Heap

# 311. Implement a Maxheap/MinHeap using arrays and recursion.

Given an array of N elements. The task is to build a Binary Heap from the given array. The heap can be either Max Heap or Min Heap.  
**Example**: 

**Input**: arr[] = {4, 10, 3, 5, 1}

**Output**: Corresponding Max-Heap:

10

/ \

5 3

/ \

4 1

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

**Output**: Corresponding Max-Heap:

17

/ \

15 13

/ \ / \

9 6 5 10

/ \ / \

4 8 3 1

## Solution:

Suppose the given input elements are: 4, 10, 3, 5, 1.  
The corresponding complete binary tree for this array of elements [4, 10, 3, 5, 1] will be: 

4

/ \

10 3

/ \

5 1

**Note**:

Root is at index 0 in array.

Left child of i-th node is at (2\*i + 1)th index.

Right child of i-th node is at (2\*i + 2)th index.

Parent of i-th node is at (i-1)/2 index.

**Simple Approach**: Suppose, we need to build a Max-Heap from the above-given array elements. It can be clearly seen that the above complete binary tree formed does not follow the Heap property. So, the idea is to heapify the complete binary tree formed from the array in reverse level order following a top-down approach.  
That is first heapify, the last node in level order traversal of the tree, then heapify the second last node and so on.   
**Time Complexity:** Heapify a single node takes O(log N) time complexity where N is the total number of Nodes. Therefore, building the entire Heap will take N heapify operations and the total time complexity will be **O(N\*logN)**.  
In reality, building a heap takes O(n) time depending on the implementation which can be seen [here](http://www.cs.umd.edu/~meesh/351/mount/lectures/lect14-heapsort-analysis-part.pdf).  
**Optimized Approach**: The above approach can be optimized by observing the fact that the leaf nodes need not to be *heapified*as they already follow the heap property. Also, the array representation of the complete binary tree contains the level order traversal of the tree.  
So the idea is to find the position of the last non-leaf node and perform the **heapify**operation of each non-leaf node in reverse level order. 

**Last non-leaf node** = parent of last-node.

or, Last non-leaf node = parent of node at (n-1)th index.

or, Last non-leaf node = Node at index ((n-1) - 1)/2.

= (n/2) - 1.

**Illustration**: 

Array = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}

Corresponding Complete Binary Tree is:

1

/ \

3 5

/ \ / \

4 6 13 10

/ \ / \

9 8 15 17

***The task to build a Max-Heap from above array***.

Total Nodes = 11.

Last Non-leaf node index = (11/2) - 1 = 4.

Therefore, last non-leaf node = 6.

To build the heap, heapify only the nodes:

[1, 3, 5, 4, 6] in reverse order.

**Heapify 6**: Swap 6 and 17.

1

/ \

3 5

/ \ / \

4 17 13 10

/ \ / \

9 8 15 6

**Heapify 4**: Swap 4 and 9.

1

/ \

3 5

/ \ / \

9 17 13 10

/ \ / \

4 8 15 6

**Heapify 5**: Swap 13 and 5.

1

/ \

3 13

/ \ / \

9 17 5 10

/ \ / \

4 8 15 6

**Heapify 3**: First Swap 3 and 17, again swap 3 and 15.

1

/ \

17 13

/ \ / \

9 15 5 10

/ \ / \

4 8 3 6

**Heapify 1**: First Swap 1 and 17, again swap 1 and 15,

finally swap 1 and 6.

17

/ \

15 13

/ \ / \

9 6 5 10

/ \ / \

4 8 3 1

**Implementation**:

// C++ program for building Heap from Array

#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is

// an index in arr[]. N is size of heap

void heapify(int arr[], int n, int i)

{

int largest = i; // Initialize largest as root

int l = 2 \* i + 1; // left = 2\*i + 1

int r = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

if (l < n && arr[l] > arr[largest])

largest = l;

// If right child is larger than largest so far

if (r < n && arr[r] > arr[largest])

largest = r;

// If largest is not root

if (largest != i) {

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

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// Function to build a Max-Heap from the given array

void buildHeap(int arr[], int n)

{

// Index of last non-leaf node

int startIdx = (n / 2) - 1;

// Perform reverse level order traversal

// from last non-leaf node and heapify

// each node

for (int i = startIdx; i >= 0; i--) {

heapify(arr, n, i);

}

}

// A utility function to print the array

// representation of Heap

void printHeap(int arr[], int n)

{

cout << "Array representation of Heap is:\n";

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

cout << arr[i] << " ";

cout << "\n";

}

// Driver Code

int main()

{

// Binary Tree Representation

// of input array

// 1

// / \

// 3 5

// / \ / \

// 4 6 13 10

// / \ / \

// 9 8 15 17

int arr[] = { 1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17 };

int n = sizeof(arr) / sizeof(arr[0]);

buildHeap(arr, n);

printHeap(arr, n);

// Final Heap:

// 17

// / \

// 15 13

// / \ / \

// 9 6 5 10

// / \ / \

// 4 8 3 1

return 0;

}

**Output:**

Array representation of Heap is:

17 15 13 9 6 5 10 4 8 3 1

# 312. Sort an Array using heap. (HeapSort)

Given an array of size N. The task is to sort the array elements by completing functions **heapify**() and **buildHeap**() which are used to implement Heap Sort.

**Example 1:**

**Input:**

N = 5

arr[] = {4,1,3,9,7}

**Output:**

1 3 4 7 9

**Explanation:**

After sorting elements

using heap sort, elements will be

in order as 1,3,4,7,9.

**Example 2:**

**Input:**

N = 10

arr[] = {10,9,8,7,6,5,4,3,2,1}

**Output:**

1 2 3 4 5 6 7 8 9 10

**Explanation:**

After sorting elements

using heap sort, elements will be

in order as 1, 2,3,4,5,6,7,8,9,10.

**Your Task** **:**  
You don't have to read input or print anything. Your task is to complete the functions **heapify()**, **buildheap()** and **heapSort()**where heapSort() and buildheap() takes the array and it's size as input and heapify() takes the array, it's size and an index i as input. Complete and use these functions to sort the array using heap sort algorithm.  
**Note:**You don't have to return the sorted list. You need to sort the array "arr" in place.

**Expected Time Complexity:** O(N \* Log(N)).  
**Expected Auxiliary Space:** O(1).

**Constraints:**  
1 ≤ N ≤ 106  
1 ≤ arr[i] ≤ 106

## Solution:

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

**What is**[**Binary Heap**](https://www.geeksforgeeks.org/binary-heap/)**?**   
Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source [Wikipedia](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees))  
A [Binary Heap](https://www.geeksforgeeks.org/binary-heap/) is a Complete Binary Tree where items are stored in a special order such that the value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

**Why array based representation for Binary Heap?**   
Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-efficient. If the parent node is stored at index I, the left child can be calculated by 2 \* I + 1 and the right child by 2 \* I + 2 (assuming the indexing starts at 0).

**How to “heapify” a tree?**

The process of reshaping a binary tree into a Heap data structure is known as ‘heapify’. A binary tree is a tree data structure that has two child nodes at max. If a node’s children nodes are ‘heapified’, then only ‘heapify’ process can be applied over that node. A heap should always be a complete binary tree.

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called ‘heapify’ on all the non-leaf elements of the heap. i.e. ‘heapify’ uses recursion.

**Algorithm for “heapify”:**

heapify(array)

Root = array[0]

Largest = largest( array[0] , array [2 \* 0 + 1]. array[2 \* 0 + 2])

if(Root != Largest)

Swap(Root, Largest)

**Example of “heapify”:**

30(0)

/ \

70(1) 50(2)

Child (**70(1)**) is greater than the parent (**30(0**))

Swap Child (**70(1)**) with the parent (**30(0)**)

70(0)

/ \

30(1) 50(2)

**Heap Sort Algorithm for sorting in increasing order:**   
**1.** Build a max heap from the input data.   
**2.** At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.   
**3.** Repeat step 2 while the size of the heap is greater than 1.

**How to build the heap?**   
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.  
Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

The numbers in bracket represent the indices in the array

representation of data.

Applying heapify procedure to index 1:

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

Applying heapify procedure to index 0:

10(0)

/ \

5(1) 3(2)

/ \

4(3) 1(4)

The heapify procedure calls itself recursively to build heap

in top down manner.

// C++ program for implementation of Heap Sort

#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is

// an index in arr[]. n is size of heap

void heapify(int arr[], int n, int i)

{

int largest = i; // Initialize largest as root

int l = 2 \* i + 1; // left = 2\*i + 1

int r = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

if (l < n && arr[l] > arr[largest])

largest = l;

// If right child is larger than largest so far

if (r < n && arr[r] > arr[largest])

largest = r;

// If largest is not root

if (largest != i) {

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

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// main function to do heap sort

void heapSort(int arr[], int n)

{

// Build heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// One by one extract an element from heap

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

// Move current root to end

swap(arr[0], arr[i]);

// call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

/\* A utility function to print array of size n \*/

void printArray(int arr[], int n)

{

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

cout << arr[i] << " ";

cout << "\n";

}

// Driver code

int main()

{

int arr[] = { 12, 11, 13, 5, 6, 7 };

int n = sizeof(arr) / sizeof(arr[0]);

heapSort(arr, n);

cout << "Sorted array is \n";

printArray(arr, n);

}

**Output**

Sorted array is

5 6 7 11 12 13

**Notes:**   
Heap sort is an in-place algorithm.   
Its typical implementation is not stable, but can be made stable (See [this](https://www.geeksforgeeks.org/stability-in-sorting-algorithms/))

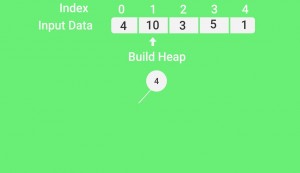
**Time Complexity:**Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and the overall time complexity of Heap Sort is O(nLogn).

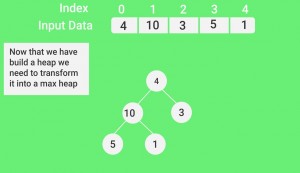
Advantages of heapsort –

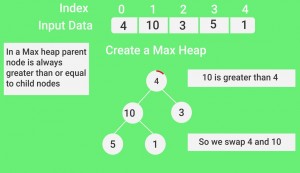
* **Efficiency –** The time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. This sorting algorithm is very efficient.
* **Memory Usage –**Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
* **Simplicity –** It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion

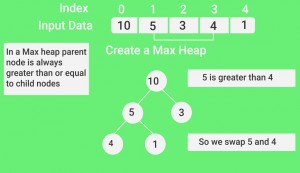
**Applications of HeapSort**   
**1.** [Sort a nearly sorted (or K sorted) array](https://www.geeksforgeeks.org/nearly-sorted-algorithm/)   
**2.**[k largest(or smallest) elements in an array](https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/)   
Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

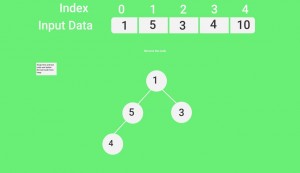
**Snapshots:** 

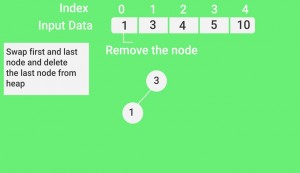












# 313. Maximum of all subarrays of size k.

Given an array arr[] of size N and an integer K. Find the maximum for each and every contiguous subarray of size K.

