get a sorted array.



2) **Priority Queue:**

Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a difference that the logical order of elements in the priority queue depends on the priority of the elements. The element with highest priority will be moved to the front of the queue and one with lowest priority will move to the back of the queue. Thus it is possible that when you enqueue an element at the back in the queue, it can move to front because of its highest priority.

**Example:**  
Let’s say we have an array of 5 elements : {4, 8, 1, 7, 3} and we have to insert all the elements in the max-priority queue.  
First as the priority queue is empty, so 4 will be inserted initially.  
Now when 8 will be inserted it will moved to front as 8 is greater than 4.  
While inserting 1, as it is the current minimum element in the priority queue, it will remain in the back of priority queue.  
Now 7 will be inserted between 8 and 4 as 7 is smaller than 8.  
Now 3 will be inserted before 1 as it is the 2nd minimum element in the priority queue. All the steps are represented in the diagram below:



We can think of many ways to implement the priority queue.

**Naive Approach:**  
Suppose we have N elements and we have to insert these elements in the priority queue. We can use list and can insert elements in O(N) time and can sort them to maintain a priority queue in O(NlogN) time.

**Efficient Approach:**  
We can use heaps to implement the priority queue. It will take O(logN) time to insert and delete each element in the priority queue.

Based on heap structure, priority queue also has two types max- priority queue and min - priority queue.

Let’s focus on Max Priority Queue.

Max Priority Queue is based on the structure of max heap and can perform following operations:

maximum(Arr) : It returns maximum element from the Arr.  
extract\_maximum (Arr) - It removes and return the maximum element from the Arr.  
increase\_val (Arr, i , val) - It increases the key of element stored at index i in Arr to new value **val**.  
insert\_val (Arr, val ) - It inserts the element with value **val** in Arr.

**Implementation:**

length = number of elements in Arr.

**Maximum** :

int maximum(int Arr[ ])

{

    return Arr[ 1 ];  //as the maximum element is the root element in the max heap.

}

**Complexity:** O(1)

**Extract Maximum:** In this operation, the maximum element will be returned and the last element of heap will be placed at index 1 and max\_heapify will be performed on node 1 as placing last element on index 1 will violate the property of max-heap.

int extract\_maximum (int Arr[ ])

{

    if(length == 0)

    {

        cout<< “Can’t remove element as queue is empty”;

        return -1;

    }

    int max = Arr[1];

    Arr[1] = Arr[length];

    length = length -1;

    max\_heapify(Arr, 1);

    return max;

}

**Complexity:** O(logN).

**Increase Value:** In case increasing value of any node, it may violate the property of max-heap, so we may have to swap the parent’s value with the node’s value until we get a larger value on parent node.

void increase\_value (int Arr[ ], int i, int val)

{

    if(val < Arr[ i ])

    {

        cout<<”New value is less than current value, can’t be inserted” <<endl;

        return;

    }

    Arr[ i ] = val;

    while( i > 1 and Arr[ i/2 ] < Arr[ i ])

    {

        swap(Arr[ i/2 ], Arr[ i ]);

        i = i/2;

    }

}

**Complexity :** O(logN).

**Insert Value :**

void insert\_value (int Arr[ ], int val)

{

    length = length + 1;

    Arr[ length ] = -1;  //assuming all the numbers greater than 0 are to be inserted in queue.

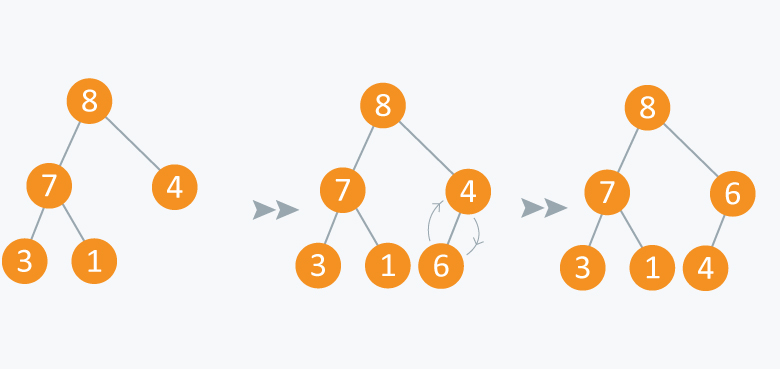
    increase\_val (Arr, length, val);

}

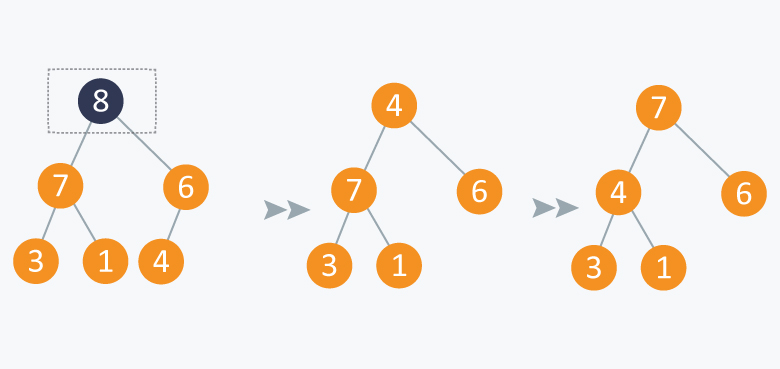
**Complexity:** O(logN).

**Example:**

Initially there are 5 elements in priority queue.  
**Operation:** Insert Value(Arr, 6)  
In the diagram below, inserting another element having value 6 is violating the property of max-priority queue, so it is swapped with its parent having value 4, thus maintaining the max priority queue.



**Operation:** Extract Maximum:  
In the diagram below, after removing 8 and placing 4 at node 1, violates the property of max-priority queue. So max\_heapify(Arr, 1) will be performed which will maintain the property of max - priority queue.



As discussed above, like heaps we can use priority queues in scheduling of jobs. When there are N jobs in queue, each having its own priority. If the job with maximum priority will be completed first and will be removed from the queue, we can use priority queue’s operation extract\_maximum here. If at every instant we have to add a new job in the queue, we can use **insert\_value** operation as it will insert the element in O(logN) and will also maintain the property of max heap.

**Code**

**Min Heap**

public class MinHeap {

    private int capacity;

    private int heapSize;

    int[] heapArray;

    public MinHeap() {

        capacity = 5;

        heapSize = 0;

        heapArray = new int[capacity];

    }

    public MinHeap(int heapCapacity) {

        capacity = heapCapacity;

        heapSize = 0;

        heapArray = new int[heapCapacity];

    }

    public MinHeap(int[] arr) {

        this.capacity = arr.length;

        this.heapSize = arr.length;

        heapArray = new int[arr.length];

        for (int i = 0; i < arr.length; i++)

            heapArray[i] = arr[i];

    }

    public void insertKey(int key) {

        if (heapSize < capacity) {

            heapArray[heapSize] = key;

            heapSize = heapSize + 1;

        }

        else {

            doubleHeapCapacity();

            heapArray[heapSize] = key;

            heapSize = heapSize + 1;

        }

        int i = heapSize - 1;

        while (i > 0 && heapArray[parent(i)] > heapArray[i]) {

            swap(heapArray,parent(i),i);

            i = parent(i);

        }

    }

    private void decreaseKey(int index,int key) {

        heapArray[index] = key;

        while(index > 0 && heapArray[parent(index)] > heapArray[index]) {

            swap(heapArray,parent(index),index);

            index = parent(index);

        }

    }

    private int extractMin() {

        int min = heapArray[0];

        swap(heapArray,0,heapSize-1);

        heapSize = heapSize - 1;

        minHeapify(heapArray,0,heapSize);

        return min;

    }

    private boolean isEmpty() {

        return heapSize == 0;

    }

    private int parent(int i) {

        return (i-1)/2;

    }

    private void swap(int[] A,int i,int j) {

        int temp = A[i];

        A[i] = A[j];

        A[j] = temp;

    }

    private void buildMinHeap() {

        for (int i = heapSize/2; i >=0; i--) {

            minHeapify(heapArray,i,heapSize);

        }

    }

    private void minHeapSort() {

        buildMinHeap();

        for (int i = heapSize-1; i >= 0; i--) {

            swap(heapArray,0,i);

            minHeapify(heapArray,0,i);

        }

    }

    private void minHeapify(int[] A,int i,int n) {

        int left = 2\*i + 1;

        int right = 2\*i + 2;

        int smallest = i;

        if (left < n && A[left] < A[i]) {

            smallest = left;

        }

        if (right < n && A[right] < A[smallest]) {

            smallest = right;

        }

        if (smallest != i) {

            swap(A,i,smallest);

            minHeapify(A,smallest,n);

        }

    }

    private void doubleHeapCapacity() {

        int doubleCapacity = 2\*capacity;

        int[] temp = new int[capacity];

        for (int i = 0; i < heapSize; i++) {

            temp[i] = heapArray[i];

        }

        heapArray = new int[doubleCapacity];

        for (int i = 0; i < capacity; i++)

            heapArray[i] = temp[i];

        this.capacity = doubleCapacity;

    }

    private  void printHeapArray() {

        for (int i = 0; i < heapSize; i++) {

            System.out.print(heapArray[i] + " ");

        }

    }

    public static void main(String[] args) {

        int[] arr = {16,14,10,8,7,9,3,2,4,1};

        MinHeap minHeap = new MinHeap(arr);

        minHeap.buildMinHeap();

       // minHeap.minHeapSort();

        minHeap.printHeapArray();

        minHeap.decreaseKey(3,0);

        System.out.println();

        minHeap.printHeapArray();

//        for (int i = 0; i < 5; i++) {

//            System.out.println(minHeap.extractMin());

//        }

//        minHeap.insertKey(11);

//        System.out.println("added new element ");

//        System.out.println(minHeap.extractMin());

//        System.out.println("\n\nremaining array: ");

//        minHeap.printHeapArray();

    }

}

**Max Heap**

public class MaxHeap {

    private int capacity;

    private int heapSize;

    int[] heapArray;

    public MaxHeap() {

        capacity = 5;

        heapSize = 0;

        heapArray = new int[capacity];

    }

    public MaxHeap(int heapCapacity) {

        capacity = heapCapacity;

        heapSize = 0;

        heapArray = new int[heapCapacity];

    }

    public MaxHeap(int[] arr) {

        this.capacity = arr.length;

        this.heapSize = arr.length;

        heapArray = new int[arr.length];

        for (int i = 0; i < arr.length; i++)

            heapArray[i] = arr[i];

    }

    public void insertKey(int key) {

        if (heapSize < capacity) {

            heapArray[heapSize] = key;

            heapSize = heapSize + 1;

        }

        else {

            doubleHeapCapacity();

            heapArray[heapSize] = key;

            heapSize = heapSize + 1;

        }

        int i = heapSize - 1;

        while (i > 0 && heapArray[parent(i)] < heapArray[i]) {

            swap(heapArray,parent(i),i);

            i = parent(i);

        }

    }

    private int parent(int i) {

        return (i-1)/2;

    }

    private int extractMax() {

        int max = heapArray[0];

        swap(heapArray,0,heapSize-1);

        heapSize = heapSize - 1;

        maxHeapify(heapArray,0,heapSize);

        return max;

    }

    private void doubleHeapCapacity() {

        int doubleCapacity = 2\*capacity;

        int[] temp = new int[capacity];

        for (int i = 0; i < heapSize; i++) {

            temp[i] = heapArray[i];

        }

        heapArray = new int[doubleCapacity];

        for (int i = 0; i < capacity; i++)

            heapArray[i] = temp[i];

        this.capacity = doubleCapacity;

    }

    private void buildMaxHeap() {

        for (int i = heapSize/2; i >=0; i--) {

            maxHeapify(heapArray,i,heapSize);

        }

    }

    private void maxHeapSort() {

        buildMaxHeap();

        for (int i = heapSize-1; i >= 0; i--) {

            swap(heapArray,0,i);

            maxHeapify(heapArray,0,i);

        }

    }

    private void maxHeapify(int[] A,int i,int n) {

        int left = 2\*i + 1;

        int right = 2\*i + 2;

        int largest = i;

        if (left < n && A[left] > A[i]) {

            largest = left;

        }

        if (right < n && A[right] > A[largest]) {

            largest = right;

        }

        if (largest != i) {

            swap(A,i,largest);

            maxHeapify(A,largest,n);

        }

    }

    private void swap(int[] A,int i,int j) {

        int temp = A[i];

        A[i] = A[j];

