**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.

Let’s jump onto our first problem to develop an understanding of this pattern.

# Top 'K' Numbers (easy)

**We'll cover the following**

* + - [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/RM535yM9DW0#Problem-Statement)
    - [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/RM535yM9DW0#Try-it-yourself)
  + [Solution](https://www.educative.io/courses/grokking-the-coding-interview/RM535yM9DW0#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/RM535yM9DW0#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/RM535yM9DW0#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/RM535yM9DW0#Space-complexity)

### Problem Statement[**#**](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.

Note: For a detailed discussion about different approaches to solve this problem, take a look at [Kth Smallest Number](https://www.educative.io/collection/page/5668639101419520/5671464854355968/4817079184130048).

**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) 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.

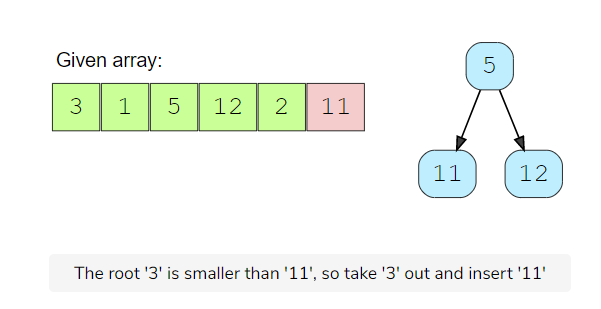
Let’s take Example-1 to go through each step of our algorithm:

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).

Here is the visual representation of our algorithm:



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

Here is what our algorithm will look like:

import java.util.\*;

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 (easy)

**We'll cover the following**

* + - [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#Problem-Statement)
    - [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#Try-it-yourself)
    - [Solution](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#Space-complexity)
  + [An Alternate Approach](https://www.educative.io/courses/grokking-the-coding-interview/gxxPGn8vE8G#An-Alternate-Approach)

### 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.

Note: For a detailed discussion about different approaches to solve this problem, take a look at [Kth Smallest Number](https://www.educative.io/collection/page/5668639101419520/5671464854355968/4817079184130048).

**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](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/) pattern but has two differences:

1. Here we need to find the Kth smallest number, whereas in [Top ‘K’ Numbers](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/) 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)

Here is what our algorithm will look like:

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) 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 (easy)

**We'll cover the following**

* + [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#Problem-Statement)
  + [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#Try-it-yourself)
  + [Solution](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#Solution)
  + [Code](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#Space-complexity)

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

Given an array of points in a 2D2*D* 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}​

This problem follows the [Top ‘K’ Numbers](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/) 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[**#**](https://www.educative.io/courses/grokking-the-coding-interview/3YxNVYwNR5p#Code)

Here is what our algorithm will look like:

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);

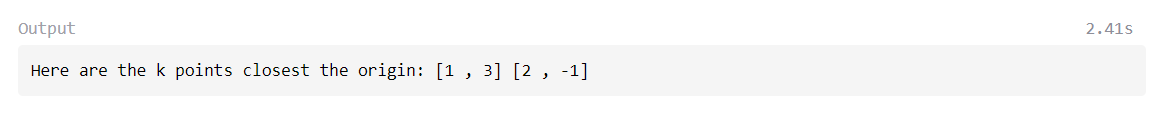
System.out.print("Here are the k points closest the origin: ");

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 are 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 (easy)

**We'll cover the following**

* + - [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#Problem-Statement)
    - [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#Try-it-yourself)
    - [Solution](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/qVZmZJVxPY0#Space-complexity)

### 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](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5696381570252800/). 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)

Here is what our algorithm will look like:

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 (medium)

**We'll cover the following**

* + - [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#Problem-Statement)
    - [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#Try-it-yourself)
    - [Solution](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/B89rvR6XZ3J#Space-complexity)

### 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](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/). 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)

Here is what our algorithm will look like:

## 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 (medium)

**We'll cover the following**

* + - [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#Problem-Statement)
    - [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#Try-it-yourself)
    - [Solution](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/gxZz615BPPG#Space-complexity)

### 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](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5761493274460160/).