**Example 1:**

**Input:**

N = 9, K = 3

arr[] = 1 2 3 1 4 5 2 3 6

**Output:**

3 3 4 5 5 5 6

**Explanation:**

1st contiguous subarray = {1 2 3} Max = 3

2nd contiguous subarray = {2 3 1} Max = 3

3rd contiguous subarray = {3 1 4} Max = 4

4th contiguous subarray = {1 4 5} Max = 5

5th contiguous subarray = {4 5 2} Max = 5

6th contiguous subarray = {5 2 3} Max = 5

7th contiguous subarray = {2 3 6} Max = 6

**Example 2:**

**Input:**

N = 10, K = 4

arr[] = 8 5 10 7 9 4 15 12 90 13

**Output:**

10 10 10 15 15 90 90

**Explanation:**

1st contiguous subarray = {8 5 10 7}, Max = 10

2nd contiguous subarray = {5 10 7 9}, Max = 10

3rd contiguous subarray = {10 7 9 4}, Max = 10

4th contiguous subarray = {7 9 4 15}, Max = 15

5th contiguous subarray = {9 4 15 12},

Max = 15

6th contiguous subarray = {4 15 12 90},

Max = 90

7th contiguous subarray = {15 12 90 13},

Max = 90

**Your Task:**  
You dont need to read input or print anything. Complete the function **max\_of\_subarrays()** which takes the array, N and K as input parameters and returns a list of integers denoting the maximum of every contiguous subarray of size K.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(k)

**Constraints:**  
1 ≤ N ≤ 107  
1 ≤ K ≤ N  
0 ≤ arr[i] ≤ 107

## Solution:

**Method 1:** This is a simple method to solve the above problem.

**Approach:**   
The idea is very basic run a nested loop, the outer loop which will mark the starting point of the subarray of length k, the inner loop will run from the starting index to index+k, k elements from starting index and print the maximum element among these k elements.

**Algorithm:**

1. Create a nested loop, the outer loop from starting index to n – k th elements. The inner loop will run for k iterations.
2. Create a variable to store the maximum of k elements traversed by the inner loop.
3. Find the maximum of k elements traversed by the inner loop.
4. Print the maximum element in every iteration of outer loop

**Implementation:**

* C++
* C
* Java
* Python3
* C#
* PHP
* Javascript

|  |
| --- |
| // C++ Program to find the maximum for  // each and every contiguous subarray of size k.  #include <bits/stdc++.h>  using namespace std;    // Method to find the maximum for each  // and every contiguous subarray of size k.  void printKMax(int arr[], int n, int k)  {      int j, max;        for (int i = 0; i <= n - k; i++)      {          max = arr[i];            for (j = 1; j < k; j++)          {              if (arr[i + j] > max)                  max = arr[i + j];          }          cout << max << " ";      }  }    // Driver code  int main()  {      int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };      int n = sizeof(arr) / sizeof(arr[0]);      int k = 3;      printKMax(arr, n, k);      return 0;  }    // This code is contributed by rathbhupendra |

**Output**

3 4 5 6 7 8 9 10

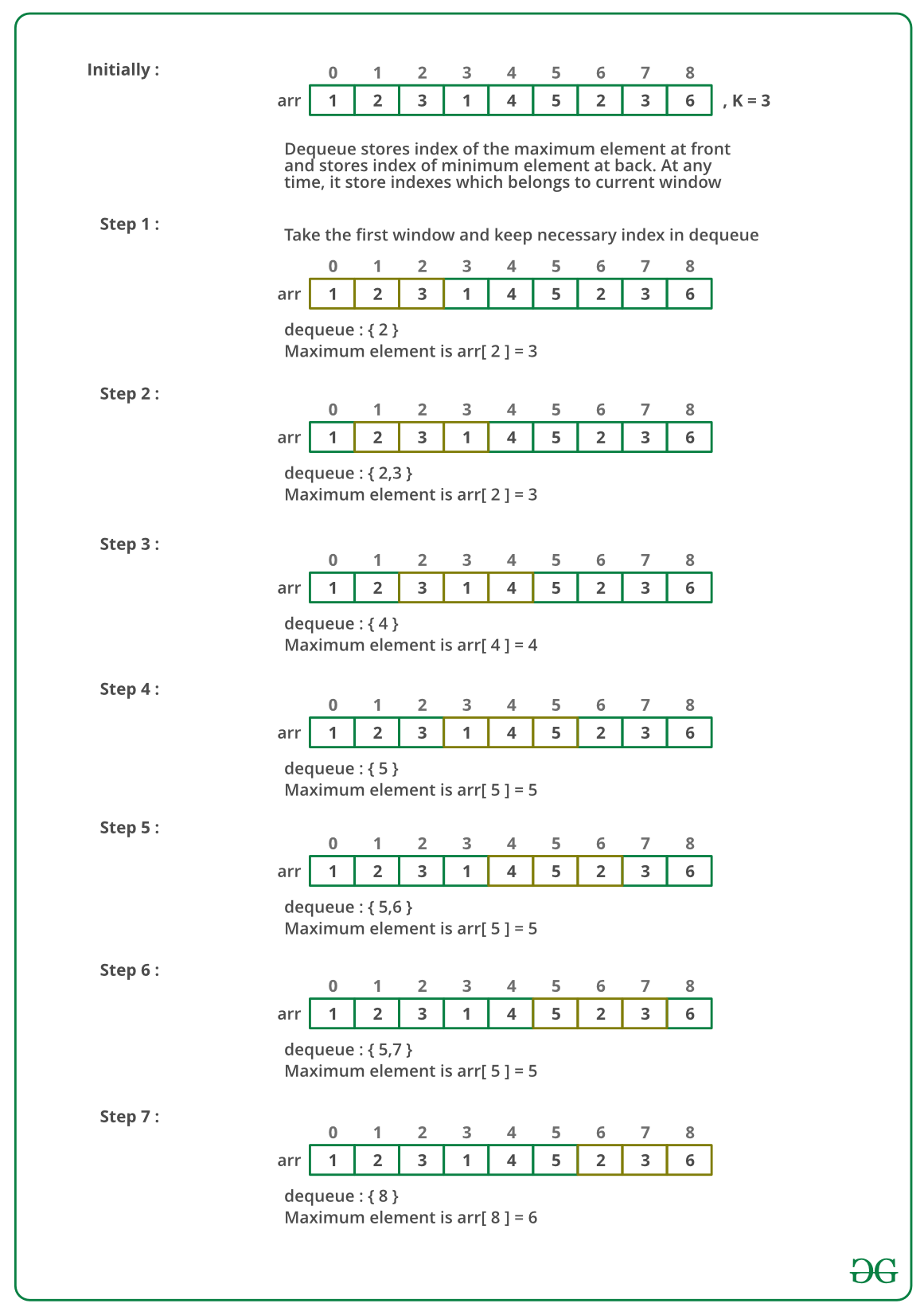
**Complexity Analysis:**

* **Time Complexity:**O(N \* K).   
  The outer loop runs n-k+1 times and the inner loop runs k times for every iteration of outer loop. So time complexity is O((n-k+1)\*k) which can also be written as **O(N \* K)**.
* **Space Complexity:**O(1).   
  No extra space is required.

**Method 2:**This method uses Deque to solve the above problem

**Approach:**   
Create a [Deque](https://www.geeksforgeeks.org/deque-set-1-introduction-applications/), *Qi*of capacity k, that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on right side of it in current window. Process all array elements one by one and maintain *Qi*to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the *Qi*is the largest and element at rear/back of *Qi*is the smallest of current window. Thanks to [Aashish](https://www.geeksforgeeks.org/maximum-of-all-subarrays-of-size-k/#comment-10874)for suggesting this method.

**Dry run of the above approach:**



**Algorithm:**

1. Create a deque to store k elements.
2. Run a loop and insert first k elements in the deque. Before inserting the element, check if the element at the back of the queue is smaller than the current element, if it is so remove the element from the back of the deque, until all elements left in the deque are greater than the current element. Then insert the current element, at the back of the deque.
3. Now, run a loop from k to end of the array.
4. Print the front element of the deque.
5. Remove the element from the front of the queue if they are out of the current window.
6. Insert the next element in the deque. Before inserting the element, check if the element at the back of the queue is smaller than the current element, if it is so remove the element from the back of the deque, until all elements left in the deque are greater than the current element. Then insert the current element, at the back of the deque.
7. Print the maximum element of the last window.

**Implementation:**

// CPP program for the above approach

#include <bits/stdc++.h>

using namespace std;

// A Dequeue (Double ended queue) based

// method for printing maximum element of

// all subarrays of size k

void printKMax(int arr[], int n, int k)

{

// Create a Double Ended Queue,

// Qi that will store indexes

// of array elements

// The queue will store indexes

// of useful elements in every

// window and it will

// maintain decreasing order of

// values from front to rear in Qi, i.e.,

// arr[Qi.front[]] to arr[Qi.rear()]

// are sorted in decreasing order

std::deque<int> Qi(k);

/\* Process first k (or first window)

elements of array \*/

int i;

for (i = 0; i < k; ++i)

{

// For every element, the previous

// smaller elements are useless so

// remove them from Qi

while ((!Qi.empty()) && arr[i] >=

arr[Qi.back()])

// Remove from rear

Qi.pop\_back();

// Add new element at rear of queue

Qi.push\_back(i);

}

// Process rest of the elements,

// i.e., from arr[k] to arr[n-1]

for (; i < n; ++i)

{

// The element at the front of

// the queue is the largest element of

// previous window, so print it

cout << arr[Qi.front()] << " ";

// Remove the elements which

// are out of this window

while ((!Qi.empty()) && Qi.front() <=

i - k)

// Remove from front of queue

Qi.pop\_front();

// Remove all elements

// smaller than the currently

// being added element (remove

// useless elements)

while ((!Qi.empty()) && arr[i] >=

arr[Qi.back()])

Qi.pop\_back();

// Add current element at the rear of Qi

Qi.push\_back(i);

}

// Print the maximum element

// of last window

cout << arr[Qi.front()];

}

// Driver code

int main()

{

int arr[] = { 12, 1, 78, 90, 57, 89, 56 };

int n = sizeof(arr) / sizeof(arr[0]);

int k = 3;

printKMax(arr, n, k);

return 0;

}

**Output**

78 90 90 90 89

**Complexity Analysis:**

* **Time Complexity:** O(n).   
  It seems more than O(n) at first look. It can be observed that every element of array is added and removed at most once. So there are total 2n operations.
* **Auxiliary Space:** O(k).   
  Elements stored in the dequeue take O(k) space.