        A[j] = temp;

    }

    public static void main(String[] args) {

        int[] arr = {16,14,10,8,7,9,3,2,4,1};

        MaxHeap maxHeap = new MaxHeap(arr);

        maxHeap.buildMaxHeap();

       // maxHeap.maxHeapSort();

      //  maxHeap.printHeapArray();

        for (int i = 0; i < 5; i++) {

            System.out.println(maxHeap.extractMax());

        }

        maxHeap.insertKey(11);

        System.out.println("added new element ");

        System.out.println(maxHeap.extractMax());

        System.out.println("\n\nremaining array: ");

        maxHeap.printHeapArray();

    }

    private  void printHeapArray() {

        for (int i = 0; i < heapSize; i++) {

            System.out.print(heapArray[i] + " ");

        }

    }

}

**Heap sort**

public class HeapSort {

    public static void main(String[] args) {

        int[] arr = {16,14,10,8,7,9,3,2,4,1};

        heapSort(arr);

        printArray(arr);

    }

    public static void printArray(int[] arr) {

        for (int i = 0; i < arr.length; i++) {

            System.out.print(arr[i] + " ");

        }

    }

    public static void heapSort(int[] A) {

        int n = A.length;

        buildHeap(A);

        for (int i = n-1; i>= 0; i--) {

            swap(A,0,i);

            heapify(A,0,i);

        }

    }

    public static void buildHeap(int[] A) {

        int n = A.length;

        for (int i = n/2; i >= 0; i--) {

            heapify(A,i,n);

        }

    }

    public static void heapify(int[] A,int i,int n) {

        int left = 2\*i + 1;

        int right = 2\*i + 2;

        int largest = i;

        if (left < n && A[left] > A[i]) {

            largest = left;

        }

        if (right < n && A[right] > A[largest]) {

            largest = right;

        }

        if (largest != i) {

            swap(A,i,largest);

            heapify(A,largest,n);

        }

   }

    public static void swap(int[] arr,int i,int j) {

        int temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

    }

}

**Two heaps**

**Introduction**

In many problems, where we are given a set of elements such that we can divide them into two parts. To solve the problem, we are interested in knowing the smallest element in one part and the biggest element in the other part. This pattern is an efficient approach to solve such problems.

This pattern uses two **Heaps** to solve these problems; A **Min Heap** to find the smallest element and a **Max Heap** to find the biggest element.

# Find the Median of a Number Stream

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3Yj2BmpyEy4#problem-statement)

Design a class to calculate the median of a number stream. The class should have the following two methods:

1. insertNum(int num): stores the number in the class
2. findMedian(): returns the median of all numbers inserted in the class

If the count of numbers inserted in the class is even, the median will be the average of the middle two numbers.

**Example 1:**

1. insertNum(3)  
2. insertNum(1)  
3. findMedian() -> output: 2  
4. insertNum(5)  
5. findMedian() -> output: 3  
6. insertNum(4)  
7. findMedian() -> output: 3.5

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3Yj2BmpyEy4#solution)

As we know, the median is the middle value in an ordered integer list. So a brute force solution could be to maintain a sorted list of all numbers inserted in the class so that we can efficiently return the median whenever required. Inserting a number in a sorted list will take O(N)*O*(*N*) time if there are ‘N’ numbers in the list. This insertion will be similar to the [Insertion sort](https://en.wikipedia.org/wiki/Insertion_sort). Can we do better than this? Can we utilize the fact that we don’t need the fully sorted list - we are only interested in finding the middle element?

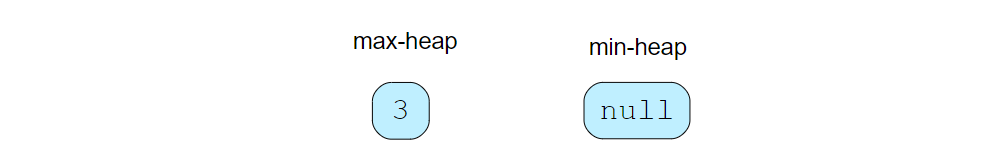
Assume ‘x’ is the median of a list. This means that half of the numbers in the list will be smaller than (or equal to) ‘x’ and half will be greater than (or equal to) ‘x’. This leads us to an approach where we can divide the list into two halves: one half to store all the smaller numbers (let’s call it smallNumList) and one half to store the larger numbers (let’s call it largNumList). The median of all the numbers will either be the largest number in the smallNumList or the smallest number in the largNumList. If the total number of elements is even, the median will be the average of these two numbers.

The best data structure that comes to mind to find the smallest or largest number among a list of numbers is a [Heap](https://en.wikipedia.org/wiki/Heap_(data_structure)). Let’s see how we can use a heap to find a better algorithm.

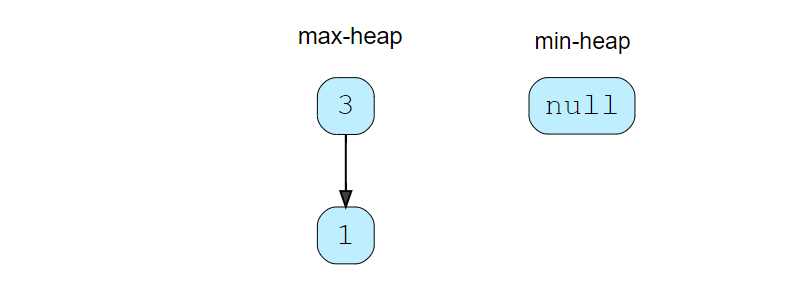
1. We can store the first half of numbers (i.e., smallNumList) in a **Max Heap**. We should use a **Max Heap** as we are interested in knowing the largest number in the first half.
2. We can store the second half of numbers (i.e., largeNumList) in a **Min Heap**, as we are interested in knowing the smallest number in the second half.
3. Inserting a number in a heap will take O(logN)*O*(*logN*), which is better than the brute force approach.
4. At any time, the median of the current list of numbers can be calculated from the top element of the two heaps.

Let’s take the Example-1 mentioned above to go through each step of our algorithm:

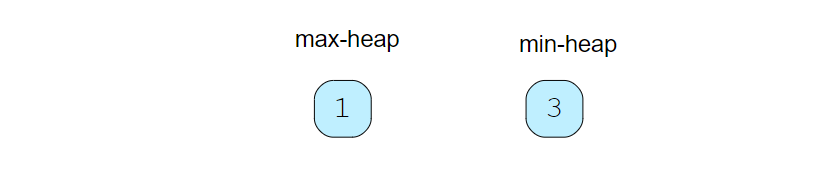
1. insertNum(3): We can insert a number in the **Max Heap** (i.e. first half) if the number is smaller than the top (largest) number of the heap. After every insertion, we will balance the number of elements in both heaps, so that they have an equal number of elements. If the count of numbers is odd, let’s decide to have more numbers in max-heap than the **Min Heap**.



1. insertNum(1): As ‘1’ is smaller than ‘3’, let’s insert it into the **Max Heap**.

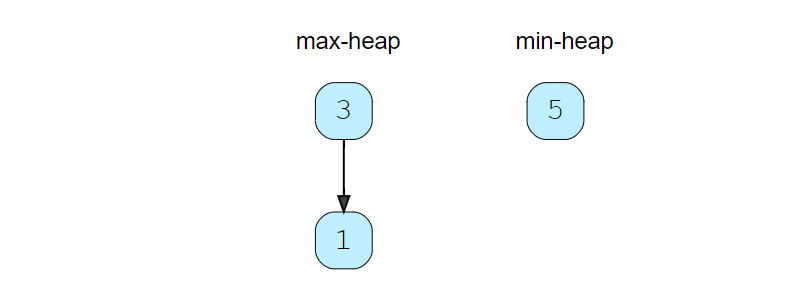


Now, we have two elements in the **Max Heap** and no elements in **Min Heap**. Let’s take the largest element from the **Max Heap** and insert it into the **Min Heap**, to balance the number of elements in both heaps.

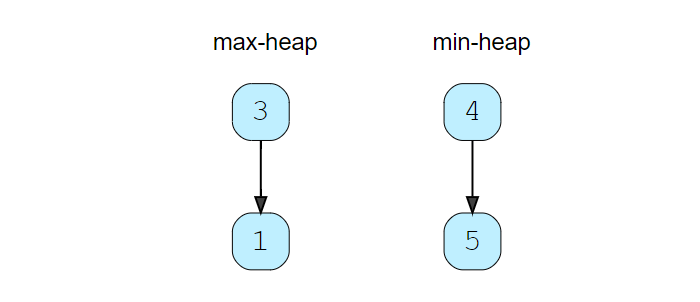


1. findMedian(): As we have an even number of elements, the median will be the average of the top element of both the heaps -> (1+3)/2 = 2.0(1+3)/2=2.0
2. insertNum(5): As ‘5’ is greater than the top element of the **Max Heap**, we can insert it into the **Min Heap**. After the insertion, the total count of elements will be odd. As we had decided to have more numbers in the **Max Heap** than the **Min Heap**, we can take the top (smallest) number from the **Min Heap** and insert it into the **Max Heap**.

[] [1] [5] [1,5] [3] [1,3] [5,3] [1,5,3]



1. findMedian(): Since we have an odd number of elements, the median will be the top element of **Max Heap** -> 3. An odd number of elements also means that the **Max Heap** will have one extra element than the **Min Heap**.
2. insertNum(4): Insert ‘4’ into **Min Heap**.



1. findMedian(): As we have an even number of elements, the median will be the average of the top element of both the heaps -> (3+4)/2 = 3.5(3+4)/2=3.5

#### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3Yj2BmpyEy4#code)

import java.util.\*;

class MedianOfAStream {

  PriorityQueue<Integer> maxHeap; //containing first half of numbers

  PriorityQueue<Integer> minHeap; //containing second half of numbers

  public MedianOfAStream() {

    maxHeap = new PriorityQueue<>((a, b) -> b - a);

    minHeap = new PriorityQueue<>((a, b) -> a - b);

  }

  public void insertNum(int num) {

    if (maxHeap.isEmpty() || maxHeap.peek() >= num)

      maxHeap.add(num);

    else

      minHeap.add(num);

    // either both the heaps will have equal number of elements or max-heap will have one

    // more element than the min-heap

    if (maxHeap.size() > minHeap.size() + 1)

      minHeap.add(maxHeap.poll());

    else if (maxHeap.size() < minHeap.size())

      maxHeap.add(minHeap.poll());

  }

  public double findMedian() {

    if (maxHeap.size() == minHeap.size()) {

      // we have even number of elements, take the average of middle two elements

      return maxHeap.peek() / 2.0 + minHeap.peek() / 2.0;

    }

    // because max-heap will have one more element than the min-heap

    return maxHeap.peek();

  }

  public static void main(String[] args) {

    MedianOfAStream medianOfAStream = new MedianOfAStream();

    medianOfAStream.insertNum(3);

    medianOfAStream.insertNum(1);

    System.out.println("The median is: " + medianOfAStream.findMedian());

    medianOfAStream.insertNum(5);

    System.out.println("The median is: " + medianOfAStream.findMedian());

    medianOfAStream.insertNum(4);

    System.out.println("The median is: " + medianOfAStream.findMedian());

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3Yj2BmpyEy4#time-complexity)

The time complexity of the insertNum() will be O(logN)*O*(*logN*) due to the insertion in the heap. The time complexity of the findMedian() will be O(1)*O*(1) as we can find the median from the top elements of the heaps.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3Yj2BmpyEy4#space-complexity)

The space complexity will be O(N)*O*(*N*) because, as at any time, we will be storing all the numbers.

# Sliding Window Median

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3Y9jm7XRrXO#problem-statement)

Given an array of numbers and a number ‘k’, find the median of all the ‘k’ sized sub-arrays (or windows) of the array.

**Example 1:**

Input: nums=[1, 2, -1, 3, 5], k = 2  
Output: [1.5, 0.5, 1.0, 4.0]  
Explanation: Lets consider all windows of size ‘2’:

* [1, 2, -1, 3, 5] -> median is 1.5
* [1, 2, -1, 3, 5] -> median is 0.5
* [1, 2, -1, 3, 5] -> median is 1.0
* [1, 2, -1, 3, 5] -> median is 4.0

**Example 2:**

Input: nums=[1, 2, -1, 3, 5], k = 3  
Output: [1.0, 2.0, 3.0]  
Explanation: Lets consider all windows of size ‘3’:

* [1, 2, -1, 3, 5] -> median is 1.0
* [1, 2, -1, 3, 5] -> median is 2.0
* [1, 2, -1, 3, 5] -> median is 3.0

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3Y9jm7XRrXO#solution)

This problem follows the **Two Heaps** pattern and share similarities with **Find the Median of a Number stream**. We can follow a similar approach of maintaining a max-heap and a min-heap for the list of numbers to find their median.