We can follow the same approach as discussed in the [Top ‘K’ Frequent Numbers](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5761493274460160/) 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)

Here is what our algorithm will look like:

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);

}

}

----------------------- Python -----------------------------------

from heapq import \*

def sort\_character\_by\_frequency(str):

# find the frequency of each character

charFrequencyMap = {}

for char in str:

charFrequencyMap[char] = charFrequencyMap.get(char, 0) + 1

maxHeap = []

# add all characters to the max heap

for char, frequency in charFrequencyMap.items():

heappush(maxHeap, (-frequency, char))

# build a string, appending the most occurring characters first

sortedString = []

while maxHeap:

frequency, char = heappop(maxHeap)

for \_ in range(-frequency):

sortedString.append(char)

return ''.join(sortedString)

def main():

print("String after sorting characters by frequency: " +

sort\_character\_by\_frequency("Programming"))

print("String after sorting characters by frequency: " +

sort\_character\_by\_frequency("abcbab"))

main()

### 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 (medium)

**We'll cover the following**

* + - [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#Problem-Statement)
    - [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#Try-it-yourself)
    - [Solution](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/B819G5DZBxX#Space-complexity)

### 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](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5696381570252800/).

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)

Here is what our algorithm will look like:

------------------- Java --------------------------------

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));

}

}

---------------------- python ---------------------------------

from heapq import \*

class KthLargestNumberInStream:

minHeap = []

def \_\_init\_\_(self, nums, k):

self.k = k

# add the numbers in the min heap

for num in nums:

self.add(num)

def add(self, num):

# add the new number in the min heap

heappush(self.minHeap, num)

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

if len(self.minHeap) > self.k:

heappop(self.minHeap)

# return the 'Kth largest number

return self.minHeap[0]

def main():

kthLargestNumber = KthLargestNumberInStream([3, 1, 5, 12, 2, 11], 4)

print("4th largest number is: " + str(kthLargestNumber.add(6)))

print("4th largest number is: " + str(kthLargestNumber.add(13)))

print("4th largest number is: " + str(kthLargestNumber.add(4)))

main()

### 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 (medium)

**We'll cover the following**

* + - [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#Problem-Statement-)
    - [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#Try-it-yourself-)
    - [Solution](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#Solution-)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#Code-)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#Space-complexity)
  + [Alternate Solution using Two Pointers](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#Alternate-Solution-using-Two-Pointers)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/N8MJQNYyJPL#Space-complexity)

### 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](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/) 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-)

Here is what our algorithm will look like:

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)*O*(*K*), as we need to put a maximum of 2K2*K* 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 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.

Here is what our algorithm will look like:

import java.util.\*;

class KClosestElements {

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

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

int index = binarySearch(arr, X);

int leftPointer = index;

int rightPointer = index + 1;

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

if (leftPointer >= 0 && rightPointer < arr.length) {

int diff1 = Math.abs(X - arr[leftPointer]);

int diff2 = Math.abs(X - arr[rightPointer]);

if (diff1 <= diff2)

result.add(0, arr[leftPointer--]); // append in the beginning

else

result.add(arr[rightPointer++]); // append at the end

} else if (leftPointer >= 0) {

result.add(0, arr[leftPointer--]);

} else if (rightPointer < arr.length) {

result.add(arr[rightPointer++]);

}

}

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) We need O(logN) for Binary Search and O(K)for finding the ‘K’ closest numbers using the two pointers.

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

If we ignoring the space required for the output list, the algorithm runs in constant space O(1)

# Maximum Distinct Elements (medium)

**We'll cover the following**

* + [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#Problem-Statement)
  + [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#Try-it-yourself)
  + [Solution](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#Solution)
  + [Code](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/gx6oKY8PGYY#Space-complexity)

## 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 appeared 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 appeared 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](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/) pattern, and shares similarities with [Top ‘K’ Frequent Numbers](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5761493274460160/).