**Approach 3:**Using Dynamic Programming:

* Firstly, divide the entire array into blocks of k elements such that each block contains k elements of the array(not always for the last block).
* Maintain two dp arrays namely, left and right.
* **left[i]** is the maximum of all elements that are to the left of current element(including current element) in the current block(block in which current element is present).
* Similarly, **right[i]** is the maximum of all elements that are to the right of current element(including current element) in the current block(block in which current element is present).
* Finally, when calculating the maximum element in any subarray of length k, we calculate the maximum of right[l] and left[r]  
  where **l = starting index of current sub array, r = ending index of current sub array**

Below is the implementation of above approach,

// C++ program to find the maximum for each

// and every contiguous subarray of size K

#include <bits/stdc++.h>

using namespace std;

// Function to find the maximum for each

// and every contiguous subarray of size k

void printKMax(int a[], int n, int k)

{

// If k = 1, print all elements

if (k == 1) {

for (int i = 0; i < n; i += 1)

cout << a[i] << " ";

return;

}

//left[i] stores the maximum value to left of i in the current block

//right[i] stores the maximum value to the right of i in the current block

int left[n],right[n];

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

//if the element is starting element of that block

if(i%k == 0) left[i] = a[i];

else left[i] = max(left[i-1],a[i]);

//if the element is ending element of that block

if((n-i)%k == 0 || i==0) right[n-1-i] = a[n-1-i];

else right[n-1-i] = max(right[n-i],a[n-1-i]);

}

for(int i=0,j=k-1; j<n; i++,j++)

cout << max(left[j],right[i]) << ' ';

}

// Driver Code

int main()

{

int a[] = { 1, 2, 3, 4, 5,

6, 7, 8, 9, 10 };

int n = sizeof(a) / sizeof(a[0]);

int K = 3;

printKMax(a, n, K);

return 0;

}

**Output**

3 4 5 6 7 8 9 10

**Time Complexity : O(n)**  
**Auxiliary Space    : O(n)**

# 314. “k” largest element in an array

Given an array **Arr** of **N** positive integers, find **K** **largest elements**from the array.  The output elements should be printed in decreasing order.

**Example 1:**

**Input:**

N = 5, K = 2

Arr[] = {12, 5, 787, 1, 23}

**Output:** 787 23

**Explanation:** 1st largest element in the

array is 787 and second largest is 23.

**Example 2:**

**Input:**

N = 7, K = 3

Arr[] = {1, 23, 12, 9, 30, 2, 50}

**Output:** 50 30 23

**Explanation:** 3 Largest element in the

array are 50, 30 and 23.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **kLargest()** which takes the array of integers **arr,** **n**and**k**as parameters and returns an array of integers denoting the answer. The array should be in decreasing order.

**Expected Time Complexity:** O(N + KlogK)  
**Expected Auxiliary Space:** O(K+(N-K)\*logK)

**Constraints:**  
1 ≤ K ≤ N ≤ 105  
1 ≤ Arr[i] ≤ 106

## Solution:

**Method 1 (Use Bubble k times)**   
Thanks to Shailendra for suggesting this approach.   
1) Modify [Bubble Sort](https://www.geeksforgeeks.org/bubble-sort/) to run the outer loop at most k times.   
2) Print the last k elements of the array obtained in step 1.  
Time Complexity: O(n\*k)

Like Bubble sort, other sorting algorithms like [Selection Sort](http://en.wikipedia.org/wiki/Selection_sort) can also be modified to get the k largest elements.

**Method 2 (Use temporary array)**   
K largest elements from arr[0..n-1]

1) Store the first k elements in a temporary array temp[0..k-1].   
2) Find the smallest element in temp[], let the smallest element be *min*.   
3-a) For each element *x* in arr[k] to arr[n-1]. **O(n-k)**   
If *x*is greater than the min then remove *min*from temp[] and insert *x*.   
3-b)Then, determine the new *min* from temp[]. **O(k)**   
4) Print final k elements of *temp[]*

Time Complexity: O((n-k)\*k). If we want the output sorted then O((n-k)\*k + k\*log(k))  
Thanks to nesamani1822 for suggesting this method.

**Method 3(Use Sorting)**   
1) Sort the elements in descending order in O(n\*log(n))   
2) Print the first k numbers of the sorted array O(k).

Following is the implementation of the above.

// C++ code for k largest elements in an array

#include <bits/stdc++.h>

using namespace std;

void kLargest(int arr[], int n, int k)

{

// Sort the given array arr in reverse

// order.

sort(arr, arr + n, greater<int>());

// Print the first kth largest elements

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

cout << arr[i] << " ";

}

// driver program

int main()

{

int arr[] = { 1, 23, 12, 9, 30, 2, 50 };

int n = sizeof(arr) / sizeof(arr[0]);

int k = 3;

kLargest(arr, n, k);

}

**Output**

50 30 23

**Time complexity:** O(n\*log(n))

**Method 4 (Use Max Heap)**   
1) Build a Max Heap tree in O(n)   
2) Use Extract Max k times to get k maximum elements from the Max Heap O(k\*log(n))

**Time complexity:** O(n + k\*log(n))

**Method 5(Use Order Statistics)**   
1) Use an order statistic algorithm to find the kth largest element. Please [see the topic selection in worst-case linear time](https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-3-worst-case-linear-time/)O(n)   
2) Use [QuickSort](https://www.geeksforgeeks.org/quick-sort/)Partition algorithm to partition around the kth largest number O(n).   
3) Sort the k-1 elements (elements greater than the kth largest element) O(k\*log(k)). This step is needed only if the sorted output is required.

**Time complexity:** O(n) if we don’t need the sorted output, otherwise O(n+k\*log(k))  
Thanks to Shilpi for suggesting the first two approaches.

**Method 6 (Use Min Heap)**   
This method is mainly an optimization of method 1. Instead of using temp[] array, use Min Heap.  
1) Build a Min Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. **O**(k\*log(k))  
2) For each element, after the kth element (arr[k] to arr[n-1]), compare it with root of MH.   
……a) If the element is greater than the root then make it root and call [heapify](https://www.geeksforgeeks.org/binary-heap/)for MH   
……b) Else ignore it.   
// The step 2 is O((n-k)\*log(k))  
3) Finally, MH has k largest elements, and the root of the MH is the kth largest element.  
Time Complexity: O(k\*log(k) + (n-k)\*log(k)) without sorted output. If sorted output is needed then O(k\*log(k) + (n-k)\*log(k) + k\*log(k)) so overall it is O(k\*log(k) + (n-k)\*log(k))

All of the above methods can also be used to find the kth largest (or smallest) element.

#include <iostream>

using namespace std;

// Swap function to interchange

// the value of variables x and y

int swap(int& x, int& y)

{

int temp = x;

x = y;

y = temp;

}

// Min Heap Class

// arr holds reference to an integer

// array size indicate the number of

// elements in Min Heap

class MinHeap {

int size;

int\* arr;

public:

// Constructor to initialize the size and arr

MinHeap(int size, int input[]);

// Min Heapify function, that assumes that

// 2\*i+1 and 2\*i+2 are min heap and fix the

// heap property for i.

void heapify(int i);

// Build the min heap, by calling heapify

// for all non-leaf nodes.

void buildHeap();

};

// Constructor to initialize data

// members and creating mean heap

MinHeap::MinHeap(int size, int input[])

{

// Initializing arr and size

this->size = size;

this->arr = input;

// Building the Min Heap

buildHeap();

}

// Min Heapify function, that assumes

// 2\*i+1 and 2\*i+2 are min heap and

// fix min heap property for i

void MinHeap::heapify(int i)

{

// If Leaf Node, Simply return

if (i >= size / 2)

return;

// variable to store the smallest element

// index out of i, 2\*i+1 and 2\*i+2

int smallest;

// Index of left node

int left = 2 \* i + 1;

// Index of right node

int right = 2 \* i + 2;

// Select minimum from left node and

// current node i, and store the minimum

// index in smallest variable

smallest = arr[left] < arr[i] ? left : i;

// If right child exist, compare and

// update the smallest variable

if (right < size)

smallest = arr[right] < arr[smallest]

? right : smallest;

// If Node i violates the min heap

// property, swap current node i with

// smallest to fix the min-heap property

// and recursively call heapify for node smallest.

if (smallest != i) {

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

heapify(smallest);

}

}

// Build Min Heap

void MinHeap::buildHeap()

{

// Calling Heapify for all non leaf nodes

for (int i = size / 2 - 1; i >= 0; i--) {

heapify(i);

}

}

void FirstKelements(int arr[],int size,int k){

// Creating Min Heap for given

// array with only k elements

MinHeap\* m = new MinHeap(k, arr);

// Loop For each element in array

// after the kth element

for (int i = k; i < size; i++) {

// if current element is smaller

// than minimum element, do nothing

// and continue to next element

if (arr[0] > arr[i])

continue;

// Otherwise Change minimum element to

// current element, and call heapify to

// restore the heap property

else {

arr[0] = arr[i];

m->heapify(0);

}

}

// Now min heap contains k maximum

// elements, Iterate and print

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

cout << arr[i] << " ";

}

}

// Driver Program

int main()

{

int arr[] = { 11, 3, 2, 1, 15, 5, 4,

45, 88, 96, 50, 45 };

int size = sizeof(arr) / sizeof(arr[0]);

// Size of Min Heap

int k = 3;

FirstKelements(arr,size,k);

return 0;

}

**Output**

50 88 96

**My Implementation using same above approach:**

vector<int> kLargest(int arr[], int n, int k) {

priority\_queue<int, vector<int>, greater<int>> pq;

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

pq.push(arr[i]);

for(int i=k;i<n;i++){

if(pq.top()<arr[i]){

pq.pop();

pq.push(arr[i]);

}

}

vector<int> res(k);

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

res[i] = pq.top();

pq.pop();

}

return res;

}

**Method 7(Using Quick Sort partitioning algorithm):**

1. Choose a pivot number.
2. if K is lesser than the pivot\_Index then repeat the step.
3. if K == pivot\_Index : Print the array (low to pivot to get K-smallest elements and (n-pivot\_Index) to n fotr K-largest elements)
4. if  K > pivot\_Index : Repeat the steps for right part.

We can improve on the standard quicksort algorithm by using the random() function. Instead of using the pivot element as the last element, we can randomly choose the pivot element. The worst-case time complexity of this version is O(n2) and the average time complexity is O(n).