The only difference is that we need to keep track of a sliding window of ‘k’ numbers. This means, in each iteration, when we insert a new number in the heaps, we need to remove one number from the heaps which is going out of the sliding window. After the removal, we need to rebalance the heaps in the same way that we did while inserting.

#### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3Y9jm7XRrXO#code)

import java.util.\*;

class SlidingWindowMedian {

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

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

  public double[] findSlidingWindowMedian(int[] nums, int k) {

    double[] result = new double[nums.length - k + 1];

    for (int i = 0; i < nums.length; i++) {

      if (maxHeap.size() == 0 || maxHeap.peek() >= nums[i]) {

        maxHeap.add(nums[i]);

      } else {

        minHeap.add(nums[i]);

      }

      rebalanceHeaps();

      if (i - k + 1 >= 0) { // if we have at least 'k' elements in the sliding window

        // add the median to the the result array

        if (maxHeap.size() == minHeap.size()) {

          // we have even number of elements, take the average of middle two elements

          result[i - k + 1] = maxHeap.peek() / 2.0 + minHeap.peek() / 2.0;

        } else { // because max-heap will have one more element than the min-heap

          result[i - k + 1] = maxHeap.peek();

        }

        // remove the the element going out of the sliding window

        int elementToBeRemoved = nums[i - k + 1];

        if (elementToBeRemoved <= maxHeap.peek()) {

          maxHeap.remove(elementToBeRemoved);

        } else {

          minHeap.remove(elementToBeRemoved);

        }

        rebalanceHeaps();

      }

    }

    return result;

  }

  private void rebalanceHeaps() {

    // either both the heaps will have equal number of elements or max-heap will have

    // one more element than the min-heap

    if (maxHeap.size() > minHeap.size() + 1)

      minHeap.add(maxHeap.poll());

    else if (maxHeap.size() < minHeap.size())

      maxHeap.add(minHeap.poll());

  }

  public static void main(String[] args) {

    SlidingWindowMedian slidingWindowMedian = new SlidingWindowMedian();

    double[] result = slidingWindowMedian.findSlidingWindowMedian(new int[] { 1, 2, -1, 3, 5 }, 2);

    System.out.print("Sliding window medians are: ");

    for (double num : result)

      System.out.print(num + " ");

    System.out.println();

    slidingWindowMedian = new SlidingWindowMedian();

    result = slidingWindowMedian.findSlidingWindowMedian(new int[] { 1, 2, -1, 3, 5 }, 3);

    System.out.print("Sliding window medians are: ");

    for (double num : result)

      System.out.print(num + " ");

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3Y9jm7XRrXO#time-complexity)

The time complexity of our algorithm is O(N\*K) where ‘N’ is the total number of elements in the input array and ‘K’ is the size of the sliding window. This is due to the fact that we are going through all the ‘N’ numbers and, while doing so, we are doing two things:

1. Inserting/removing numbers from heaps of size ‘K’. This will take O(logK)
2. Removing the element going out of the sliding window. This will take O(K) as we will be searching this element in an array of size ‘K’ (i.e., a heap).

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3Y9jm7XRrXO#space-complexity)

Ignoring the space needed for the output array, the space complexity will be O(K) because, at any time, we will be storing all the numbers within the sliding window.

# Maximize Capital

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B6x69OLX4jY#problem-statement)

Given a set of investment projects with their respective profits, we need to find the **most profitable projects**. We are given an initial capital and are allowed to invest only in a fixed number of projects. Our goal is to choose projects that give us the maximum profit.

We can start an investment project only when we have the required capital. Once a project is selected, we can assume that its profit has become our capital.

**Example 1:**

**Input:** Project Capitals=[0,1,2], Project Profits=[1,2,3], Initial Capital=1, Number of Projects=2  
**Output:** 6  
**Explanation:**

1. With initial capital of ‘1’, we will start the second project which will give us profit of ‘2’. Once we selected our first project, our total capital will become 3 (profit + initial capital).
2. With ‘3’ capital, we will select the third project, which will give us ‘3’ profit.

After the completion of the two projects, our total capital will be 6 (1+2+3).

**Example 2:**

**Input:** Project Capitals=[0,1,2,3], Project Profits=[1,2,3,5], Initial Capital=0, Number of Projects=3  
**Output:** 8  
**Explanation:**

1. With ‘0’ capital, we can only select the first project, bringing out capital to 1.
2. Next, we will select the second project, which will bring our capital to 3.
3. Next, we will select the fourth project, giving us a profit of 5.

After selecting the three projects, our total capital will be 8 (1+2+5).

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B6x69OLX4jY#solution)

While selecting projects we have two constraints:

1. We can select a project only when we have the required capital.
2. There is a maximum limit on how many projects we can select.

Since we don’t have any constraint on time, we should choose a project, among the projects for which we have enough capital, which gives us a maximum profit. Following this greedy approach will give us the best solution.

While selecting a project, we will do two things:

1. Find all the projects that we can choose with the available capital.
2. From the list of projects in the 1st step, choose the project that gives us a maximum profit.

We can follow the **Two Heaps** approach similar to **Find the Median of a Number stream**. Here are the steps of our algorithm:

1. Add all project capitals to a min-heap, so that we can select a project with the smallest capital requirement.
2. Go through the top projects of the min-heap and filter the projects that can be completed within our available capital. Insert the profits of all these projects into a max-heap, so that we can choose a project with the maximum profit.
3. Finally, select the top project of the max-heap for investment.
4. Repeat the 2nd and 3rd steps for the required number of projects.

#### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B6x69OLX4jY#code)

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B6x69OLX4jY#time-complexity)

Since, at the most, all the projects will be pushed to both the heaps once, the time complexity of our algorithm is O(NlogN + KlogN), where ‘N’ is the total number of projects and ‘K’ is the number of projects we are selecting.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B6x69OLX4jY#space-complexity)

The space complexity will be O(N) because we will be storing all the projects in the heaps.

### Next Interval

Given an array of intervals, find the next interval of each interval. In a list of intervals, for an interval ‘i’ its next interval ‘j’ will have the smallest ‘start’ greater than or equal to the ‘end’ of ‘i’.

Write a function to return an array containing indices of the next interval of each input interval. If there is no next interval of a given interval, return -1. It is given that none of the intervals have the same start point.

**Example 1:**

**Input:** Intervals [[2,3], [3,4], [5,6]]  
**Output:** [1, 2, -1]  
**Explanation:** The next interval of [2,3] is [3,4] having index ‘1’. Similarly, the next interval of [3,4] is [5,6] having index ‘2’. There is no next interval for [5,6] hence we have ‘-1’.

**Example 2:**

**Input:** Intervals [[3,4], [1,5], [4,6]]  
**Output:** [2, -1, -1]  
**Explanation:** The next interval of [3,4] is [4,6] which has index ‘2’. There is no next interval for [1,5] and [4,6].

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JP8VKGOEpXl#solution)

A brute force solution could be to take one interval at a time and go through all the other intervals to find the next interval. This algorithm will take O(N^2)*O*(*N*​2​​) where ‘N’ is the total number of intervals. Can we do better than that?

We can utilize the **Two Heaps** approach. We can push all intervals into two heaps: one heap to sort the intervals on maximum start time (let’s call it maxStartHeap) and the other on maximum end time (let’s call it maxEndHeap). We can then iterate through all intervals of the `maxEndHeap’ to find their next interval. Our algorithm will have the following steps:

1. Take out the top (having highest end) interval from the maxEndHeap to find its next interval. Let’s call this interval topEnd.
2. Find an interval in the maxStartHeap with the closest start greater than or equal to the start of topEnd. Since maxStartHeap is sorted by ‘start’ of intervals, it is easy to find the interval with the highest ‘start’. Let’s call this interval topStart.
3. Add the index of topStart in the result array as the next interval of topEnd. If we can’t find the next interval, add ‘-1’ in the result array.
4. Put the topStart back in the maxStartHeap, as it could be the next interval of other intervals.
5. Repeat the steps 1-4 until we have no intervals left in maxEndHeap.

#### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JP8VKGOEpXl#code)

import java.util.\*;

class Interval {

  int start = 0;

  int end = 0;

  Interval(int start, int end) {

    this.start = start;

    this.end = end;

  }

}

class NextInterval {

  public static int[] findNextInterval(Interval[] intervals) {

    int n = intervals.length;

    // heap for finding the maximum start

    PriorityQueue<Integer> maxStartHeap = new PriorityQueue<>(n, (i1, i2) -> intervals[i2].start - intervals[i1].start);

    // heap for finding the minimum end

    PriorityQueue<Integer> maxEndHeap = new PriorityQueue<>(n, (i1, i2) -> intervals[i2].end - intervals[i1].end);

    int[] result = new int[n];

    for (int i = 0; i < intervals.length; i++) {

      maxStartHeap.offer(i);

      maxEndHeap.offer(i);

    }

    // go through all the intervals to find each interval's next interval

    for (int i = 0; i < n; i++) {

      int topEnd = maxEndHeap.poll(); // let's find the next interval of the interval which has the highest 'end'

      result[topEnd] = -1; // defaults to -1

      if (intervals[maxStartHeap.peek()].start >= intervals[topEnd].end) {

        int topStart = maxStartHeap.poll();

        // find the the interval that has the closest 'start'

        while (!maxStartHeap.isEmpty() && intervals[maxStartHeap.peek()].start >= intervals[topEnd].end) {

          topStart = maxStartHeap.poll();

        }

        result[topEnd] = topStart;

        maxStartHeap.add(topStart); // put the interval back as it could be the next interval of other intervals

      }

    }

    return result;

  }

  public static void main(String[] args) {

    Interval[] intervals = new Interval[] { new Interval(2, 3), new Interval(3, 4), new Interval(5, 6) };

    int[] result = NextInterval.findNextInterval(intervals);

    System.out.print("Next interval indices are: ");

    for (int index : result)

      System.out.print(index + " ");

    System.out.println();

    intervals = new Interval[] { new Interval(3, 4), new Interval(1, 5), new Interval(4, 6) };

    result = NextInterval.findNextInterval(intervals);

    System.out.print("Next interval indices are: ");

    for (int index : result)

      System.out.print(index + " ");

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JP8VKGOEpXl#time-complexity)

The time complexity of our algorithm will be O(NlogN), where ‘N’ is the total number of intervals.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JP8VKGOEpXl#space-complexity)

The space complexity will be O(N) because we will be storing all the intervals in the heaps.

Top 'K' Elements

# Introduction

Any problem that asks us to find the top/smallest/frequent ‘K’ elements among a given set falls under this pattern.

The best data structure that comes to mind to keep track of ‘K’ elements is [**Heap**](https://en.wikipedia.org/wiki/Heap_(data_structure)). This pattern will make use of the Heap to solve multiple problems dealing with ‘K’ elements at a time from a set of given elements.

# Top 'K' Numbers

### Problem Statement 1[**#**](https://www.educative.io/courses/grokking-the-coding-interview/RM535yM9DW0#problem-statement)

Given an unsorted array of numbers, find the ‘K’ largest numbers in it.

**Example 1:**

Input: [3, 1, 5, 12, 2, 11], K = 3  
Output: [5, 12, 11]

**Example 2:**

Input: [5, 12, 11, -1, 12], K = 3  
Output: [12, 11, 12]

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RM535yM9DW0#solution)

A brute force solution could be to sort the array and return the largest K numbers. The time complexity of such an algorithm will be O(N\*logN)*O*(*N*∗*logN*) as we need to use a sorting algorithm like [Timsort](https://en.wikipedia.org/wiki/Timsort" \t "_blank) if we use Java’s Collection.sort(). Can we do better than that?

The best data structure that comes to mind to keep track of top ‘K’ elements is [Heap](https://en.wikipedia.org/wiki/Heap_(data_structure)). Let’s see if we can use a heap to find a better algorithm.