We can follow a similar approach as discussed in [Top ‘K’ Frequent Numbers](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5761493274460160/) 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.
5. import java.util.\*;
6. class MaximumDistinctElements {
7. public static int findMaximumDistinctElements(int[] nums, int k) {
8. int distinctElementsCount = 0;
9. if (nums.length <= k)
10. return distinctElementsCount;
11. // find the frequency of each number
12. Map<Integer, Integer> numFrequencyMap = new HashMap<>();
13. for (int i : nums)
14. numFrequencyMap.put(i, numFrequencyMap.getOrDefault(i, 0) + 1);
15. PriorityQueue<Map.Entry<Integer, Integer>> minHeap = new PriorityQueue<Map.Entry<Integer, Integer>>(
16. (e1, e2) -> e1.getValue() - e2.getValue());
17. // insert all numbers with frequency greater than '1' into the min-heap
18. for (Map.Entry<Integer, Integer> entry : numFrequencyMap.entrySet()) {
19. if (entry.getValue() == 1)
20. distinctElementsCount++;
21. else
22. minHeap.add(entry);
23. }
24. // following a greedy approach, try removing the least frequent numbers first from the min-heap
25. while (k > 0 && !minHeap.isEmpty()) {
26. Map.Entry<Integer, Integer> entry = minHeap.poll();
27. // to make an element distinct, we need to remove all of its occurrences except one
28. k -= entry.getValue() - 1;
29. if (k >= 0)
30. distinctElementsCount++;
31. }
32. // if k > 0, this means we have to remove some distinct numbers
33. if (k > 0)
34. distinctElementsCount -= k;
35. return distinctElementsCount;
36. }
37. public static void main(String[] args) {
38. int result = MaximumDistinctElements.findMaximumDistinctElements(new int[] { 7, 3, 5, 8, 5, 3, 3 }, 2);
39. System.out.println("Maximum distinct numbers after removing K numbers: " + result);
40. result = MaximumDistinctElements.findMaximumDistinctElements(new int[] { 3, 5, 12, 11, 12 }, 3);
41. System.out.println("Maximum distinct numbers after removing K numbers: " + result);
42. result = MaximumDistinctElements.findMaximumDistinctElements(new int[] { 1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5 }, 2);
43. System.out.println("Maximum distinct numbers after removing K numbers: " + result);
44. }
45. }

### 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 (medium)

**We'll cover the following**

* + - [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#Problem-Statement)
    - [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#Try-it-yourself)
    - [Solution](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#Space-complexity)
  + [Alternate Solution](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#Alternate-Solution)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/qVljv3Plr67#Space-complexity)

### 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](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/) pattern, and shares similarities with [Kth Smallest Number](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5696381570252800/).

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

Here is what our algorithm will look like:

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)*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)*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)

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)

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

# Rearrange String (hard)

**We'll cover the following**

* + - [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#Problem-Statement)
    - [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#Try-it-yourself)
    - [Solution](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/xV7wx4o8ymB#Space-complexity)

### 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](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/) 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)

Here is what our algorithm will look like:

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"));

}

}

Output

2.46s

Rearranged string: papap

Rearranged string: rgmrgmPiano

Rearranged string:

### 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**.

# Solution Review: Problem Challenge 1

**We'll cover the following**

* + - [Rearrange String K Distance Apart (hard)](https://www.educative.io/courses/grokking-the-coding-interview/qA7n6820GjG#Rearrange-String-K-Distance-Apart-(hard))
    - [Solution](https://www.educative.io/courses/grokking-the-coding-interview/qA7n6820GjG#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/qA7n6820GjG#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/qA7n6820GjG#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/qA7n6820GjG#Space-complexity)

### Rearrange String K Distance Apart (hard) [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qA7n6820GjG#Rearrange-String-K-Distance-Apart-(hard))

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.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qA7n6820GjG#Solution)

This problem follows the [Top ‘K’ Numbers](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/) pattern and is quite similar to [Rearrange String](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5724822843686912/). The only difference is that in the ‘Rearrange String’ the same characters need not be adjacent i.e., they should be at least ‘2’ distance apart (in other words, there should be at least one character between two same characters), while in the current problem, the same characters should be ‘K’ distance apart.

Following a similar approach, since we were inserting a character back in the heap in the next iteration, in this problem, we will re-insert the character after ‘K’ iterations. We can keep track of previous characters in a queue to insert them back in the heap after ‘K’ iterations.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qA7n6820GjG#Code)

Here is what our algorithm will look like:

import java.util.\*;

class RearrangeStringKDistanceApart {

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

if (k <= 1)

return 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());

Queue<Map.Entry<Character, Integer>> queue = new LinkedList<>();

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

while (!maxHeap.isEmpty()) {

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

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

resultString.append(currentEntry.getKey());

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

queue.offer(currentEntry);

if (queue.size() == k) {

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

if (entry.getValue() > 0)

maxHeap.add(entry);

}

}

// 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("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));

}

}

### Time complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/qA7n6820GjG#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/qA7n6820GjG#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.