Following is the implementation of the above algorithm:

#include <bits/stdc++.h>

using namespace std;

//picks up last element between start and end

int findPivot(int a[], int start, int end)

{

// Selecting the pivot element

int pivot = a[end];

// Initially partition-index will be

// at starting

int pIndex = start;

for (int i = start; i < end; i++) {

// If an element is lesser than pivot, swap it.

if (a[i] <= pivot) {

swap(a[i], a[pIndex]);

// Incrementing pIndex for further

// swapping.

pIndex++;

}

}

// Lastly swapping or the

// correct position of pivot

swap(a[pIndex], a[end]);

return pIndex;

}

//THIS PART OF CODE IS CONTRIBUTED BY - rjrachit

//Picks up random pivot element between start and end

int findRandomPivot(int arr[], int start, int end)

{

int n = end - start + 1;

// Selecting the random pivot index

int pivotInd = random()%n;

swap(arr[end],arr[start+pivotInd]);

int pivot = arr[end];

//initialising pivoting point to start index

pivotInd = start;

for (int i = start; i < end; i++) {

// If an element is lesser than pivot, swap it.

if (arr[i] <= pivot) {

swap(arr[i], arr[pivotInd]);

// Incrementing pivotIndex for further

// swapping.

pivotInd++;

}

}

// Lastly swapping or the

// correct position of pivot

swap(arr[pivotInd], arr[end]);

return pivotInd;

}

//THIS PART OF CODE IS CONTRIBUTED BY - rjrachit

void SmallestLargest(int a[], int low, int high, int k,

int n)

{

if (low == high)

return;

else {

int pivotIndex = findRandomPivot(a, low, high);

if (k == pivotIndex) {

cout << k << " smallest elements are : ";

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

cout << a[i] << " ";

cout << endl;

cout << k << " largest elements are : ";

for (int i = (n - pivotIndex); i < n; i++)

cout << a[i] << " ";

}

else if (k < pivotIndex)

SmallestLargest(a, low, pivotIndex - 1, k, n);

else if (k > pivotIndex)

SmallestLargest(a, pivotIndex + 1, high, k, n);

}

}

// Driver Code

int main()

{

int a[] = { 11, 3, 2, 1, 15, 5, 4, 45, 88, 96, 50, 45 };

int n = sizeof(a) / sizeof(a[0]);

int low = 0;

int high = n - 1;

// Lets assume k is 3

int k = 4;

// Function Call

SmallestLargest(a, low, high, k, n);

return 0;

}

**Output**

3 smallest elements are : 3 2 1

3 largest elements are : 96 50 88

# 315. Kth smallest and largest element in an unsorted array

Given an array **arr[]** and an integer **K** where K is smaller than size of array, the task is to find the **Kth smallest** element in the given array. It is given that all array elements are distinct.

**Example 1:**

**Input:**

N = 6

arr[] = 7 10 4 3 20 15

K = 3

**Output :** 7

**Explanation :**

3rd smallest element in the given

array is 7.

**Example 2:**

**Input:**

N = 5

arr[] = 7 10 4 20 15

K = 4

**Output :** 15

**Explanation :**

4th smallest element in the given

array is 15.

**Your Task:**  
You don't have to read input or print anything. Your task is to complete the function **kthSmallest()**which takes the array **arr[]**, integers **l** and **r** denoting the **starting** and **ending** index of the array and an integer **K** as inputand returns the **Kth** smallest element.

**Expected Time Complexity:**O(n)

**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 <= N <= 105  
1 <= arr[i] <= 105  
1 <= K <= N

## Solution:

**Method 1 (Simple Solution)**  
A simple solution is to sort the given array using a O(N log N) sorting algorithm like [Merge Sort](http://geeksquiz.com/merge-sort/), [Heap Sort](http://geeksquiz.com/heap-sort/), etc, and return the element at index k-1 in the sorted array.   
Time Complexity of this solution is O(N Log N)

// Simple C++ program to find k'th smallest element

#include <algorithm>

#include <iostream>

using namespace std;

// Function to return k'th smallest element in a given array

int kthSmallest(int arr[], int n, int k)

{

// Sort the given array

sort(arr, arr + n);

// Return k'th element in the sorted array

return arr[k - 1];

}

// Driver program to test above methods

int main()

{

int arr[] = { 12, 3, 5, 7, 19 };

int n = sizeof(arr) / sizeof(arr[0]), k = 2;

cout << "K'th smallest element is " << kthSmallest(arr, n, k);

return 0;

}

**Output**

K'th smallest element is 5

**Method 2 (using set from C++ STL)**

we can find the kth smallest element in time complexity better than O(N log N). we know the Set in C++ STL is implemented using Binary Search Tree and we also know that the time complexity of all cases(searching , inserting, deleting ) in BST is log (n) in average case and O(n) in worst case .   we are using set because it is mentioned in the question that all the elements in array re distinct.

The following is C++ implementation of above method.

/\* the following code demonstrates how to find kth smallest

element using set from C++ STL \*/

#include <bits/stdc++.h>

using namespace std;

int main()

{

int arr[] = { 12, 3, 5, 7, 19 };

int n = sizeof(arr) / sizeof(arr[0]);

int k = 4;

set<int> s(arr, arr + n);

set<int>::iterator itr

= s.begin(); // s.begin() returns a pointer to first

// element in the set

advance(itr, k - 1); // itr points to kth element in set

cout << \*itr << "\n";

return 0;

}

**Output**

10

**Time Complexity:**  O( log N) in Average Case and O(N) in Worst Case  
**Auxiliary Space:** O(N)

**Method 3 (Using Min Heap – HeapSelect)**   
We can find k’th smallest element in time complexity better than O(N Log N). A simple optimization is to create a [Min Heap](http://geeksquiz.com/binary-heap/)of the given n elements and call extractMin() k times.

The following is C++ implementation of above method.

// A C++ program to find k'th smallest element using min heap

#include <climits>

#include <iostream>

using namespace std;

// Prototype of a utility function to swap two integers

void swap(int\* x, int\* y);

// A class for Min Heap

class MinHeap {

int\* harr; // pointer to array of elements in heap

int capacity; // maximum possible size of min heap

int heap\_size; // Current number of elements in min heap

public:

MinHeap(int a[], int size); // Constructor

void MinHeapify(int i); // To minheapify subtree rooted with index i

int parent(int i) { return (i - 1) / 2; }

int left(int i) { return (2 \* i + 1); }

int right(int i) { return (2 \* i + 2); }

int extractMin(); // extracts root (minimum) element

int getMin() { return harr[0]; } // Returns minimum

};

MinHeap::MinHeap(int a[], int size)

{

heap\_size = size;

harr = a; // store address of array

int i = (heap\_size - 1) / 2;

while (i >= 0) {

MinHeapify(i);

i--;

}

}

// Method to remove minimum element (or root) from min heap

int MinHeap::extractMin()

{

if (heap\_size == 0)

return INT\_MAX;

// Store the minimum vakue.

int root = harr[0];

// If there are more than 1 items, move the last item to root

// and call heapify.

if (heap\_size > 1) {

harr[0] = harr[heap\_size - 1];

MinHeapify(0);

}

heap\_size--;

return root;

}

// A recursive method to heapify a subtree with root at given index

// This method assumes that the subtrees are already heapified

void MinHeap::MinHeapify(int i)

{

int l = left(i);

int r = right(i);

int smallest = i;

if (l < heap\_size && harr[l] < harr[i])

smallest = l;

if (r < heap\_size && harr[r] < harr[smallest])

smallest = r;

if (smallest != i) {

swap(&harr[i], &harr[smallest]);

MinHeapify(smallest);

}

}

// A utility function to swap two elements

void swap(int\* x, int\* y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

// Function to return k'th smallest element in a given array

int kthSmallest(int arr[], int n, int k)

{

// Build a heap of n elements: O(n) time

MinHeap mh(arr, n);

// Do extract min (k-1) times

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

mh.extractMin();

// Return root

return mh.getMin();

}

// Driver program to test above methods

int main()

{

int arr[] = { 12, 3, 5, 7, 19 };

int n = sizeof(arr) / sizeof(arr[0]), k = 2;

cout << "K'th smallest element is " << kthSmallest(arr, n, k);

return 0;

}

**Output**

K'th smallest element is 5

Time complexity of this solution is O(n + kLogn).

**Method 4 (Using Max-Heap)**   
We can also use Max Heap for finding the k’th smallest element. Following is an algorithm.   
1) Build a Max-Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. O(k)  
2) For each element, after the k’th element (arr[k] to arr[n-1]), compare it with root of MH.   
……a) If the element is less than the root then make it root and call heapify for MH   
……b) Else ignore it.   
// The step 2 is O((n-k)\*logk)  
3) Finally, the root of the MH is the kth smallest element.  
Time complexity of this solution is O(k + (n-k)\*Logk)

The following is C++ implementation of the above algorithm

// A C++ program to find k'th smallest element using max heap

#include <climits>

#include <iostream>

using namespace std;

// Prototype of a utility function to swap two integers

void swap(int\* x, int\* y);

// A class for Max Heap

class MaxHeap {

int\* harr; // pointer to array of elements in heap

int capacity; // maximum possible size of max heap

int heap\_size; // Current number of elements in max heap

public:

MaxHeap(int a[], int size); // Constructor

void maxHeapify(int i); // To maxHeapify subtree rooted with index i

int parent(int i) { return (i - 1) / 2; }

int left(int i) { return (2 \* i + 1); }

int right(int i) { return (2 \* i + 2); }

int extractMax(); // extracts root (maximum) element

int getMax() { return harr[0]; } // Returns maximum

// to replace root with new node x and heapify() new root

void replaceMax(int x)

{

harr[0] = x;

maxHeapify(0);

}

};

MaxHeap::MaxHeap(int a[], int size)

{

heap\_size = size;

harr = a; // store address of array

int i = (heap\_size - 1) / 2;

while (i >= 0) {

maxHeapify(i);

i--;

}

}

// Method to remove maximum element (or root) from max heap

int MaxHeap::extractMax()

{

if (heap\_size == 0)

return INT\_MAX;

// Store the maximum vakue.

int root = harr[0];

// If there are more than 1 items, move the last item to root

// and call heapify.

if (heap\_size > 1) {

harr[0] = harr[heap\_size - 1];

maxHeapify(0);

}

heap\_size--;

return root;

}

// A recursive method to heapify a subtree with root at given index

// This method assumes that the subtrees are already heapified

void MaxHeap::maxHeapify(int i)

{

int l = left(i);

int r = right(i);

int largest = i;

if (l < heap\_size && harr[l] > harr[i])

largest = l;

if (r < heap\_size && harr[r] > harr[largest])

largest = r;

if (largest != i) {

swap(&harr[i], &harr[largest]);

maxHeapify(largest);

}

}

// A utility function to swap two elements

void swap(int\* x, int\* y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

// Function to return k'th largest element in a given array

int kthSmallest(int arr[], int n, int k)

{

// Build a heap of first k elements: O(k) time

MaxHeap mh(arr, k);

// Process remaining n-k elements. If current element is

// smaller than root, replace root with current element

for (int i = k; i < n; i++)

if (arr[i] < mh.getMax())

mh.replaceMax(arr[i]);

// Return root

return mh.getMax();

}

// Driver program to test above methods

int main()

{

int arr[] = { 12, 3, 5, 7, 19 };

int n = sizeof(arr) / sizeof(arr[0]), k = 4;

cout << "K'th smallest element is " << kthSmallest(arr, n, k);

return 0;

}

**Output**

K'th smallest element is 12

**Method 5 (QuickSelect)**  
This is an optimization over method 1 if [QuickSort](http://geeksquiz.com/quick-sort/)is used as a sorting algorithm in first step. In QuickSort, we pick a pivot element, then move the pivot element to its correct position and partition the surrounding array. The idea is, not to do complete quicksort, but stop at the point where pivot itself is k’th smallest element. Also, not to recur for both left and right sides of pivot, but recur for one of them according to the position of pivot. The worst case time complexity of this method is O(n2), but it works in O(n) on average.