If we iterate through the array one element at a time and keep ‘K’ largest numbers in a heap such that each time we find a larger number than the smallest number in the heap, we do two things:

1. Take out the smallest number from the heap, and
2. Insert the larger number into the heap.

This will ensure that we always have ‘K’ largest numbers in the heap. The most efficient way to repeatedly find the smallest number among a set of numbers will be to use a min-heap. As we know, we can find the smallest number in a min-heap in constant time O(1), since the smallest number is always at the root of the heap. Extracting the smallest number from a min-heap will take O(logN) (if the heap has ‘N’ elements) as the heap needs to readjust after the removal of an element.

Let’s take Example-1 to go through each step of our algorithm:

Given array: [3, 1, 5, 12, 2, 11], and K=3

1. First, let’s insert ‘K’ elements in the min-heap.
2. After the insertion, the heap will have three numbers [3, 1, 5] with ‘1’ being the root as it is the smallest element.
3. We’ll iterate through the remaining numbers and perform the above-mentioned two steps if we find a number larger than the root of the heap.
4. The 4th number is ‘12’ which is larger than the root (which is ‘1’), so let’s take out ‘1’ and insert ‘12’. Now the heap will have [3, 5, 12] with ‘3’ being the root as it is the smallest element.
5. The 5th number is ‘2’ which is not bigger than the root of the heap (‘3’), so we can skip this as we already have top three numbers in the heap.
6. The last number is ‘11’ which is bigger than the root (which is ‘3’), so let’s take out ‘3’ and insert ‘11’. Finally, the heap has the largest three numbers: [5, 12, 11]

As discussed above, it will take us O(logK) to extract the minimum number from the min-heap. So the overall time complexity of our algorithm will be O(K\*logK+(N-K)\*logK) since, first, we insert ‘K’ numbers in the heap and then iterate through the remaining numbers and at every step, in the worst case, we need to extract the minimum number and insert a new number in the heap. This algorithm is better than O(N\*logN).

### Code

class KLargestNumbers {

  public static List<Integer> findKLargestNumbers(int[] nums, int k) {

    PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>((n1, n2) -> n1 - n2);

    // put first 'K' numbers in the min heap

    for (int i = 0; i < k; i++)

      minHeap.add(nums[i]);

    // go through the remaining numbers of the array, if the number from the array is bigger than the

    // top (smallest) number of the min-heap, remove the top number from heap and add the number from array

    for (int i = k; i < nums.length; i++) {

      if (nums[i] > minHeap.peek()) {

        minHeap.poll();

        minHeap.add(nums[i]);

      }

    }

    // the heap has the top 'K' numbers, return them in a list

    return new ArrayList<>(minHeap);

  }

  public static void main(String[] args) {

    List<Integer> result = KLargestNumbers.findKLargestNumbers(new int[] { 3, 1, 5, 12, 2, 11 }, 3);

    System.out.println("Here are the top K numbers: " + result);

    result = KLargestNumbers.findKLargestNumbers(new int[] { 5, 12, 11, -1, 12 }, 3);

    System.out.println("Here are the top K numbers: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RM535yM9DW0#time-complexity)

As discussed above, the time complexity of this algorithm is O(K\*logK+(N-K)\*logK), which is asymptotically equal to O(N\*logK)

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RM535yM9DW0#space-complexity)

The space complexity will be O(K) since we need to store the top ‘K’ numbers in the heap.

# Kth Smallest Number

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#problem-statement)

Given an unsorted array of numbers, find Kth smallest number in it.

Please note that it is the Kth smallest number in the sorted order, not the Kth distinct element.

**Example 1:**

Input: [1, 5, 12, 2, 11, 5], K = 3  
Output: 5  
Explanation: The 3rd smallest number is '5', as the first two smaller numbers are [1, 2].

**Example 2:**

Input: [1, 5, 12, 2, 11, 5], K = 4  
Output: 5  
Explanation: The 4th smallest number is '5', as the first three small numbers are [1, 2, 5].

**Example 3:**

Input: [5, 12, 11, -1, 12], K = 3  
Output: 11  
Explanation: The 3rd smallest number is '11', as the first two small numbers are [5, -1].

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#solution)

This problem follows the Top ‘K’ Numbers pattern but has two differences:

1. Here we need to find the Kth smallest number, whereas in Top ‘K’ Numbers  we were dealing with ‘K’ largest numbers.
2. In this problem, we need to find only one number (Kth smallest) compared to finding all ‘K’ largest numbers.

We can follow the same approach as discussed in the ‘Top K Elements’ problem. To handle the first difference mentioned above, we can use a max-heap instead of a min-heap. As we know, the root is the biggest element in the max heap. So, since we want to keep track of the ‘K’ smallest numbers, we can compare every number with the root while iterating through all numbers, and if it is smaller than the root, we’ll take the root out and insert the smaller number.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#code)

import java.util.\*;

class KthSmallestNumber {

  public static int findKthSmallestNumber(int[] nums, int k) {

    PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>((n1, n2) -> n2 - n1);

    // put first k numbers in the max heap

    for (int i = 0; i < k; i++)

      maxHeap.add(nums[i]);

    // go through the remaining numbers of the array, if the number from the array is smaller than the

    // top (biggest) number of the heap, remove the top number from heap and add the number from array

    for (int i = k; i < nums.length; i++) {

      if (nums[i] < maxHeap.peek()) {

        maxHeap.poll();

        maxHeap.add(nums[i]);

      }

    }

    // the root of the heap has the Kth smallest number

    return maxHeap.peek();

  }

  public static void main(String[] args) {

    int result = KthSmallestNumber.findKthSmallestNumber(new int[] { 1, 5, 12, 2, 11, 5 }, 3);

    System.out.println("Kth smallest number is: " + result);

    // since there are two 5s in the input array, our 3rd and 4th smallest numbers should be a '5'

    result = KthSmallestNumber.findKthSmallestNumber(new int[] { 1, 5, 12, 2, 11, 5 }, 4);

    System.out.println("Kth smallest number is: " + result);

    result = KthSmallestNumber.findKthSmallestNumber(new int[] { 5, 12, 11, -1, 12 }, 3);

    System.out.println("Kth smallest number is: " + result);  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#time-complexity)

The time complexity of this algorithm is O(K\*logK+(N-K)\*logK)*O*(*K*∗*logK*+(*N*−*K*)∗*logK*), which is asymptotically equal to O(N\*logK)*O*(*N*∗*logK*)

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#space-complexity)

The space complexity will be O(K)*O*(*K*) because we need to store ‘K’ smallest numbers in the heap.

### An Alternate Approach [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#an-alternate-approach)

Alternatively, we can use a **Min Heap** to find the Kth smallest number. We can insert all the numbers in the min-heap and then extract the top ‘K’ numbers from the heap to find the Kth smallest number. Initializing the min-heap with all numbers will take O(N) and extracting ‘K’ numbers will take O(KlogN). Overall, the time complexity of this algorithm will be O(N+KlogN) and the space complexity will be O(N).

# 'K' Closest Points to the Origin

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#problem-statement)

Given an array of points in the a 2D plane, find ‘K’ closest points to the origin.

**Example 1:**

Input: points = [[1,2],[1,3]], K = 1  
Output: [[1,2]]  
Explanation: The Euclidean distance between (1, 2) and the origin is sqrt(5).  
The Euclidean distance between (1, 3) and the origin is sqrt(10).  
Since sqrt(5) < sqrt(10), therefore (1, 2) is closer to the origin.

**Example 2:**

Input: point = [[1, 3], [3, 4], [2, -1]], K = 2  
Output: [[1, 3], [2, -1]]

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#solution)

The [Euclidean distance](https://en.wikipedia.org/wiki/Euclidean_distance) of a point P(x,y) from the origin can be calculated through the following formula:

\sqrt{x^2 + y^2}√​*x*​2​​+*y*​2​​​​​

This problem follows the Top ‘K’ Numbers pattern. The only difference in this problem is that we need to find the closest point (to the origin) as compared to finding the largest numbers.

Following a similar approach, we can use a **Max Heap** to find ‘K’ points closest to the origin. While iterating through all points, if a point (say ‘P’) is closer to the origin than the top point of the max-heap, we will remove that top point from the heap and add ‘P’ to always keep the closest points in the heap.

### Code

import java.util.\*;

class Point {

  int x;

  int y;

  public Point(int x, int y) {

    this.x = x;

    this.y = y;

  }

  public int distFromOrigin() {

    // ignoring sqrt

    return (x \* x) + (y \* y);

  }

}

class KClosestPointsToOrigin {

  public static List<Point> findClosestPoints(Point[] points, int k) {

    PriorityQueue<Point> maxHeap = new PriorityQueue<>((p1, p2) -> p2.distFromOrigin() - p1.distFromOrigin());

    // put first 'k' points in the max heap

    for (int i = 0; i < k; i++)

      maxHeap.add(points[i]);

    // go through the remaining points of the input array, if a point is closer to the origin than the top point

    // of the max-heap, remove the top point from heap and add the point from the input array

    for (int i = k; i < points.length; i++) {

      if (points[i].distFromOrigin() < maxHeap.peek().distFromOrigin()) {

        maxHeap.poll();

        maxHeap.add(points[i]);

      }

    }

    // the heap has 'k' points closest to the origin, return them in a list

    return new ArrayList<>(maxHeap);

  }

  public static void main(String[] args) {

    Point[] points = new Point[] { new Point(1, 3), new Point(3, 4), new Point(2, -1) };

    List<Point> result = KClosestPointsToOrigin.findClosestPoints(points, 2);

    System.out.print("Here are the k points closest the origin: ");

    for (Point p : result)

      System.out.print("[" + p.x + " , " + p.y + "] ");

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#time-complexity)

The time complexity of this algorithm is (N\*logK) as we iterating all points and pushing them into the heap.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#space-complexity)

The space complexity will be O(K) because we need to store ‘K’ point in the heap.

# Connect Ropes

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#problem-statement)

Given ‘N’ ropes with different lengths, we need to connect these ropes into one big rope with minimum cost. The cost of connecting two ropes is equal to the sum of their lengths.

**Example 1:**

Input: [1, 3, 11, 5]  
Output: 33  
Explanation: First connect 1+3(=4), then 4+5(=9), and then 9+11(=20). So the total cost is 33 (4+9+20)

**Example 2:**

Input: [3, 4, 5, 6]  
Output: 36  
Explanation: First connect 3+4(=7), then 5+6(=11), 7+11(=18). Total cost is 36 (7+11+18)

**Example 3:**

Input: [1, 3, 11, 5, 2]  
Output: 42  
Explanation: First connect 1+2(=3), then 3+3(=6), 6+5(=11), 11+11(=22). Total cost is 42 (3+6+11+22)

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#solution)

In this problem, following a greedy approach to connect the smallest ropes first will ensure the lowest cost. We can use a **Min Heap** to find the smallest ropes following a similar approach as discussed in Kth smallest number. Once we connect two ropes, we need to insert the resultant rope back in the **Min Heap** so that we can connect it with the remaining ropes.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#code)

import java.util.\*;

class ConnectRopes {

  public static int minimumCostToConnectRopes(int[] ropeLengths) {

    PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>((n1, n2) -> n1 - n2);

    // add all ropes to the min heap

    for (int i = 0; i < ropeLengths.length; i++)

      minHeap.add(ropeLengths[i]);

    // go through the values of the heap, in each step take top (lowest) rope lengths from the min heap

    // connect them and push the result back to the min heap.

    // keep doing this until the heap is left with only one rope

    int result = 0, temp = 0;

    while (minHeap.size() > 1) {

      temp = minHeap.poll() + minHeap.poll();

      result += temp;

      minHeap.add(temp);

    }

    return result;

  }

  public static void main(String[] args) {

    int result = ConnectRopes.minimumCostToConnectRopes(new int[] { 1, 3, 11, 5 });

    System.out.println("Minimum cost to connect ropes: " + result);

    result = ConnectRopes.minimumCostToConnectRopes(new int[] { 3, 4, 5, 6 });

    System.out.println("Minimum cost to connect ropes: " + result);

    result = ConnectRopes.minimumCostToConnectRopes(new int[] { 1, 3, 11, 5, 2 });

    System.out.println("Minimum cost to connect ropes: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#time-complexity)

Given ‘N’ ropes, we need O(N\*logN) to insert all the ropes in the heap. In each step, while processing the heap, we take out two elements from the heap and insert one. This means we will have a total of ‘N’ steps, having a total time complexity of O(N\*logN).

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#space-complexity)

The space complexity will be O(N) because we need to store all the ropes in the heap.

# Top 'K' Frequent Numbers

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#problem-statement)

Given an unsorted array of numbers, find the top ‘K’ frequently occurring numbers in it.