# Solution Review: Problem Challenge 2

**We'll cover the following**

* + - [Scheduling Tasks (hard)](https://www.educative.io/courses/grokking-the-coding-interview/B1gBkopEBzk#Scheduling-Tasks-(hard))
    - [Solution](https://www.educative.io/courses/grokking-the-coding-interview/B1gBkopEBzk#Solution)
    - [Code](https://www.educative.io/courses/grokking-the-coding-interview/B1gBkopEBzk#Code)
    - [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/B1gBkopEBzk#Time-complexity)
    - [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/B1gBkopEBzk#Space-complexity)

### Scheduling Tasks (hard)[**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1gBkopEBzk#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

### Solution[**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1gBkopEBzk#Solution)

This problem follows the [Top ‘K’ Elements](https://www.educative.io/pageeditor/5671464854355968/5728885882748928) pattern and is quite similar to [Rearrange String K Distance Apart](https://www.educative.io/pageeditor/5671464854355968/5684793748488192). We need to rearrange tasks such that same tasks are ‘K’ distance apart.

Following a similar approach, we will use a **Max Heap** to execute the highest frequency task first. After executing a task we decrease its frequency and put it in a waiting list. In each iteration, we will try to execute as many as k+1 tasks. For the next iteration, we will put all the waiting tasks back in the **Max Heap**. If, for any iteration, we are not able to execute k+1 tasks, the CPU has to remain idle for the remaining time in the next iteration.

### Code[**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1gBkopEBzk#Code)

Here is what our algorithm will look like:

import java.util.\*;

class TaskScheduler {

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

int intervalCount = 0;

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

for (char chr : tasks)

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

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

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

// add all tasks to the max heap

maxHeap.addAll(taskFrequencyMap.entrySet());

while (!maxHeap.isEmpty()) {

List<Map.Entry<Character, Integer>> waitList = new ArrayList<>();

int n = k + 1; // try to execute as many as 'k+1' tasks from the max-heap

for (; n > 0 && !maxHeap.isEmpty(); n--) {

intervalCount++;

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

if (currentEntry.getValue() > 1) {

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

waitList.add(currentEntry);

}

}

maxHeap.addAll(waitList); // put all the waiting list back on the heap

if (!maxHeap.isEmpty())

intervalCount += n; // we'll be having 'n' idle intervals for the next iteration

}

return intervalCount;

}

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));

}

}

from heapq import \*

def schedule\_tasks(tasks, k):

intervalCount = 0

taskFrequencyMap = {}

for char in tasks:

taskFrequencyMap[char] = taskFrequencyMap.get(char, 0) + 1

maxHeap = []

# add all tasks to the max heap

for char, frequency in taskFrequencyMap.items():

heappush(maxHeap, (-frequency, char))

while maxHeap:

waitList = []

n = k + 1 # try to execute as many as 'k+1' tasks from the max-heap

while n > 0 and maxHeap:

intervalCount += 1

frequency, char = heappop(maxHeap)

if -frequency > 1:

# decrement the frequency and add to the waitList

waitList.append((frequency+1, char))

n -= 1

# put all the waiting list back on the heap

for frequency, char in waitList:

heappush(maxHeap, (frequency, char))

if maxHeap:

intervalCount += n # we'll be having 'n' idle intervals for the next iteration

return intervalCount

def main():

print("Minimum intervals needed to execute all tasks: " +

str(schedule\_tasks(['a', 'a', 'a', 'b', 'c', 'c'], 2)))

print("Minimum intervals needed to execute all tasks: " +

str(schedule\_tasks(['a', 'b', 'a'], 3)))

main()

### Time complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1gBkopEBzk#Time-complexity)

The time complexity of the above algorithm is O(N\*logN) where ‘N’ is the number of tasks. Our while loop will iterate once for each occurrence of the task in the input (i.e. ‘N’) and in each iteration we will remove a task from the heap which will take O(logN) time. Hence the overall time complexity of our algorithm is O(N\*logN)

### Space complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1gBkopEBzk#Space-complexity)

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

# Problem Challenge 3

**We'll cover the following**

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

### Frequency Stack (hard)[**#**](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

### Solution[**#**](https://www.educative.io/courses/grokking-the-coding-interview/R8o6vV83DLY#Solution)

This problem follows the [Top ‘K’ Elements](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5728885882748928/) pattern, and shares similarities with [Top ‘K’ Frequent Numbers](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5761493274460160/).