#include <climits>

#include <iostream>

using namespace std;

int partition(int arr[], int l, int r);

// This function returns k'th smallest element in arr[l..r] using

// QuickSort based method. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT

int kthSmallest(int arr[], int l, int r, int k)

{

// If k is smaller than number of elements in array

if (k > 0 && k <= r - l + 1) {

// Partition the array around last element and get

// position of pivot element in sorted array

int pos = partition(arr, l, r);

// If position is same as k

if (pos - l == k - 1)

return arr[pos];

if (pos - l > k - 1) // If position is more, recur for left subarray

return kthSmallest(arr, l, pos - 1, k);

// Else recur for right subarray

return kthSmallest(arr, pos + 1, r, k - pos + l - 1);

}

// If k is more than number of elements in array

return INT\_MAX;

}

void swap(int\* a, int\* b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Standard partition process of QuickSort(). It considers the last

// element as pivot and moves all smaller element to left of it

// and greater elements to right

int partition(int arr[], int l, int r)

{

int x = arr[r], i = l;

for (int j = l; j <= r - 1; j++) {

if (arr[j] <= x) {

swap(&arr[i], &arr[j]);

i++;

}

}

swap(&arr[i], &arr[r]);

return i;

}

// Driver program to test above methods

int main()

{

int arr[] = { 12, 3, 5, 7, 4, 19, 26 };

int n = sizeof(arr) / sizeof(arr[0]), k = 3;

cout << "K'th smallest element is " << kthSmallest(arr, 0, n - 1, k);

return 0;

}

**Output**

K'th smallest element is 5

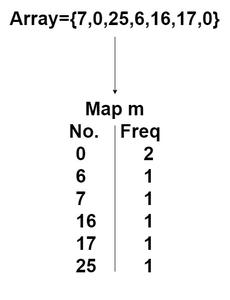
**Method 6 (Map STL)**

A map based STL approach is although very much similar to the quickselect and counting sort algorithm but much easier to implement. We can use an ordered map and map each element with it’s frequency. And as we know that an ordered map would store the data in an sorted manner, we keep on adding the frequency of each element till it does not become greater than or equal to k so that we reach the k’th element from the start i.e. the k’th smallest element.

Eg –

Array=**{7,0,25,6,16,17,0}**

k=**3**



Now in order to get the k’th largest element, we need to add the frequencies till it becomes **greater than or equal to 3.**It is clear from the above that the frequency of 0 + frequency of 6 will become equal to 3 so the third smallest number in the array will be 6.

We can achieve the above using an iterator to traverse the map.

#include <bits/stdc++.h>

using namespace std;

int Kth\_smallest(map<int, int> m, int k)

{

int freq = 0;

for (auto it = m.begin(); it != m.end(); it++) {

freq += (it->second); // adding the frequencies of

// each element

if (freq >= k) // if at any point frequency becomes

// greater than or equal to k then

// return that element

{

return it->first;

}

}

return -1; // returning -1 if k>size of the array which

// is an impossible scenario

}

int main()

{

int n = 5;

int k = 2;

vector<int> arr = { 12, 3, 5, 7, 19 };

map<int, int> m;

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

m[arr[i]] += 1; // mapping every element with it's

// frequency

}

int ans = Kth\_smallest(m, k);

cout << "The " << k << "rd smallest element is " << ans

<< endl;

return 0;

}

**Output**

The 2rd smallest element is 5

There are two more solutions which are better than above discussed ones: One solution is to do randomized version of [quickSelect()](https://www.geeksforgeeks.org/quickselect-algorithm/) and other solution is the worst case linear time algorithm (see the following posts).

In this post method 5 is discussed which is mainly an extension of method 4 (QuickSelect) discussed in the [previous](https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array/)post. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, [rand()](http://www.cplusplus.com/reference/cstdlib/rand/) to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

Following is an implementation of the above Randomized QuickSelect.

// C++ implementation of randomized quickSelect

#include<iostream>

#include<climits>

#include<cstdlib>

using namespace std;

int randomPartition(int arr[], int l, int r);

// This function returns k'th smallest element in arr[l..r] using

// QuickSort based method. ASSUMPTION: ELEMENTS IN ARR[] ARE DISTINCT

int kthSmallest(int arr[], int l, int r, int k)

{

// If k is smaller than number of elements in array

if (k > 0 && k <= r - l + 1)

{

// Partition the array around a random element and

// get position of pivot element in sorted array

int pos = randomPartition(arr, l, r);

// If position is same as k

if (pos-l == k-1)

return arr[pos];

if (pos-l > k-1) // If position is more, recur for left subarray

return kthSmallest(arr, l, pos-1, k);

// Else recur for right subarray

return kthSmallest(arr, pos+1, r, k-pos+l-1);

}

// If k is more than the number of elements in the array

return INT\_MAX;

}

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Standard partition process of QuickSort(). It considers the last

// element as pivot and moves all smaller element to left of it and

// greater elements to right. This function is used by randomPartition()

int partition(int arr[], int l, int r)

{

int x = arr[r], i = l;

for (int j = l; j <= r - 1; j++)

{

if (arr[j] <= x)

{

swap(&arr[i], &arr[j]);

i++;

}

}

swap(&arr[i], &arr[r]);

return i;

}

// Picks a random pivot element between l and r and partitions

// arr[l..r] around the randomly picked element using partition()

int randomPartition(int arr[], int l, int r)

{

int n = r-l+1;

int pivot = rand() % n;

swap(&arr[l + pivot], &arr[r]);

return partition(arr, l, r);

}

// Driver program to test above methods

int main()

{

int arr[] = {12, 3, 5, 7, 4, 19, 26};

int n = sizeof(arr)/sizeof(arr[0]), k = 3;

cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);

return 0;

}

**Output:**

K'th smallest element is 5

**Time Complexity:**  
The worst case time complexity of the above solution is still O(n2). In the worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is O(n), see [CLRS book](http://www.flipkart.com/introduction-algorithms-english-3rd/p/itmdwxyrafdburzg?pid=9788120340077&affid=sandeepgfg) or [MIT video lecture](http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-6-order-statistics-median/) for proof. The assumption in the analysis is, random number generator is equally likely to generate any number in the input range.

**Worst Case Linear Time**

In [previous post](https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/), we discussed an expected linear time algorithm. In this post, a worst-case linear time method is discussed. *The idea in this new method is similar to quickSelect(), we get worst-case linear time by selecting a pivot that divides array in a balanced way (there are not very few elements on one side and many on another side)*. After the array is divided in a balanced way, we apply the same steps as used in quickSelect() to decide whether to go left or right of the pivot.  
Following is complete algorithm.

***kthSmallest(arr[0..n-1], k)******1)****Divide arr[] into ⌈n/5⌉ groups where size of each group is 5 except possibly the last group which may have less than 5 elements.****2)****Sort the above created ⌈n/5⌉ groups and find median of all groups. Create an auxiliary array ‘median[]’ and store medians of all ⌈n/5⌉ groups in this median array.  
// Recursively call this method to find median of median[0..⌈n/5⌉-1]****3)****medOfMed = kthSmallest(median[0..⌈n/5⌉-1], ⌈n/10⌉)****4)****Partition arr[] around medOfMed and obtain its position.   
pos = partition(arr, n, medOfMed)****5)****If pos == k return medOfMed****6)****If pos > k return kthSmallest(arr[l..pos-1], k)****7)****If pos < k return kthSmallest(arr[pos+1..r], k-pos+l-1)*

In above algorithm, last 3 steps are same as algorithm in [previous post](https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/). The first four steps are used to obtain a good point for partitioning the array (to make sure that there are not too many elements either side of pivot).  
Following is the implementation of above algorithm.

// C++ implementation of worst case linear time algorithm

// to find k'th smallest element

#include<iostream>

#include<algorithm>

#include<climits>

using namespace std;

int partition(int arr[], int l, int r, int k);

// A simple function to find median of arr[]. This is called

// only for an array of size 5 in this program.

int findMedian(int arr[], int n)

{

sort(arr, arr+n); // Sort the array

return arr[n/2]; // Return middle element

}

// Returns k'th smallest element in arr[l..r] in worst case

// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT

int kthSmallest(int arr[], int l, int r, int k)

{

// If k is smaller than number of elements in array

if (k > 0 && k <= r - l + 1)

{

int n = r-l+1; // Number of elements in arr[l..r]

// Divide arr[] in groups of size 5, calculate median

// of every group and store it in median[] array.

int i, median[(n+4)/5]; // There will be floor((n+4)/5) groups;

for (i=0; i<n/5; i++)

median[i] = findMedian(arr+l+i\*5, 5);

if (i\*5 < n) //For last group with less than 5 elements

{

median[i] = findMedian(arr+l+i\*5, n%5);

i++;

}

// Find median of all medians using recursive call.

// If median[] has only one element, then no need

// of recursive call

int medOfMed = (i == 1)? median[i-1]:

kthSmallest(median, 0, i-1, i/2);

// Partition the array around a random element and

// get position of pivot element in sorted array

int pos = partition(arr, l, r, medOfMed);

// If position is same as k

if (pos-l == k-1)

return arr[pos];

if (pos-l > k-1) // If position is more, recur for left

return kthSmallest(arr, l, pos-1, k);

// Else recur for right subarray

return kthSmallest(arr, pos+1, r, k-pos+l-1);

}

// If k is more than number of elements in array

return INT\_MAX;

}

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// It searches for x in arr[l..r], and partitions the array

// around x.

int partition(int arr[], int l, int r, int x)

{

// Search for x in arr[l..r] and move it to end

int i;

for (i=l; i<r; i++)

if (arr[i] == x)

break;

swap(&arr[i], &arr[r]);

// Standard partition algorithm

i = l;

for (int j = l; j <= r - 1; j++)

{

if (arr[j] <= x)

{

swap(&arr[i], &arr[j]);

i++;

}

}

swap(&arr[i], &arr[r]);

return i;

}

// Driver program to test above methods

int main()

{

int arr[] = {12, 3, 5, 7, 4, 19, 26};

int n = sizeof(arr)/sizeof(arr[0]), k = 3;

cout << "K'th smallest element is "

<< kthSmallest(arr, 0, n-1, k);

return 0;

}

**Output:** 

K'th smallest element is 5

**Time Complexity:**  
The worst case time complexity of the above algorithm is O(n). Let us analyze all steps.   
The steps 1) and 2) take O(n) time as finding median of an array of size 5 takes O(1) time and there are n/5 arrays of size 5.   
The step 3) takes T(n/5) time. The step 4 is standard partition and takes O(n) time.   
The interesting steps are 6) and 7). At most, one of them is executed. These are recursive steps. What is the worst case size of these recursive calls. The answer is maximum number of elements greater than medOfMed (obtained in step 3) or maximum number of elements smaller than medOfMed.   
*How many elements are greater than medOfMed and how many are smaller?*   
At least half of the medians found in step 2 are greater than or equal to medOfMed. Thus, at least half of the n/5 groups contribute 3 elements that are greater than medOfMed, except for the one group that has fewer than 5 elements. Therefore, the number of elements greater than medOfMed is at least.   
  