**Example 1:**

Input: [1, 3, 5, 12, 11, 12, 11], K = 2  
Output: [12, 11]  
Explanation: Both '11' and '12' apeared twice.

**Example 2:**

Input: [5, 12, 11, 3, 11], K = 2  
Output: [11, 5] or [11, 12] or [11, 3]  
Explanation: Only '11' appeared twice, all other numbers appeared once.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#solution)

This problem follows Top K numbers. The only difference is that in this problem, we need to find the most frequently occurring number compared to finding the largest numbers.

We can follow the same approach as discussed in the **Top K Elements** problem. However, in this problem, we first need to know the frequency of each number, for which we can use a **HashMap**. Once we have the frequency map, we can use a **Min Heap** to find the ‘K’ most frequently occurring number. In the **Min Heap**, instead of comparing numbers we will compare their frequencies in order to get frequently occurring numbers

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#code)

import java.util.\*;

class TopKFrequentNumbers {

  public static List<Integer> findTopKFrequentNumbers(int[] nums, int k) {

    // find the frequency of each number

    Map<Integer, Integer> numFrequencyMap = new HashMap<>();

    for (int n : nums)

      numFrequencyMap.put(n, numFrequencyMap.getOrDefault(n, 0) + 1);

    PriorityQueue<Map.Entry<Integer, Integer>> minHeap = new PriorityQueue<Map.Entry<Integer, Integer>>(

        (e1, e2) -> e1.getValue() - e2.getValue());

    // go through all numbers of the numFrequencyMap and push them in the minHeap, which will have

    // top k frequent numbers. If the heap size is more than k, we remove the smallest (top) number

    for (Map.Entry<Integer, Integer> entry : numFrequencyMap.entrySet()) {

      minHeap.add(entry);

      if (minHeap.size() > k) {

        minHeap.poll();

      }

    }

    // create a list of top k numbers

    List<Integer> topNumbers = new ArrayList<>(k);

    while (!minHeap.isEmpty()) {

      topNumbers.add(minHeap.poll().getKey());

    }

    return topNumbers;

  }

  public static void main(String[] args) {

    List<Integer> result = TopKFrequentNumbers.findTopKFrequentNumbers(new int[] { 1, 3, 5, 12, 11, 12, 11 }, 2);

    System.out.println("Here are the K frequent numbers: " + result);

    result = TopKFrequentNumbers.findTopKFrequentNumbers(new int[] { 5, 12, 11, 3, 11 }, 2);

    System.out.println("Here are the K frequent numbers: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#time-complexity)

The time complexity of the above algorithm is O(N+N\*logK).

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#space-complexity)

The space complexity will be O(N). Even though we are storing only ‘K’ numbers in the heap. For the frequency map, however, we need to store all the ‘N’ numbers.

# Frequency Sort

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#problem-statement)

Given a string, sort it based on the decreasing frequency of its characters.

**Example 1:**

Input: "Programming"  
Output: "rrggmmPiano"  
Explanation: 'r', 'g', and 'm' appeared twice, so they need to appear before any other character.

**Example 2:**

Input: "abcbab"  
Output: "bbbaac"  
Explanation: 'b' appeared three times, 'a' appeared twice, and 'c' appeared only once.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#solution)

This problem follows the **Top ‘K’ Elements** pattern, and shares similarities with Top K frequent numbers.

We can follow the same approach as discussed in the Top K frequent numbers problem. First, we will find the frequencies of all characters, then use a max-heap to find the most occurring characters.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#code)

import java.util.\*;

class FrequencySort {

  public static String sortCharacterByFrequency(String str) {

    // find the frequency of each character

    Map<Character, Integer> characterFrequencyMap = new HashMap<>();

    for (char chr : str.toCharArray()) {

      characterFrequencyMap.put(chr, characterFrequencyMap.getOrDefault(chr, 0) + 1);

    }

    PriorityQueue<Map.Entry<Character, Integer>> maxHeap = new PriorityQueue<Map.Entry<Character, Integer>>(

        (e1, e2) -> e2.getValue() - e1.getValue());

    // add all characters to the max heap

    maxHeap.addAll(characterFrequencyMap.entrySet());

    // build a string, appending the most occurring characters first

    StringBuilder sortedString = new StringBuilder(str.length());

    while (!maxHeap.isEmpty()) {

      Map.Entry<Character, Integer> entry = maxHeap.poll();

      for (int i = 0; i < entry.getValue(); i++)

        sortedString.append(entry.getKey());

    }

    return sortedString.toString();

  }

  public static void main(String[] args) {

    String result = FrequencySort.sortCharacterByFrequency("Programming");

    System.out.println("Here is the given string after sorting characters by frequency: " + result);

    result = FrequencySort.sortCharacterByFrequency("abcbab");

    System.out.println("Here is the given string after sorting characters by frequency: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#time-complexity)

The time complexity of the above algorithm is O(D\*logD) where ‘D’ is the number of distinct characters in the input string. This means, in the worst case, when all characters are unique the time complexity of the algorithm will be O(N\*logN) where ‘N’ is the total number of characters in the string.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#space-complexity)

The space complexity will be O(N), as in the worst case, we need to store all the ‘N’ characters in the HashMap.

# Kth Largest Number in a Stream

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#problem-statement)

Design a class to efficiently find the Kth largest element in a stream of numbers.

The class should have the following two things:

1. The constructor of the class should accept an integer array containing initial numbers from the stream and an integer ‘K’.
2. The class should expose a function add(int num) which will store the given number and return the Kth largest number.

**Example 1:**

Input: [3, 1, 5, 12, 2, 11], K = 4  
1. Calling add(6) should return '5'.  
2. Calling add(13) should return '6'.  
2. Calling add(4) should still return '6'.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#solution)

This problem follows the **Top ‘K’ Elements** pattern and shares similarities with Kth Smallest number.

We can follow the same approach as discussed in the ‘Kth Smallest number’ problem. However, we will use a **Min Heap** (instead of a **Max Heap**) as we need to find the Kth largest number.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#code)

import java.util.\*;

class KthLargestNumberInStream {

  PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>((n1, n2) -> n1 - n2);

  final int k;

  public KthLargestNumberInStream(int[] nums, int k) {

    this.k = k;

    // add the numbers in the min heap

    for (int i = 0; i < nums.length; i++)

      add(nums[i]);

  }

  public int add(int num) {

    // add the new number in the min heap

    minHeap.add(num);

    // if heap has more than 'k' numbers, remove one number

    if (minHeap.size() > this.k)

      minHeap.poll();

    // return the 'Kth largest number

    return minHeap.peek();

  }

  public static void main(String[] args) {

    int[] input = new int[] { 3, 1, 5, 12, 2, 11 };

    KthLargestNumberInStream kthLargestNumber = new KthLargestNumberInStream(input, 4);

    System.out.println("4th largest number is: " + kthLargestNumber.add(6));

    System.out.println("4th largest number is: " + kthLargestNumber.add(13));

    System.out.println("4th largest number is: " + kthLargestNumber.add(4));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#time-complexity)

The time complexity of the add() function will be O(logK) since we are inserting the new number in the heap.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#space-complexity)

The space complexity will be O(K) for storing numbers in the heap.

# 'K' Closest Numbers

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#problem-statement)

Given a sorted number array and two integers ‘K’ and ‘X’, find ‘K’ closest numbers to ‘X’ in the array. Return the numbers in the sorted order. ‘X’ is not necessarily present in the array.

**Example 1:**

Input: [5, 6, 7, 8, 9], K = 3, X = 7  
Output: [6, 7, 8]

**Example 2:**

Input: [2, 4, 5, 6, 9], K = 3, X = 6  
Output: [4, 5, 6]

**Example 3:**

Input: [2, 4, 5, 6, 9], K = 3, X = 10  
Output: [5, 6, 9]

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#solution)

This problem follows the Top K Numbers pattern. The biggest difference in this problem is that we need to find the closest (to ‘X’) numbers compared to finding the overall largest numbers. Another difference is that the given array is sorted.

Utilizing a similar approach, we can find the numbers closest to ‘X’ through the following algorithm:

1. Since the array is sorted, we can first find the number closest to ‘X’ through **Binary Search**. Let’s say that number is ‘Y’.
2. The ‘K’ closest numbers to ‘Y’ will be adjacent to ‘Y’ in the array. We can search in both directions of ‘Y’ to find the closest numbers.
3. We can use a heap to efficiently search for the closest numbers. We will take ‘K’ numbers in both directions of ‘Y’ and push them in a **Min Heap** sorted by their absolute difference from ‘X’. This will ensure that the numbers with the smallest difference from ‘X’ (i.e., closest to ‘X’) can be extracted easily from the **Min Heap**.
4. Finally, we will extract the top ‘K’ numbers from the **Min Heap** to find the required numbers.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#code)

import java.util.\*;

class Entry {

  int key;

  int value;

  public Entry(int key, int value) {

    this.key = key;

    this.value = value;

  }

}

class KClosestElements {

  public static List<Integer> findClosestElements(int[] arr, int K, Integer X) {

    int index = binarySearch(arr, X);

    int low = index - K, high = index + K;

    low = Math.max(low, 0); // 'low' should not be less than zero

    high = Math.min(high, arr.length - 1); // 'high' should not be greater the size of the array

    PriorityQueue<Entry> minHeap = new PriorityQueue<>((n1, n2) -> n1.key - n2.key);

    // add all candidate elements to the min heap, sorted by their absolute difference from 'X'

    for (int i = low; i <= high; i++)

      minHeap.add(new Entry(Math.abs(arr[i] - X), i));

    // we need the top 'K' elements having smallest difference from 'X'

    List<Integer> result = new ArrayList<>();

    for (int i = 0; i < K; i++)

      result.add(arr[minHeap.poll().value]);

    Collections.sort(result);

    return result;

  }

  private static int binarySearch(int[] arr, int target) {

    int low = 0;

    int high = arr.length - 1;

    while (low <= high) {

      int mid = low + (high - low) / 2;

      if (arr[mid] == target)

        return mid;

      if (arr[mid] < target) {

        low = mid + 1;

      } else {

        high = mid - 1;

      }

    }

    if (low > 0) {

      return low - 1;

    }

    return low;

  }

  public static void main(String[] args) {

    List<Integer> result = KClosestElements.findClosestElements(new int[] { 5, 6, 7, 8, 9 }, 3, 7);

    System.out.println("'K' closest numbers to 'X' are: " + result);

    result = KClosestElements.findClosestElements(new int[] { 2, 4, 5, 6, 9 }, 3, 6);

    System.out.println("'K' closest numbers to 'X' are: " + result);

    result = KClosestElements.findClosestElements(new int[] { 2, 4, 5, 6, 9 }, 3, 10);

    System.out.println("'K' closest numbers to 'X' are: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#time-complexity)

The time complexity of the above algorithm is O(logN + K\*logK). We need O(logN) for Binary Search and O(K\*logK) to insert the numbers in the **Min Heap**, as well as to sort the output array.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#space-complexity)

The space complexity will be O(K), as we need to put a maximum of 2K numbers in the heap.

### Alternate Solution using Two Pointers [**#**](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#alternate-solution-using-two-pointers)

After finding the number closest to ‘X’ through **Binary Search**, we can use the **Two Pointers** approach ( We will discuss this soon) to find the ‘K’ closest numbers. Let’s say the closest number is ‘Y’. We can have a left pointer to move back from ‘Y’ and a right pointer to move forward from ‘Y’. At any stage, whichever number pointed out by the left or the right pointer gives the smaller difference from ‘X’ will be added to our result list.

To keep the resultant list sorted we can use a **Queue**. So whenever we take the number pointed out by the left pointer, we will append it at the beginning of the list and whenever we take the number pointed out by the right pointer we will append it at the end of the list.

# Maximum Distinct Elements

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#problem-statement)

Given an array of numbers and a number ‘K’, we need to remove ‘K’ numbers from the array such that we are left with maximum distinct numbers.

**Example 1:**

Input: [7, 3, 5, 8, 5, 3, 3], and K=2  
Output: 3  
Explanation: We can remove two occurrences of 3 to be left with 3 distinct numbers [7, 3, 8], we have   
to skip 5 because it is not distinct and occurred twice.   
Another solution could be to remove one instance of '5' and '3' each to be left with three   
distinct numbers [7, 5, 8], in this case, we have to skip 3 because it occurred twice.

**Example 2:**

Input: [3, 5, 12, 11, 12], and K=3  
Output: 2  
Explanation: We can remove one occurrence of 12, after which all numbers will become distinct. Then   
we can delete any two numbers which will leave us 2 distinct numbers in the result.