We can use a **Max Heap** to store the numbers. Instead of comparing the numbers we will compare their frequencies so that the root of the heap is always the most frequently occurring number. There are two issues that need to be resolved though:

1. How can we keep track of the frequencies of numbers in the heap? When we are pushing a new number to the **Max Heap**, we don’t know how many times the number has already appeared in the **Max Heap**. To resolve this, we will maintain a **HashMap** to store the current frequency of each number. Thus whenever we push a new number in the heap, we will increment its frequency in the HashMap and when we pop, we will decrement its frequency.
2. If two numbers have the same frequency, we will need to return the number which was pushed later while popping. To resolve this, we need to attach a sequence number to every number to know which number came first.

In short, we will keep three things with every number that we push to the heap:

    1. number // value of the number  
    2. frequency // current frequency of the number when it was pushed to the heap  
    3. sequenceNumber // a sequence number, to know what number came first

### Code[**#**](https://www.educative.io/courses/grokking-the-coding-interview/R8o6vV83DLY#Code)

Here is what our algorithm will look like:

------------------------- Java ---------------------------------------

import java.util.\*;

class Element {

int number;

int frequency;

int sequenceNumber;

public Element(int number, int frequency, int sequenceNumber) {

this.number = number;

this.frequency = frequency;

this.sequenceNumber = sequenceNumber;

}

}

class ElementComparator implements Comparator<Element> {

public int compare(Element e1, Element e2) {

if (e1.frequency != e2.frequency)

return e2.frequency - e1.frequency;

// if both elements have same frequency, return the one that was pushed later

return e2.sequenceNumber - e1.sequenceNumber;

}

}

class FrequencyStack {

int sequenceNumber = 0;

PriorityQueue<Element> maxHeap = new PriorityQueue<Element>(new ElementComparator());

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

public void push(int num) {

frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);

maxHeap.offer(new Element(num, frequencyMap.get(num), sequenceNumber++));

}

public int pop() {

int num = maxHeap.poll().number;

// decrement the frequency or remove if this is the last number

if (frequencyMap.get(num) > 1)

frequencyMap.put(num, frequencyMap.get(num) - 1);

else

frequencyMap.remove(num);

return num;

}

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());

}

}

------------------------------- python --------------------------

from heapq import \*

class Element:

def \_\_init\_\_(self, number, frequency, sequenceNumber):

self.number = number

self.frequency = frequency

self.sequenceNumber = sequenceNumber

def \_\_lt\_\_(self, other):

# higher frequency wins

if self.frequency != other.frequency:

return self.frequency > other.frequency

# if both elements have same frequency, return the element that was pushed later

return self.sequenceNumber > other.sequenceNumber

class FrequencyStack:

sequenceNumber = 0

maxHeap = []

frequencyMap = {}

def push(self, num):

self.frequencyMap[num] = self.frequencyMap.get(num, 0) + 1

heappush(self.maxHeap, Element(

num, self.frequencyMap[num], self.sequenceNumber))

self.sequenceNumber += 1

def pop(self):

num = heappop(self.maxHeap).number

# decrement the frequency or remove if this is the last number

if self.frequencyMap[num] > 1:

self.frequencyMap[num] -= 1

else:

del self.frequencyMap[num]

return num

def main():

frequencyStack = FrequencyStack()

frequencyStack.push(1)

frequencyStack.push(2)

frequencyStack.push(3)

frequencyStack.push(2)

frequencyStack.push(1)

frequencyStack.push(2)

frequencyStack.push(5)

print(frequencyStack.pop())

print(frequencyStack.pop())

print(frequencyStack.pop())

main()

### Time complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/R8o6vV83DLY#Time-complexity)

The time complexity of push() and pop() is O(logN) where ‘N’ is the current number of elements in the heap.

### Space complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/R8o6vV83DLY#Space-complexity)

We will need O(N) space for the heap and the map, so the overall space complexity of the algorithm is O(N)