Similarly, the number of elements that are less than medOfMed is at least 3n/10 – 6. In the worst case, the function recurs for at most n – (3n/10 – 6) which is 7n/10 + 6 elements.  
Note that 7n/10 + 6 20 20 and that any input of 80 or fewer elements requires O(1) time. We can therefore obtain the recurrence   
  
We show that the running time is linear by substitution. Assume that T(n) cn for some constant c and all n > 80. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

T(n) <= cn/5 + c(7n/10 + 6) + O(n)

<= cn/5 + c + 7cn/10 + 6c + O(n)

<= 9cn/10 + 7c + O(n)

<= cn,

since we can pick c large enough so that c(n/10 – 7) is larger than the function described by the O(n) term for all n > 80. The worst-case running time of is therefore linear (Source: <http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap10.htm> ).  
Note that the above algorithm is linear in worst case, but the constants are very high for this algorithm. Therefore, this algorithm doesn’t work well in practical situations, [randomized quickSelect](https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/) works much better and preferred.

# 316. Merge “K” sorted arrays. [ IMP ]

Given **K** sorted arrays arranged in the form of a matrix of size K\*K. The task is to merge them into one sorted array.  
**Example 1:**

**Input:**

K = 3

arr[][] = {{1,2,3},{4,5,6},{7,8,9}}

**Output:** 1 2 3 4 5 6 7 8 9

**Explanation:**Above test case has 3 sorted

arrays of size 3, 3, 3

arr[][] = [[1, 2, 3],[4, 5, 6],

[7, 8, 9]]

The merged list will be

[1, 2, 3, 4, 5, 6, 7, 8, 9].

**Example 2:**

**Input:**

K = 4

arr[][]={{1,2,3,4}{2,2,3,4},

{5,5,6,6},{7,8,9,9}}

**Output:**

1 2 2 2 3 3 4 4 5 5 6 6 7 8 9 9

**Explanation:** Above test case has 4 sorted

arrays of size 4, 4, 4, 4

arr[][] = [[1, 2, 2, 2], [3, 3, 4, 4],

[5, 5, 6, 6]  [7, 8, 9, 9 ]]

The merged list will be

[1, 2, 2, 2, 3, 3, 4, 4, 5, 5,

6, 6, 7, 8, 9, 9 ].

**Your Task:**  
You do not need to read input or print anything. Your task is to complete **mergeKArrays**() function which takes 2 arguments, an arr[K][K] 2D Matrix containing K sorted arrays and an integer K denoting the number of sorted arrays, as input and returns the merged sorted array ( as a pointer to the merged sorted arrays in **cpp,**as an ArrayList in **java,**and list in **python**)

**Expected Time Complexity:** O(K2\*Log(K))  
**Expected Auxiliary Space:** O(K)

**Constraints**:  
1 <= K <= 100

## Solution:

**Naive Approach:** The very naive method is to create an output array of size n \* k and then copy all the elements into the output array followed by sorting.

* **Algorithm:**
  1. Create a output array of size n \* k.
  2. Traverse the matrix from start to end and insert all the elements in output array.
  3. Sort and print the output array.
* **Implementation:**

// C++ program to merge k sorted arrays of size n each.

#include<bits/stdc++.h>

using namespace std;

#define n 4

// A utility function to print array elements

void printArray(int arr[], int size)

{

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

cout << arr[i] << " ";

}

// This function takes an array of arrays as an argument and

// All arrays are assumed to be sorted. It merges them together

// and prints the final sorted output.

void mergeKArrays(int arr[][n], int a, int output[])

{

int c=0;

//traverse the matrix

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

{

for(int j=0; j<n ;j++)

output[c++]=arr[i][j];

}

//sort the array

sort(output,output + n\*a);

}

// Driver program to test above functions

int main()

{

// Change n at the top to change number of elements

// in an array

int arr[][n] = {{2, 6, 12, 34},

{1, 9, 20, 1000},

{23, 34, 90, 2000}};

int k = sizeof(arr)/sizeof(arr[0]);

int output[n\*k];

mergeKArrays(arr, 3, output);

cout << "Merged array is " << endl;

printArray(output, n\*k);

return 0;

}

**Output:**

Merged array is

1 2 6 9 12 20 23 34 34 90 1000 2000

* **Complexity Analysis:**
  + **Time Complexity :**O(n\*k\*log(n\*k)).   
    since resulting array is of N\*k size.
  + **Space Complexity :**O(n\*k), The output array is of size n\*k.

**Efficient Approach** The process might begin with merging arrays into groups of two. After the first merge, we have k/2 arrays. Again merge arrays in groups, now we have k/4 arrays. This is similar to merge sort. Divide k arrays into two halves containing an equal number of arrays until there are two arrays in a group. This is followed by merging the arrays in a bottom-up manner.

* **Algorithm:**
  1. Create a recursive function which takes k arrays and returns the output array.
  2. In the recursive function, if the value of k is 1 then return the array else if the value of k is 2 then merge the two arrays in linear time and return the array.
  3. If the value of k is greater than 2 then divide the group of k elements into two equal halves and recursively call the function, i.e 0 to k/2 array in one recursive function and k/2 to k array in another recursive function.
  4. Print the output array.
* **Implementation:**

// C++ program to merge k sorted arrays of size n each.

#include<bits/stdc++.h>

using namespace std;

#define n 4

// Merge arr1[0..n1-1] and arr2[0..n2-1] into

// arr3[0..n1+n2-1]

void mergeArrays(int arr1[], int arr2[], int n1,

int n2, int arr3[])

{

int i = 0, j = 0, k = 0;

// Traverse both array

while (i<n1 && j <n2)

{

// Check if current element of first

// array is smaller than current element

// of second array. If yes, store first

// array element and increment first array

// index. Otherwise do same with second array

if (arr1[i] < arr2[j])

arr3[k++] = arr1[i++];

else

arr3[k++] = arr2[j++];

}

// Store remaining elements of first array

while (i < n1)

arr3[k++] = arr1[i++];

// Store remaining elements of second array

while (j < n2)

arr3[k++] = arr2[j++];

}

// A utility function to print array elements

void printArray(int arr[], int size)

{

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

cout << arr[i] << " ";

}

// This function takes an array of arrays as an argument and

// All arrays are assumed to be sorted. It merges them together

// and prints the final sorted output.

void mergeKArrays(int arr[][n],int i,int j, int output[])

{

//if one array is in range

if(i==j)

{

for(int p=0; p < n; p++)

output[p]=arr[i][p];

return;

}

//if only two arrays are left them merge them

if(j-i==1)

{

mergeArrays(arr[i],arr[j],n,n,output);

return;

}

//output arrays

int out1[n\*(((i+j)/2)-i+1)],out2[n\*(j-((i+j)/2))];

//divide the array into halves

mergeKArrays(arr,i,(i+j)/2,out1);

mergeKArrays(arr,(i+j)/2+1,j,out2);

//merge the output array

mergeArrays(out1,out2,n\*(((i+j)/2)-i+1),n\*(j-((i+j)/2)),output);

}

// Driver program to test above functions

int main()

{

// Change n at the top to change number of elements

// in an array

int arr[][n] = {{2, 6, 12, 34},

{1, 9, 20, 1000},

{23, 34, 90, 2000}};

int k = sizeof(arr)/sizeof(arr[0]);

int output[n\*k];

mergeKArrays(arr,0,2, output);

cout << "Merged array is " << endl;

printArray(output, n\*k);

return 0;

}

**Output:**

Merged array is

1 2 6 9 12 20 23 34 34 90 1000 2000

**Complexity Analysis:**

* **Time Complexity:** O( n \* k \* log k).   
  There are log k levels as in each level the k arrays are divided in half and at each level the k arrays are traversed. So time Complexity is O( n \* k ).
* **Space Complexity:**O( n \* k \* log k).   
  In each level O( n\*k ) space is required So Space Complexity is O( n \* k \* log k).

**Alternative Efficient Approach:**The idea is to use [Min Heap](https://www.geeksforgeeks.org/binary-heap/). This MinHeap based solution has the same time complexity which is O(NK log K). But for a [different and particular sized array](https://www.geeksforgeeks.org/merge-k-sorted-arrays-set-2-different-sized-arrays/), this solution works much better. The process must start with creating a MinHeap and inserting the first element of all the k arrays. Remove the root element of Minheap and put it in the output array and insert the next element from the array of removed element. To get the result the step must continue until there is no element in the MinHeap left.

*MinHeap:*A Min-Heap is a complete binary tree in which the value in each internal node is smaller than or equal to the values in the children of that node. Mapping the elements of a heap into an array is trivial: if a node is stored at index k, then its left child is stored at index 2k + 1 and its right child at index 2k + 2.

* **Algorithm:**
  1. Create a min Heap and insert the first element of all k arrays.
  2. Run a loop until the size of MinHeap is greater than zero.
  3. Remove the top element of the MinHeap and print the element.
  4. Now insert the next element from the same array in which the removed element belonged.
  5. If the array doesn’t have any more elements, then replace root with infinite.After replacing the root, heapify the tree.

**Implementation:**

// C++ program to merge k sorted

// arrays of size n each.

#include<bits/stdc++.h>

using namespace std;

#define n 4

// A min-heap node

struct MinHeapNode

{

// The element to be stored

int element;

// index of the array from which the element is taken

int i;

// index of the next element to be picked from the array

int j;

};

// Prototype of a utility function to swap two min-heap nodes

void swap(MinHeapNode \*x, MinHeapNode \*y);

// A class for Min Heap

class MinHeap

{

// pointer to array of elements in heap

MinHeapNode \*harr;

// size of min heap

int heap\_size;

public:

// Constructor: creates a min heap of given size

MinHeap(MinHeapNode a[], int size);

// to heapify a subtree with root at given index

void MinHeapify(int );

// to get index of left child of node at index i

int left(int i) { return (2\*i + 1); }

// to get index of right child of node at index i

int right(int i) { return (2\*i + 2); }

// to get the root

MinHeapNode getMin() { return harr[0]; }

// to replace root with new node x and heapify() new root

void replaceMin(MinHeapNode x) { harr[0] = x; MinHeapify(0); }

};

// This function takes an array of arrays as an argument and

// All arrays are assumed to be sorted. It merges them together

// and prints the final sorted output.

int \*mergeKArrays(int arr[][n], int k)

{

// To store output array

int \*output = new int[n\*k];

// Create a min heap with k heap nodes.