**Example 3:**

Input: [1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5], and K=2  
Output: 3  
Explanation: We can remove one occurrence of '4' to get three distinct numbers.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#solution)

This problem follows the Top K numbers pattern, and shares similarities with Top K frequent numbers.

We can follow a similar approach as discussed in Top K frequent Numbers problem:

1. First, we will find the frequencies of all the numbers.
2. Then, push all numbers that are not distinct (i.e., have a frequency higher than one) in a **Min Heap** based on their frequencies. At the same time, we will keep a running count of all the distinct numbers.
3. Following a greedy approach, in a stepwise fashion, we will remove the least frequent number from the heap (i.e., the top element of the min-heap), and try to make it distinct. We will see if we can remove all occurrences of a number except one. If we can, we will increment our running count of distinct numbers. We have to also keep a count of how many removals we have done.
4. If after removing elements from the heap, we are still left with some deletions, we have to remove some distinct elements.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#code)

import java.util.\*;

class MaximumDistinctElements {

  public static int findMaximumDistinctElements(int[] nums, int k) {

    int distinctElementsCount = 0;

    if (nums.length <= k)

      return distinctElementsCount;

    // find the frequency of each number

    Map<Integer, Integer> numFrequencyMap = new HashMap<>();

    for (int i : nums)

      numFrequencyMap.put(i, numFrequencyMap.getOrDefault(i, 0) + 1);

    PriorityQueue<Map.Entry<Integer, Integer>> minHeap = new PriorityQueue<Map.Entry<Integer, Integer>>(

        (e1, e2) -> e1.getValue() - e2.getValue());

    // insert all numbers with frequency greater than '1' into the min-heap

    for (Map.Entry<Integer, Integer> entry : numFrequencyMap.entrySet()) {

      if (entry.getValue() == 1)

        distinctElementsCount++;

      else

        minHeap.add(entry);

    }

    // following a greedy approach, try removing the least frequent numbers first from the min-heap

    while (k > 0 && !minHeap.isEmpty()) {

      Map.Entry<Integer, Integer> entry = minHeap.poll();

      // to make an element distinct, we need to remove all of its occurrences except one

      k -= entry.getValue() - 1;

      if (k >= 0)

        distinctElementsCount++;

    }

    // if k > 0, this means we have to remove some distinct numbers

    if (k > 0)

      distinctElementsCount -= k;

    return distinctElementsCount;

  }

  public static void main(String[] args) {

    int result = MaximumDistinctElements.findMaximumDistinctElements(new int[] { 7, 3, 5, 8, 5, 3, 3 }, 2);

    System.out.println("Maximum distinct numbers after removing K numbers: " + result);

    result = MaximumDistinctElements.findMaximumDistinctElements(new int[] { 3, 5, 12, 11, 12 }, 3);

    System.out.println("Maximum distinct numbers after removing K numbers: " + result);

    result = MaximumDistinctElements.findMaximumDistinctElements(new int[] { 1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5 }, 2);

    System.out.println("Maximum distinct numbers after removing K numbers: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#time-complexity)

Since we will insert all numbers in a **HashMap** and a **Min Heap**, this will take O(N\*logN) where ‘N’ is the total input numbers. While extracting numbers from the heap, in the worst case, we will need to take out ‘K’ numbers. This will happen when we have at least ‘K’ numbers with a frequency of two. Since the heap can have a maximum of ‘N/2’ numbers, therefore, extracting an element from the heap will take O(logN) and extracting ‘K’ numbers will take O(KlogN). So overall, the time complexity of our algorithm will be O(N\*logN + KlogN).

We can optimize the above algorithm and only push ‘K’ elements in the heap, as in the worst case we will be extracting ‘K’ elements from the heap. This optimization will reduce the overall time complexity to O(N\*logK + KlogK).

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#space-complexity)

The space complexity will be O(N) as, in the worst case, we need to store all the ‘N’ characters in the **HashMap**.

# Sum of Elements

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#problem-statement)

Given an array, find the sum of all numbers between the K1’th and K2’th smallest elements of that array.

**Example 1:**

Input: [1, 3, 12, 5, 15, 11], and K1=3, K2=6  
Output: 23  
Explanation: The 3rd smallest number is 5 and 6th smallest number 15. The sum of numbers coming  
between 5 and 15 is 23 (11+12).

**Example 2:**

Input: [3, 5, 8, 7], and K1=1, K2=4  
Output: 12  
Explanation: The sum of the numbers between the 1st smallest number (3) and the 4th smallest   
number (8) is 12 (5+7).

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#solution)

This problem follows the Top K numbers pattern, and shares similarities with Kth smallest number.

We can find the sum of all numbers coming between the K1’th and K2’th smallest numbers in the following steps:

1. First, insert all numbers in a min-heap.
2. Remove the first K1 smallest numbers from the min-heap.
3. Now take the next K2-K1-1 numbers out of the heap and add them. This sum will be our required output.

### Code

import java.util.\*;

class SumOfElements {

  public static int findSumOfElements(int[] nums, int k1, int k2) {

    PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>((n1, n2) -> n1 - n2);

    // insert all numbers to the min heap

    for (int i = 0; i < nums.length; i++)

      minHeap.add(nums[i]);

    // remove k1 small numbers from the min heap

    for (int i = 0; i < k1; i++)

      minHeap.poll();

    int elementSum = 0;

    // sum next k2-k1-1 numbers

    for (int i = 0; i < k2 - k1 - 1; i++)

      elementSum += minHeap.poll();

    return elementSum;

  }

  public static void main(String[] args) {

    int result = SumOfElements.findSumOfElements(new int[] { 1, 3, 12, 5, 15, 11 }, 3, 6);

    System.out.println("Sum of all numbers between k1 and k2 smallest numbers: " + result);

    result = SumOfElements.findSumOfElements(new int[] { 3, 5, 8, 7 }, 1, 4);

    System.out.println("Sum of all numbers between k1 and k2 smallest numbers: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#time-complexity)

Since we need to put all the numbers in a min-heap, the time complexity of the above algorithm will be O(N\*logN) where ‘N’ is the total input numbers.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#space-complexity)

The space complexity will be O(N), as we need to store all the ‘N’ numbers in the heap.

### Alternate Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#alternate-solution)

We can iterate the array and use a max-heap to keep track of the top K2 numbers. We can, then, add the top K2-K1-1 numbers in the max-heap to find the sum of all numbers coming between the K1’th and K2’th smallest numbers. Here is what the algorithm will look like:

import java.util.\*;

class SumOfElements {

  public static int findSumOfElements(int[] nums, int k1, int k2) {

    PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>((n1, n2) -> n2 - n1);

    // keep smallest k2 numbers in the max heap

    for (int i = 0; i < nums.length; i++) {

      if (i < k2 - 1) {

        maxHeap.add(nums[i]);

      } else if (nums[i] < maxHeap.peek()) {

        maxHeap.poll(); // as we are interested only in the smallest k2 numbers

        maxHeap.add(nums[i]);

      }

    }

    // get the sum of numbers between k1 and k2 indices

    // these numbers will be at the top of the max heap

    int elementSum = 0;

    for (int i = 0; i < k2 - k1 - 1; i++)

      elementSum += maxHeap.poll();

    return elementSum;

  }

  public static void main(String[] args) {

    int result = SumOfElements.findSumOfElements(new int[] { 1, 3, 12, 5, 15, 11 }, 3, 6);

    System.out.println("Sum of all numbers between k1 and k2 smallest numbers: " + result);

    result = SumOfElements.findSumOfElements(new int[] { 3, 5, 8, 7 }, 1, 4);

    System.out.println("Sum of all numbers between k1 and k2 smallest numbers: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#time-complexity-2)

Since we need to put only the top K2 numbers in the max-heap at any time, the time complexity of the above algorithm will be O(N\*logK2).

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#space-complexity-2)

The space complexity will be O(K2), as we need to store the smallest ‘K2’ numbers in the heap.

# Rearrange String

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#problem-statement)

Given a string, find if its letters can be rearranged in such a way that no two same characters come next to each other.

**Example 1:**

Input: "aappp"  
Output: "papap"  
Explanation: In "papap", none of the repeating characters come next to each other.

**Example 2:**

Input: "Programming"  
Output: "rgmrgmPiano" or "gmringmrPoa" or "gmrPagimnor", etc.  
Explanation: None of the repeating characters come next to each other.

**Example 3:**

Input: "aapa"  
Output: ""  
Explanation: In all arrangements of "aapa", atleast two 'a' will come together e.g., "apaa", "paaa".

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#solution)

This problem follows the Top K numbers pattern. We can follow a greedy approach to find an arrangement of the given string where no two same characters come next to each other.

We can work in a stepwise fashion to create a string with all characters from the input string. Following a greedy approach, we should first append the most frequent characters to the output strings, for which we can use a **Max Heap**. By appending the most frequent character first, we have the best chance to find a string where no two same characters come next to each other.

So in each step, we should append one occurrence of the highest frequency character to the output string. We will not put this character back in the heap to ensure that no two same characters are adjacent to each other. In the next step, we should process the next most frequent character from the heap in the same way and then, at the end of this step, insert the character from the previous step back to the heap after decrementing its frequency.

Following this algorithm, if we can append all the characters from the input string to the output string, we would have successfully found an arrangement of the given string where no two same characters appeared adjacent to each other.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#code)

import java.util.\*;

class RearrangeString {

  public static String rearrangeString(String str) {

    Map<Character, Integer> charFrequencyMap = new HashMap<>();

    for (char chr : str.toCharArray())

      charFrequencyMap.put(chr, charFrequencyMap.getOrDefault(chr, 0) + 1);

    PriorityQueue<Map.Entry<Character, Integer>> maxHeap = new PriorityQueue<Map.Entry<Character, Integer>>(

        (e1, e2) -> e2.getValue() - e1.getValue());

    // add all characters to the max heap

    maxHeap.addAll(charFrequencyMap.entrySet());

    Map.Entry<Character, Integer> previousEntry = null;

    StringBuilder resultString = new StringBuilder(str.length());

    while (!maxHeap.isEmpty()) {

      Map.Entry<Character, Integer> currentEntry = maxHeap.poll();

      // add the previous entry back in the heap if its frequency is greater than zero

      if (previousEntry != null && previousEntry.getValue() > 0)

        maxHeap.offer(previousEntry);

      // append the current character to the result string and decrement its count

      resultString.append(currentEntry.getKey());

      currentEntry.setValue(currentEntry.getValue() - 1);

      previousEntry = currentEntry;

    }

    // if we were successful in appending all the characters to the result string, return it

    return resultString.length() == str.length() ? resultString.toString() : "";

  }

  public static void main(String[] args) {

    System.out.println("Rearranged string: " + RearrangeString.rearrangeString("aappp"));

    System.out.println("Rearranged string: " + RearrangeString.rearrangeString("Programming"));

    System.out.println("Rearranged string: " + RearrangeString.rearrangeString("aapa"));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#time-complexity)

The time complexity of the above algorithm is O(N\*logN) where ‘N’ is the number of characters in the input string.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#space-complexity)

The space complexity will be O(N), as in the worst case, we need to store all the ‘N’ characters in the **HashMap**.

### Rearrange String K Distance Apart

Given a string and a number ‘K’, find if the string can be rearranged such that the same characters are at least ‘K’ distance apart from each other.

**Example 1:**

Input: "mmpp", K=2  
Output: "mpmp" or "pmpm"  
Explanation: All same characters are 2 distance apart.

**Example 2:**

Input: "Programming", K=3  
Output: "rgmPrgmiano" or "gmringmrPoa" or "gmrPagimnor" and a few more  
Explanation: All same characters are 3 distance apart.

**Example 3:**

Input: "aab", K=2  
Output: "aba"  
Explanation: All same characters are 2 distance apart.

**Example 4:**

Input: "aappa", K=3  
Output: ""  
Explanation: We cannot find an arrangement of the string where any two 'a' are 3 distance apart.