// Every heap node has first element of an array

MinHeapNode \*harr = new MinHeapNode[k];

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

{

// Store the first element

harr[i].element = arr[i][0];

// index of array

harr[i].i = i;

// Index of next element to be stored from the array

harr[i].j = 1;

}

// Create the heap

MinHeap hp(harr, k);

// Now one by one get the minimum element from min

// heap and replace it with next element of its array

for (int count = 0; count < n\*k; count++)

{

// Get the minimum element and store it in output

MinHeapNode root = hp.getMin();

output[count] = root.element;

// Find the next element that will replace current

// root of heap. The next element belongs to same

// array as the current root.

if (root.j < n)

{

root.element = arr[root.i][root.j];

root.j += 1;

}

// If root was the last element of its array

// INT\_MAX is for infinite

else root.element = INT\_MAX;

// Replace root with next element of array

hp.replaceMin(root);

}

return output;

}

// FOLLOWING ARE IMPLEMENTATIONS OF

// STANDARD MIN HEAP METHODS FROM CORMEN BOOK

// Constructor: Builds a heap from a given

// array a[] of given size

MinHeap::MinHeap(MinHeapNode a[], int size)

{

heap\_size = size;

harr = a; // store address of array

int i = (heap\_size - 1)/2;

while (i >= 0)

{

MinHeapify(i);

i--;

}

}

// A recursive method to heapify a

// subtree with root at given index.

// This method assumes that the subtrees

// are already heapified

void MinHeap::MinHeapify(int i)

{

int l = left(i);

int r = right(i);

int smallest = i;

if (l < heap\_size && harr[l].element < harr[i].element)

smallest = l;

if (r < heap\_size && harr[r].element < harr[smallest].element)

smallest = r;

if (smallest != i)

{

swap(&harr[i], &harr[smallest]);

MinHeapify(smallest);

}

}

// A utility function to swap two elements

void swap(MinHeapNode \*x, MinHeapNode \*y)

{

MinHeapNode temp = \*x; \*x = \*y; \*y = temp;

}

// A utility function to print array elements

void printArray(int arr[], int size)

{

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

cout << arr[i] << " ";

}

// Driver program to test above functions

int main()

{

// Change n at the top to change number of elements

// in an array

int arr[][n] = {{2, 6, 12, 34},

{1, 9, 20, 1000},

{23, 34, 90, 2000}};

int k = sizeof(arr)/sizeof(arr[0]);

int \*output = mergeKArrays(arr, k);

cout << "Merged array is " << endl;

printArray(output, n\*k);

return 0;

}

**Output:**

Merged array is

1 2 6 9 12 20 23 34 34 90 1000 2000

**Complexity Analysis:**

* **Time Complexity :**O( n \* k \* log k), Insertion and deletion in a Min Heap requires log k time. So the Overall time complexity is O( n \* k \* log k)
* **Space Complexity :**O(k), If Output is not stored then the only space required is the Min-Heap of k elements. So space Complexity is O(k).

# 317. Merge 2 Binary Max Heaps

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

**Example 1:**

**Input :**

n = 4 m = 3

a[] = {10, 5, 6, 2},

b[] = {12, 7, 9}

**Output :**

{12, 10, 9, 2, 5, 7, 6}

**Explanation :**







**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **mergeHeaps()** which takes the array **a[]**, **b[]**, its size **n**and **m,**as inputs and return the merged max heap. Since there can be multiple solutions, therefore, to check for the correctness of your solution, your answer will be checked by the driver code and will return **1** if it is correct, else it returns **0**.

**Expected Time Complexity:** O(n.Logn)  
**Expected Auxiliary Space:** O(n + m)

**Constraints:**  
1 <= n, m <= 105  
1 <= a[i], b[i] <= 2\*105

## Solution:

The idea is simple. We create an array to store result. We copy both given arrays one by one to result. Once we have copied all elements, we call standard build heap to construct full merged max heap.   
 // C++ program to merge two max heaps.

#include <bits/stdc++.h>

using namespace std;

// Standard heapify function to heapify a

// subtree rooted under idx. It assumes

// that subtrees of node are already heapified.

void maxHeapify(int arr[], int n, int idx)

{

// Find largest of node and its children

if (idx >= n)

return;

int l = 2 \* idx + 1;

int r = 2 \* idx + 2;

int max;

if (l < n && arr[l] > arr[idx])

max = l;

else

max = idx;

if (r < n && arr[r] > arr[max])

max = r;

// Put maximum value at root and

// recur for the child with the

// maximum value

if (max != idx) {

swap(arr[max], arr[idx]);

maxHeapify(arr, n, max);

}

}

// Builds a max heap of given arr[0..n-1]

void buildMaxHeap(int arr[], int n)

{

// building the heap from first non-leaf

// node by calling max heapify function

for (int i = n / 2 - 1; i >= 0; i--)

maxHeapify(arr, n, i);

}

// Merges max heaps a[] and b[] into merged[]

void mergeHeaps(int merged[], int a[], int b[],

int n, int m)

{

// Copy elements of a[] and b[] one by one

// to merged[]

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

merged[i] = a[i];

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

merged[n + i] = b[i];

// build heap for the modified array of

// size n+m

buildMaxHeap(merged, n + m);

}

// Driver code

int main()

{

int a[] = { 10, 5, 6, 2 };

int b[] = { 12, 7, 9 };

int n = sizeof(a) / sizeof(a[0]);

int m = sizeof(b) / sizeof(b[0]);

int merged[m + n];

mergeHeaps(merged, a, b, n, m);

for (int i = 0; i < n + m; i++)

cout << merged[i] << " ";

return 0;

}

Output: 

12 10 9 2 5 7 6

Since time complexity for building the heap from array of n elements is O(n). The complexity of merging the heaps is equal to O(n + m).

**My approach:**

class Solution{

public:

int parent(int a){

return (a-1)/2;

}

vector<int> mergeHeaps(vector<int> &a, vector<int> &b, int n, int m) {

while(m--){

int ele = b.back();

b.pop\_back();

int ind = n, p = parent(ind);

a.push\_back(ele);

while(ind!=0){

if(a[p]<a[ind])

swap(a[p], a[ind]);

ind = p;

p = parent(ind);

}

n++;

}

return a;

}

};

# 318. Kth largest sum continuous subarrays

Given an array of integers. Write a program to find the K-th largest sum of contiguous subarray within the array of numbers which has negative and positive numbers.

**Examples:**

Input: a[] = {20, -5, -1}

k = 3

Output: 14

Explanation: All sum of contiguous

subarrays are (20, 15, 14, -5, -6, -1)

so the 3rd largest sum is 14.

Input: a[] = {10, -10, 20, -40}

k = 6

Output: -10

Explanation: The 6th largest sum among

sum of all contiguous subarrays is -10.

## Solution:

A **brute force approach** is to store all the contiguous sums in another array and sort it and print the k-th largest. But in the case of the number of elements being large, the array in which we store the contiguous sums will run out of memory as the number of contiguous subarrays will be large (quadratic order)

An **efficient approach**is to store the pre-sum of the array in a sum[] array. We can find sum of contiguous subarray from index i to j as sum[j]-sum[i-1]

Now for storing the Kth largest sum, use a min heap (priority queue) in which we push the contiguous sums till we get K elements, once we have our K elements, check if the element is greater than the Kth element it is inserted to the min heap with popping out the top element in the min-heap, else not inserted. In the end, the top element in the min-heap will be your answer.

Below is the implementation of the above approach.

// CPP program to find the k-th largest sum

// of subarray

#include <bits/stdc++.h>

using namespace std;

// function to calculate kth largest element

// in contiguous subarray sum

int kthLargestSum(int arr[], int n, int k)

{

// array to store predix sums

int sum[n + 1];

sum[0] = 0;

sum[1] = arr[0];

for (int i = 2; i <= n; i++)

sum[i] = sum[i - 1] + arr[i - 1];

// priority\_queue of min heap

priority\_queue<int, vector<int>, greater<int> > Q;

// loop to calculate the contiguous subarray

// sum position-wise

for (int i = 1; i <= n; i++)

{

// loop to traverse all positions that

// form contiguous subarray

for (int j = i; j <= n; j++)

{

// calculates the contiguous subarray

// sum from j to i index

int x = sum[j] - sum[i - 1];

// if queue has less then k elements,

// then simply push it

if (Q.size() < k)

Q.push(x);

else

{

// it the min heap has equal to

// k elements then just check

// if the largest kth element is

// smaller than x then insert

// else its of no use

if (Q.top() < x)

{

Q.pop();

Q.push(x);

}

}

}

}

// the top element will be then kth

// largest element

return Q.top();

}

// Driver program to test above function

int main()

{

int a[] = { 10, -10, 20, -40 };

int n = sizeof(a) / sizeof(a[0]);

int k = 6;

// calls the function to find out the

// k-th largest sum

cout << kthLargestSum(a, n, k);

return 0;

}

**Output:**

-10

**Time complexity:** O(n^2 log (k))   
**Auxiliary Space :** O(k) for min-heap and we can store the sum array in the array itself as it is of no use.

# 319. Leetcode- reorganize strings

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

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

**Example 1:**

**Input:** s = "aab"

**Output:** "aba"

**Example 2:**

**Input:** s = "aaab"

**Output:** ""

**Constraints:**

* 1 <= s.length <= 500
* s consists of lowercase English letters.

## Solution:

Alternate placing the most common letters using max heap.

class Solution {

public:

struct compare{

bool operator()(pair<char, int> a, pair<char,int> b){

return a.second<b.second;

}

};

string reorganizeString(string s) {

unordered\_map<char, int> mp;

for(char ch: s)

mp[ch]++;

priority\_queue<pair<char, int>, vector<pair<char,int>>, compare> pq;

for(auto i:mp)

pq.push({i.first, i.second});

string res = "";

pair<char,int> p,q;

while(!pq.empty()){

p = pq.top(); pq.pop();

if(p.first==res.back()){

res = "";

break;

}

res = res + p.first;

if(!pq.empty()){

q = pq.top(); pq.pop();

res = res + q.first;

}

if(p.second>1)

pq.push({p.first, p.second-1});

if(q.second>1)

pq.push({q.first, q.second-1});

}

return res;

}

};

**Time Complexity:** O(nlogn)

**Space Complexity:** O(n)

# 320. Merge “K” Sorted Linked Lists [V.IMP]

## Same as ques 156 of Linked List.

# 321. Smallest range in “K” Lists

You have k lists of sorted integers in **non-decreasing order**. Find the **smallest** range that includes at least one number from each of the k lists.

We define the range [a, b] is smaller than range [c, d] if b - a < d - c **or** a < c if b - a == d - c.

**Example 1:**

**Input:** nums = [[4,10,15,24,26],[0,9,12,20],[5,18,22,30]]

**Output:** [20,24]

**Explanation:**

List 1: [4, 10, 15, 24,26], 24 is in range [20,24].

List 2: [0, 9, 12, 20], 20 is in range [20,24].

List 3: [5, 18, 22, 30], 22 is in range [20,24].

**Example 2:**

**Input:** nums = [[1,2,3],[1,2,3],[1,2,3]]

**Output:** [1,1]

**Constraints:**

* nums.length == k
* 1 <= k <= 3500
* 1 <= nums[i].length <= 50
* -105 <= nums[i][j] <= 105
* nums[i] is sorted in **non-decreasing** order.