### Try it yourself [**#**](https://www.educative.io/courses/grokking-the-coding-interview/m2E4y26k3LE#try-it-yourself)

import java.util.\*;

class RearrangeStringKDistanceApart {

  public static String reorganizeString(String str, int k) {

    // TODO: Write your code here

    return "";

  }

  public static void main(String[] args) {

    System.out.println("Reorganized string: " +

            RearrangeStringKDistanceApart.reorganizeString("mmpp", 2));

    System.out.println("Reorganized string: " +

            RearrangeStringKDistanceApart.reorganizeString("Programming", 3));

    System.out.println("Reorganized string: " +

            RearrangeStringKDistanceApart.reorganizeString("aab", 2));

    System.out.println("Reorganized string: " +

            RearrangeStringKDistanceApart.reorganizeString("aappa", 3));

  }

}

### Scheduling Tasks [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JYB20zgR32o#scheduling-tasks-hard)

You are given a list of tasks that need to be run, in any order, on a server. Each task will take one CPU interval to execute but once a task has finished, it has a cooling period during which it can’t be run again. If the cooling period for all tasks is ‘K’ intervals, find the minimum number of CPU intervals that the server needs to finish all tasks.

If at any time the server can’t execute any task then it must stay idle.

**Example 1:**

Input: [a, a, a, b, c, c], K=2  
Output: 7  
Explanation: a -> c -> b -> a -> c -> idle -> a

**Example 2:**

Input: [a, b, a], K=3  
Output: 5  
Explanation: a -> b -> idle -> idle -> a

### Try it yourself [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JYB20zgR32o#try-it-yourself)

import java.util.\*;

class TaskScheduler {

  public static int scheduleTasks(char[] tasks, int k) {

    // TODO: Write your code here

    return -1;

  }

  public static void main(String[] args) {

    char[] tasks = new char[] { 'a', 'a', 'a', 'b', 'c', 'c' };

    System.out.println("Minimum intervals needed to execute all tasks: " + TaskScheduler.scheduleTasks(tasks, 2));

    tasks = new char[] { 'a', 'b', 'a' };

    System.out.println("Minimum intervals needed to execute all tasks: " + TaskScheduler.scheduleTasks(tasks, 3));

  }

}

### Frequency Stack [**#**](https://www.educative.io/courses/grokking-the-coding-interview/Y5zDWlVRz2p#frequency-stack-hard)

Design a class that simulates a Stack data structure, implementing the following two operations:

1. push(int num): Pushes the number ‘num’ on the stack.
2. pop(): Returns the most frequent number in the stack. If there is a tie, return the number which was pushed later.

**Example:**

After following push operations: push(1), push(2), push(3), push(2), push(1), push(2), push(5)  
   
1. pop() should return 2, as it is the most frequent number  
2. Next pop() should return 1  
3. Next pop() should return 2

### Try it yourself [**#**](https://www.educative.io/courses/grokking-the-coding-interview/Y5zDWlVRz2p#try-it-yourself)

import java.util.\*;

class FrequencyStack {

  public void push(int num) {

    // TODO: Write your code here

  }

  public int pop() {

    // TODO: Write your code here

    return -1;

  }

  public static void main(String[] args) {

    FrequencyStack frequencyStack = new FrequencyStack();

    frequencyStack.push(1);

    frequencyStack.push(2);

    frequencyStack.push(3);

    frequencyStack.push(2);

    frequencyStack.push(1);

    frequencyStack.push(2);

    frequencyStack.push(5);

    System.out.println(frequencyStack.pop());

    System.out.println(frequencyStack.pop());

    System.out.println(frequencyStack.pop());

  }

}

**K-way merge**

# Introduction

This pattern helps us solve problems that involve a list of sorted arrays.

Whenever we are given ‘K’ sorted arrays, we can use a **Heap** to efficiently perform a sorted traversal of all the elements of all arrays. We can push the smallest (first) element of each sorted array in a **Min Heap** to get the overall minimum. While inserting elements to the **Min Heap** we keep track of which array the element came from. We can, then, remove the top element from the heap to get the smallest element and push the next element from the same array, to which this smallest element belonged, to the heap. We can repeat this process to make a sorted traversal of all elements.

# Merge K Sorted Lists

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/Y5n0n3vAgYK#problem-statement)

Given an array of ‘K’ sorted LinkedLists, merge them into one sorted list.

**Example 1:**

Input: L1=[2, 6, 8], L2=[3, 6, 7], L3=[1, 3, 4]  
Output: [1, 2, 3, 3, 4, 6, 6, 7, 8]

**Example 2:**

Input: L1=[5, 8, 9], L2=[1, 7]  
Output: [1, 5, 7, 8, 9]

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/Y5n0n3vAgYK#solution)

A brute force solution could be to add all elements of the given ‘K’ lists to one list and sort it. If there are a total of ‘N’ elements in all the input lists, then the brute force solution will have a time complexity of O(N\*logN)*O*(*N*∗*logN*) as we will need to sort the merged list. Can we do better than this? How can we utilize the fact that the input lists are individually sorted?

If we have to find the smallest element of all the input lists, we have to compare only the smallest (i.e. the first) element of all the lists. Once we have the smallest element, we can put it in the merged list. Following a similar pattern, we can then find the next smallest element of all the lists to add it to the merged list.

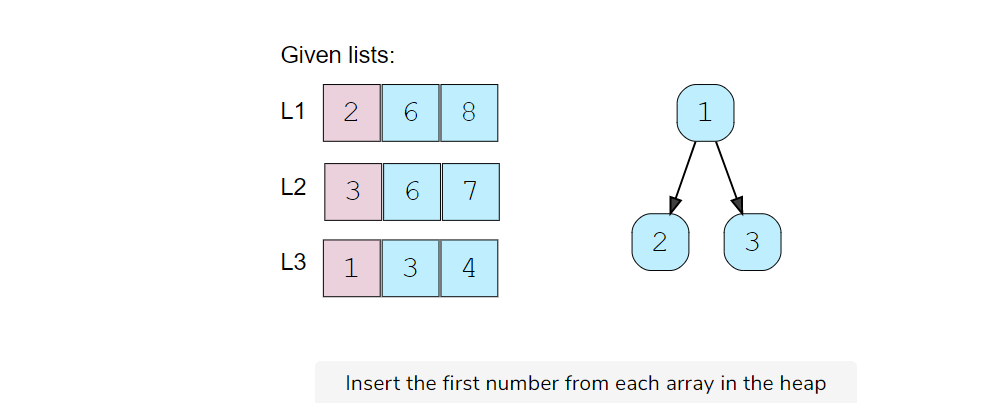
The best data structure that comes to mind to find the smallest number among a set of ‘K’ numbers is a [**Heap**](https://en.wikipedia.org/wiki/Heap_(data_structure)). Let’s see how can we use a heap to find a better algorithm.

1. We can insert the first element of each array in a **Min Heap**.
2. After this, we can take out the smallest (top) element from the heap and add it to the merged list.
3. After removing the smallest element from the heap, we can insert the next element of the same list into the heap.
4. We can repeat steps 2 and 3 to populate the merged list in sorted order.

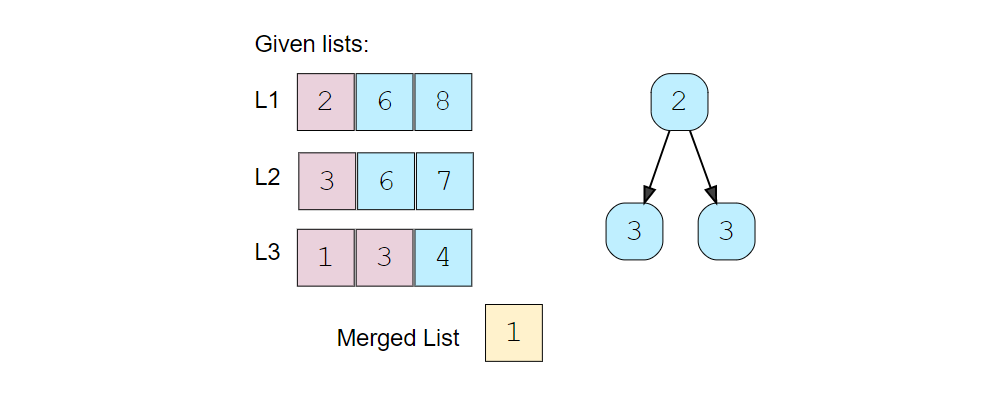
Let’s take the Example-1 mentioned above to go through each step of our algorithm:

Given lists: L1=[2, 6, 8], L2=[3, 6, 7], L3=[1, 3, 4]

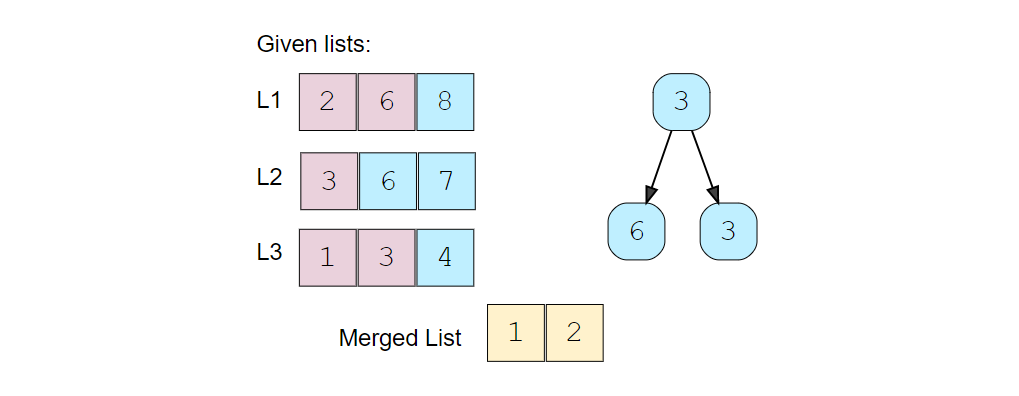
1. After inserting the 1st element of each list, the heap will have the following elements:



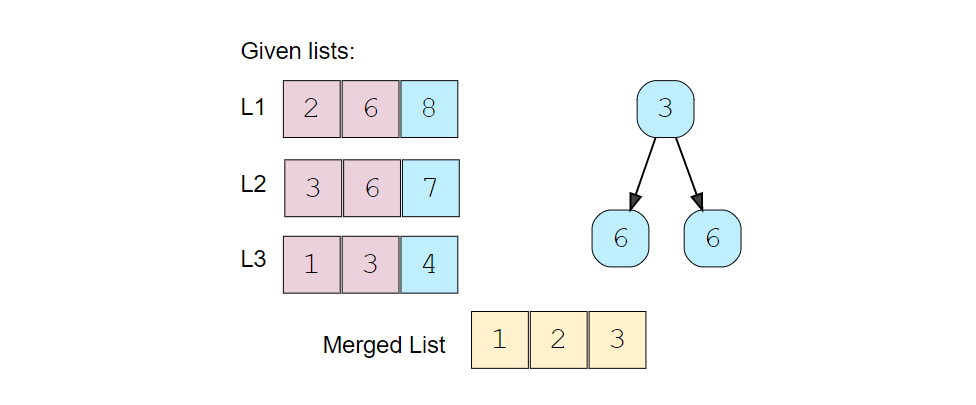
1. We’ll take the top number from the heap, insert it into the merged list and add the next number in the heap.



1. Again, we’ll take the top element of the heap, insert it into the merged list and add the next number to the heap.



1. Repeating the above step, take the top element of the heap, insert it into the merged list and add the next number to the heap. As there are two 3s in the heap, we can pick anyone but we need to take the next element from the corresponding list to insert in the heap.



### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/Y5n0n3vAgYK#code)

import java.util.\*;

class ListNode {

  int value;

  ListNode next;

  ListNode(int value) {

    this.value = value;

  }

}

class MergeKSortedLists {

  public static ListNode merge(ListNode[] lists) {

    PriorityQueue<ListNode> minHeap = new PriorityQueue<ListNode>((n1, n2) -> n1.value - n2.value);

    // put the root of each list in the min heap

    for (ListNode root : lists)

      if (root != null)

        minHeap.add(root);

    // take the smallest (top) element form the min-heap and add it to the result;

    // if the top element has a next element add it to the heap

    ListNode resultHead = null, resultTail = null;

    while (!minHeap.isEmpty()) {

      ListNode node = minHeap.poll();

      if (resultHead == null) {

        resultHead = resultTail = node;

      } else {

        resultTail.next = node;

        resultTail = resultTail.next;

      }

      if (node.next != null)

        minHeap.add(node.next);

    }

    return resultHead;

  }

  public static void main(String[] args) {

    ListNode l1 = new ListNode(2);

    l1.next = new ListNode(6);

    l1.next.next = new ListNode(8);

    ListNode l2 = new ListNode(3);

    l2.next = new ListNode(6);

    l2.next.next = new ListNode(7);

    ListNode l3 = new ListNode(1);

    l3.next = new ListNode(3);

    l3.next.next = new ListNode(4);

    ListNode result = MergeKSortedLists.merge(new ListNode[] { l1, l2, l3 });

    System.out.print("Here are the elements form the merged list: ");

    while (result != null) {

      System.out.print(result.value + " ");

      result = result.next;

    }

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/Y5n0n3vAgYK#time-complexity)

Since we’ll be going through all the elements of all arrays and will be removing/adding one element to the heap in each step, the time complexity of the above algorithm will be O(N\*logK), where ‘N’ is the total number of elements in all the ‘K’ input arrays.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/Y5n0n3vAgYK#space-complexity)

The space complexity will be O(K) because, at any time, our min-heap will be storing one number from all the ‘K’ input arrays.

# Kth Smallest Number in M Sorted Lists

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/myAqDMyRXn3#problem-statement)

Given ‘M’ sorted arrays, find the K’th smallest number among all the arrays.

**Example 1:**

Input: L1=[2, 6, 8], L2=[3, 6, 7], L3=[1, 3, 4], K=5  
Output: 4  
Explanation: The 5th smallest number among all the arrays is 4, this can be verified from the merged   
list of all the arrays: [1, 2, 3, 3, 4, 6, 6, 7, 8]

**Example 2:**

Input: L1=[5, 8, 9], L2=[1, 7], K=3  
Output: 7  
Explanation: The 3rd smallest number among all the arrays is 7.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/myAqDMyRXn3#solution)

This problem follows the **K-way merge** pattern and we can follow a similar approach as discussed in Merge K sorted list.

We can start merging all the arrays, but instead of inserting numbers into a merged list, we will keep count to see how many elements have been inserted in the merged list. Once that count is equal to ‘K’, we have found our required number.

A big difference from Merge K sorted list is that in this problem, the input is a list of arrays compared to LinkedLists. This means that when we want to push the next number in the heap we need to know what the index of the current number in the current array was. To handle this, we will need to keep track of the array and the element indices.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/myAqDMyRXn3#code)

import java.util.\*;

class Node {

  int elementIndex;

  int arrayIndex;

  Node(int elementIndex, int arrayIndex) {

    this.elementIndex = elementIndex;

    this.arrayIndex = arrayIndex;

  }

}

class KthSmallestInMSortedArrays {

  public static int findKthSmallest(List<Integer[]> lists, int k) {

    PriorityQueue<Node> minHeap = new PriorityQueue<Node>(

        (n1, n2) -> lists.get(n1.arrayIndex)[n1.elementIndex] - lists.get(n2.arrayIndex)[n2.elementIndex]);

    // put the 1st element of each array in the min heap

    for (int i = 0; i < lists.size(); i++)

      if (lists.get(i) != null)

        minHeap.add(new Node(0, i));

    // take the smallest (top) element form the min heap, if the running count is equal to k return the number

    // if the array of the top element has more elements, add the next element to the heap

    int numberCount = 0, result = 0;

    while (!minHeap.isEmpty()) {

      Node node = minHeap.poll();

      result = lists.get(node.arrayIndex)[node.elementIndex];

      if (++numberCount == k)

        break;

      node.elementIndex++;

      if (lists.get(node.arrayIndex).length > node.elementIndex)

        minHeap.add(node);

    }

    return result;

  }

  public static void main(String[] args) {

    Integer[] l1 = new Integer[] { 2, 6, 8 };

    Integer[] l2 = new Integer[] { 3, 6, 7 };

    Integer[] l3 = new Integer[] { 1, 3, 4 };

    List<Integer[]> lists = new ArrayList<Integer[]>();

    lists.add(l1);

    lists.add(l2);

    lists.add(l3);

    int result = KthSmallestInMSortedArrays.findKthSmallest(lists, 5);

    System.out.print("Kth smallest number is: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/myAqDMyRXn3#time-complexity)

Since we’ll be going through at most ‘K’ elements among all the arrays, and we will remove/add one element in the heap in each step, the time complexity of the above algorithm will be O(K\*logM) where ‘M’ is the total number of input arrays.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/myAqDMyRXn3#space-complexity)

The space complexity will be O(M) because, at any time, our min-heap will be storing one number from all the ‘M’ input arrays.

### Similar Problems [**#**](https://www.educative.io/courses/grokking-the-coding-interview/myAqDMyRXn3#similar-problems)

**Problem 1:** Given ‘M’ sorted arrays, find the median number among all arrays.

**Solution:** This problem is similar to our parent problem with K=Median. So if there are ‘N’ total numbers in all the arrays we need to find the K’th minimum number where K=N/2*K*=*N*/2.

**Problem 2:** Given a list of ‘K’ sorted arrays, merge them into one sorted list.

**Solution:** This problem is similar to Merge K sorted list except that the input is a list of arrays compared to **LinkedLists**. To handle this, we can use a similar approach as discussed in our parent problem by keeping a track of the array and the element indices.

# Kth Smallest Number in a Sorted Matrix

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/x1NJVYKNvqz#problem-statement)

Given an N \* N*N*∗*N* matrix where each row and column is sorted in ascending order, find the Kth smallest element in the matrix.

**Example 1:**

Input: Matrix=[  
    [2, 6, 8],   
    [3, 7, 10],  
    [5, 8, 11]  
  ],   
  K=5  
Output: 7  
Explanation: The 5th smallest number in the matrix is 7.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/x1NJVYKNvqz#solution)

This problem follows the **K-way merge** pattern and can be easily converted to Kth smallest number in m sorted list. As each row (or column) of the given matrix can be seen as a sorted list, we essentially need to find the Kth smallest number in ‘N’ sorted lists.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/x1NJVYKNvqz#code)

import java.util.\*;

class Node {

  int row;

  int col;

  Node(int row, int col) {

    this.row = row;

    this.col = col;

  }

}

class KthSmallestInSortedMatrix {

  public static int findKthSmallest(int[][] matrix, int k) {

    PriorityQueue<Node> minHeap = new PriorityQueue<Node>((n1, n2) -> matrix[n1.row][n1.col] - matrix[n2.row][n2.col]);

    // put the 1st element of each row in the min heap

    // we don't need to push more than 'k' elements in the heap

    for (int i = 0; i < matrix.length && i < k; i++)

      minHeap.add(new Node(i, 0));

    // take the smallest (top) element form the min heap, if the running count is equal to k return the number

    // if the row of the top element has more elements, add the next element to the heap

    int numberCount = 0, result = 0;

    while (!minHeap.isEmpty()) {

      Node node = minHeap.poll();

      result = matrix[node.row][node.col];

      if (++numberCount == k)

        break;

      node.col++;

      if (matrix[0].length > node.col)

        minHeap.add(node);

    }

    return result;

  }

  public static void main(String[] args) {

    int matrix[][] = { { 2, 6, 8 }, { 3, 7, 10 }, { 5, 8, 11 } };

    int result = KthSmallestInSortedMatrix.findKthSmallest(matrix, 5);

    System.out.print("Kth smallest number is: " + result);

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/x1NJVYKNvqz#time-complexity)

First, we inserted at most ‘K’ or one element from each of the ‘N’ rows, which will take O(min(K, N)). Then we went through at most ‘K’ elements in the matrix and remove/add one element in the heap in each step. As we can’t have more than ‘N’ elements in the heap in any condition, therefore, the overall time complexity of the above algorithm will be O(min(K, N) + K\*logN).

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/x1NJVYKNvqz#space-complexity)

The space complexity will be O(N) because, in the worst case, our min-heap will be storing one number from each of the ‘N’ rows.

# Smallest Number Range

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JPGWDNRx3w2#problem-statement)

Given ‘M’ sorted arrays, find the smallest range that includes at least one number from each of the ‘M’ lists.

**Example 1:**

Input: L1=[1, 5, 8], L2=[4, 12], L3=[7, 8, 10]  
Output: [4, 7]  
Explanation: The range [4, 7] includes 5 from L1, 4 from L2 and 7 from L3.

**Example 2:**

Input: L1=[1, 9], L2=[4, 12], L3=[7, 10, 16]  
Output: [9, 12]  
Explanation: The range [9, 12] includes 9 from L1, 12 from L2 and 10 from L3.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JPGWDNRx3w2#solution)

This problem follows the **K-way merge** pattern and we can follow a similar approach as discussed in Merge K sorted list.

We can start by inserting the first number from all the arrays in a min-heap. We will keep track of the largest number that we have inserted in the heap (let’s call it currentMaxNumber).

In a loop, we’ll take the smallest (top) element from the min-heap andcurrentMaxNumber has the largest element that we inserted in the heap. If these two numbers give us a smaller range, we’ll update our range. Finally, if the array of the top element has more elements, we’ll insert the next element to the heap.

We can finish searching the minimum range as soon as an array is completed or, in other terms, the heap has less than ‘M’ elements.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/JPGWDNRx3w2#code)

import java.util.\*;

class Node {

  int elementIndex;

  int arrayIndex;

  Node(int elementIndex, int arrayIndex) {

    this.elementIndex = elementIndex;

    this.arrayIndex = arrayIndex;

  }

}

class SmallestRange {

  public static int[] findSmallestRange(List<Integer[]> lists) {

    PriorityQueue<Node> minHeap = new PriorityQueue<Node>(

        (n1, n2) -> lists.get(n1.arrayIndex)[n1.elementIndex] - lists.get(n2.arrayIndex)[n2.elementIndex]);

    int rangeStart = 0, rangeEnd = Integer.MAX\_VALUE, currentMaxNumber = Integer.MIN\_VALUE;

    // put the 1st element of each array in the min heap

    for (int i = 0; i < lists.size(); i++)

      if (lists.get(i) != null) {

        minHeap.add(new Node(0, i));

        currentMaxNumber = Math.max(currentMaxNumber, lists.get(i)[0]);

      }

    // take the smallest (top) element form the min heap, if it gives us smaller range, update the ranges

    // if the array of the top element has more elements, insert the next element in the heap

    while (minHeap.size() == lists.size()) {

      Node node = minHeap.poll();

      if (rangeEnd - rangeStart > currentMaxNumber - lists.get(node.arrayIndex)[node.elementIndex]) {

        rangeStart = lists.get(node.arrayIndex)[node.elementIndex];

        rangeEnd = currentMaxNumber;

      }

      node.elementIndex++;

      if (lists.get(node.arrayIndex).length > node.elementIndex) {

        minHeap.add(node); // insert the next element in the heap

        currentMaxNumber = Math.max(currentMaxNumber, lists.get(node.arrayIndex)[node.elementIndex]);

      }

    }

    return new int[] { rangeStart, rangeEnd };

  }

  public static void main(String[] args) {

    Integer[] l1 = new Integer[] { 1, 5, 8 };

    Integer[] l2 = new Integer[] { 4, 12 };

    Integer[] l3 = new Integer[] { 7, 8, 10 };

    List<Integer[]> lists = new ArrayList<Integer[]>();

    lists.add(l1);

    lists.add(l2);

    lists.add(l3);

    int[] result = SmallestRange.findSmallestRange(lists);

    System.out.print("Smallest range is: [" + result[0] + ", " + result[1] + "]");

  }

}

### K Pairs with Largest Sums

Given two sorted arrays in descending order, find ‘K’ pairs with the largest sum where each pair consists of numbers from both the arrays.

**Example 1:**

Input: L1=[9, 8, 2], L2=[6, 3, 1], K=3  
Output: [9, 3], [9, 6], [8, 6]   
Explanation: These 3 pairs have the largest sum. No other pair has a sum larger than any of these.

**Example 2:**

Input: L1=[5, 2, 1], L2=[2, -1], K=3  
Output: [5, 2], [5, -1], [2, 2]

### Try it yourself [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1JKxRB8EDJ#try-it-yourself)

import java.util.\*;

class LargestPairs {

  public static List<int[]> findKLargestPairs(int[] nums1, int[] nums2, int k) {

    List<int[]> result = new ArrayList<>();

    // TODO: Write your code here

    return result;

  }

  public static void main(String[] args) {

    int[] l1 = new int[] { 9, 8, 2 };

    int[] l2 = new int[] { 6, 3, 1 };

    List<int[]> result = LargestPairs.findKLargestPairs(l1, l2, 3);

    System.out.print("Pairs with largest sum are: ");

    for (int[] pair : result)

      System.out.print("[" + pair[0] + ", " + pair[1] + "] ");

  }

}