## Solution:

Approach #1 Brute Force [Time Limit Exceeded]

The naive approach is to consider every pair of elements, nums[i][j]*nums*[*i*][*j*] and nums[k][l]*nums*[*k*][*l*] from amongst the given lists and consider the range formed by these elements. For every range currently considered, we can traverse over all the lists to find if atleast one element from these lists can be included in the current range. If so, we store the end-points of the current range and compare it with the previous minimum range found, if any, satisfying the required criteria, to find the smaller range from among them.

Once all the element pairs have been considered as the ranges, we can obtain the required minimum range.

public class Solution {

public int[] smallestRange(int[][] nums) {

int minx = 0, miny = Integer.MAX\_VALUE;

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

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

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

for (int l = (k == i ? j : 0); l < nums[k].length; l++) {

int min = Math.min(nums[i][j], nums[k][l]);

int max = Math.max(nums[i][j], nums[k][l]);

int n, m;

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

for (n = 0; n < nums[m].length; n++) {

if (nums[m][n] >= min && nums[m][n] <= max)

break;

}

if (n == nums[m].length)

break;

}

if (m == nums.length) {

if (miny - minx > max - min || (miny - minx == max - min && minx > min)) {

miny = max;

minx = min;

}

}

}

}

}

}

return new int[] {minx, miny};

}

}

**Complexity Analysis**

* Time complexity : O(n^3)*O*(*n*3). Considering every possible range(element pair) requires O(n^2)*O*(*n*2) time. For each range considered, we need to traverse over all the elements of the given lists in the worst case requiring another O(n)*O*(*n*) time. Here, n*n* refers to the total number of elements in the given lists.
* Space complexity : O(1)*O*(1). Constant extra space is used.

Approach #2 Better Brute Force [Time Limit Exceeded]

**Algorithm**

In the last approach, we consider every possible range and then traverse over every list to check if atleast one of the elements from these lists lies in the required range. Instead of doing this traversal for every range, we can make use of Binary Search to find the index of the element just larger than(or equal to) the lower limit of the range currently considered.

If all the elements in the current list are lesser than this lower limit, we'll get the index as nums[k].length*nums*[*k*].*length* for the k^{th}*kth* list being currently checked. In this case, none of the elements of the current list lies in the current range.

On the other hand, if all the elements in this list are larger than this lower limit, we'll get the index of the first element(minimum) in the current list. If this element happens to be larger than the upper limit of the range currently considered, then also, none of the elements of the current list lies within the current range.

Whenever a range is found which satisfies the required criteria, we can compare it with the minimum range found so far to determine the required minimum range.

public class Solution {

public int[] smallestRange(int[][] nums) {

int minx = 0, miny = Integer.MAX\_VALUE;

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

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

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

for (int l = (k == i ? j : 0); l < nums[k].length; l++) {

int min = Math.min(nums[i][j], nums[k][l]);

int max = Math.max(nums[i][j], nums[k][l]);

int n, m;

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

n = Arrays.binarySearch(nums[m], min);

if (n < 0)

n = -1 - n;

if (n == nums[m].length || nums[m][n] < min || nums[m][n] > max)

break;

}

if (m == nums.length) {

if (miny - minx > max - min || (miny - minx == max - min && minx > min)) {

miny = max;

minx = min;

}

}

}

}

}

}

return new int[] {minx, miny};

}

}

**Complexity Analysis**

* Time complexity : O\big(n^2log(k)\big)*O*(*n*2*log*(*k*)). The time required to consider every possible range is O(n^2)*O*(*n*2). For every range currently considered, a Binary Search requiring O\big(log(k)\big)*O*(*log*(*k*)) time is required. Here, n*n* refers to the total number of elements in the given lists and k*k* refers to the average length of each list.
* Space complexity : O(1)*O*(1). Constant extra space is used.

Approach #3 Using Pointers [Time Limit Exceeded]

**Algorithm**

We'll discuss about the implementation used in the current approach along with the idea behind it.

This approach makes use of an array of pointers, next*next*, whose length is equal to the number of given lists. In this array, next[i]*next*[*i*] refers to the element which needs to be considered next in the (i-1)^{th}(*i*−1)*th* list. The meaning of this will become more clearer when we look at the process.

We start by initializing all the elements of next*next* to 0. Thus, currently, we are considering the first(minimum) element among all the lists. Now, we find out the index of the list containing the maximum(max\_i*maxi*​) and minimum(min\_i*mini*​) elements from amongst the elements currently pointed by next*next*. The range formed by these maximum and minimum elements surely  
contains atleast one element from each list.

But, now our objective is to minimize this range. To do so, there are two options: Either decrease the maximum value or increase the minimum value.

Now, the maximum value can't be reduced any further, since it already corresponds to the minimum value in one of the lists. Reducing it any further will lead to the exclusion of all the elements of this list(containing the last maximum value) from the new range.

Thus, the only option left in our hand is to try to increase the minimum value. To do so, we now need to consider the next element in the list containing the last minimum value. Thus, we increment the entry at the corresponding index in next*next*, to indicate that the next element in this list now needs to be considered.

Thus, at every step, we find the maximum and minimum values being pointed currently, update the next*next* values appropriately, and also find out the range formed by these maximum and minimum values to find out the smallest range satisfying the given criteria.

While doing this process, if any of the lists gets completely exhausted, it means that the minimum value being increased for minimizing the range being considered can't be increased any further, without causing the exclusion of all the elements in atleast one of the lists. Thus, we can stop the search process whenever any list gets completely exhausted.

We can also stop the process, when all the elements of the given lists have been exhausted.

Summarizing the statements above, the process becomes:

1. Initialize next*next* array(pointers) with all 0's.
2. Find the indices of the lists containing the minimum(min\_i*mini*​) and the maximum(max\_i*maxi*​) elements amongst the elements pointed by the next*next* array.
3. If the range formed by the maximum and minimum elements found above is larger than the previous maximum range, update the boundary values used for the maximum range.
4. Increment the pointer nums[min\_i]*nums*[*mini*​].
5. Repeat steps 2 to 4 till any of the lists gets exhausted.

The animation below illustrates the process for a visual understanding of the process.

public class Solution {

public int[] smallestRange(int[][] nums) {

int minx = 0, miny = Integer.MAX\_VALUE;

int[] next = new int[nums.length];

boolean flag = true;

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

for (int j = 0; j < nums[i].length && flag; j++) {

int min\_i = 0, max\_i = 0;

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

if (nums[min\_i][next[min\_i]] > nums[k][next[k]])

min\_i = k;

if (nums[max\_i][next[max\_i]] < nums[k][next[k]])

max\_i = k;

}

if (miny - minx > nums[max\_i][next[max\_i]] - nums[min\_i][next[min\_i]]) {

miny = nums[max\_i][next[max\_i]];

minx = nums[min\_i][next[min\_i]];

}

next[min\_i]++;

if (next[min\_i] == nums[min\_i].length) {

flag = false;

}

}

}

return new int[] {minx, miny};

}

}

**Complexity Analysis**

* Time complexity : O(n\*m)*O*(*n*∗*m*). In the worst case, we need to traverse over next*next* array(of length m*m*) for all the elements of the given lists. Here, n*n* refers to the total number of elements in all the lists. m*m* refers to the total number of lists.
* Space complexity : O(m)*O*(*m*). next*next* array of size m*m* is used.

**Efficient approach:** The approach remains the same but the time complexity can be reduced by using min-heap or *priority queue*. Min heap can be used to find the maximum and minimum value in logarithmic time or log k time instead of linear time. Rest of the approach remains the same.

* **Algorithm:**
  1. create an Min heap to store k elements, one from each array and a variable *minrange* initialized to a maximum value and also keep a variable *max* to store the maximum integer.
  2. Initially put the first element of each element from each list and store the maximum value in *max*.
  3. Repeat the following steps until atleast one list exhausts :
     1. To find the minimum value or *min*, use the top or root of the Min heap which is the minimum element.
     2. Now update the minrange if current (max-min) is less than minrange.
     3. remove the top or root element from priority queue and insert the next element from the list which contains the min element and update the max with the new element inserted.
* **My Implementation:**

class Solution {

public:

struct compare{

bool operator()(vector<int> a, vector<int> b){

return a[0]>b[0];

}

};

vector<int> smallestRange(vector<vector<int>>& nums) {

priority\_queue<vector<int>, vector<vector<int>>, compare> pq;

int low, high=0;

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

vector<int> t = {nums[i][0], i, 0};

if(nums[i][0]>high)

high = nums[i][0];

pq.push(t);

}

low = pq.top()[0];

vector<int> res = {low, high};

while(!pq.empty()){

vector<int> t = pq.top();

pq.pop();

if(t[2]<nums[t[1]].size()-1){

vector<int> temp = {nums[t[1]][t[2]+1], t[1], t[2]+1};

if(temp[0]>high)

high = temp[0];

pq.push(temp);

}

else

break;

if(pq.size()>1)

low = pq.top()[0];

if(high-low < res[1] - res[0]){

res[0] = low;

res[1] = high;

}

}

return res;

}

};

* **Complexity Analysis:**
  + **Time complexity :** O(n \* k \*log k).   
    To find the maximum and minimum in an Min Heap of length k the time required is O(log k), and to traverse all the k arrays of length n (in worst case), the time complexity is O(n\*k), then the total time complexity is O(n \* k \*log k).
  + **Space complexity:** O(k).   
    The priority queue will store k elements so the space complexity os O(k)

# 322. Median in a stream of Integers

Given an input stream of **N** integers. The task is to insert these numbers into a new stream and find the median of the stream formed by each insertion of **X** to the new stream.

**Example 1:**

**Input:**

N = 4

X[] = 5,15,1,3

**Output:**

5

10

5

4

**Explanation:**Flow in stream : 5, 15, 1, 3

5 goes to stream --> median 5 (5)

15 goes to stream --> median 10 (5,15)

1 goes to stream --> median 5 (5,15,1)

3 goes to stream --> median 4 (5,15,1 3)

**Example 2:**

**Input:**

N = 3

X[] = 5,10,15

**Output:**

5

7.5

10

**Explanation:**Flow in stream : 5, 10, 15

5 goes to stream --> median 5 (5)

10 goes to stream --> median 7.5 (5,10)

15 goes to stream --> median 10 (5,10,15)

**Your Task:**  
You are required to complete the class Solution.   
It should have 2 data members to represent 2 heaps.   
It should have the following member functions:

1. **insertHeap()** which takes **x**as input and inserts it into the heap, the function should then call **balanceHeaps()**to balance the new heap.
2. **balanceHeaps()**does not take any arguments. It is supposed to balance the two heaps.
3. **getMedian()**does not take any arguments. It should return the current median of the stream.

**Expected Time Complexity** : O(nlogn)  
**Expected Auxilliary Space** : O(n)

**Constraints:**

1<= N <= 106  
1 <= x <= 106

## Solution:

**Method 1:** Insertion Sort  
If we can sort the data as it appears, we can easily locate the median element. *Insertion Sort* is one such online algorithm that sorts the data appeared so far. At any instance of sorting, say after sorting *i*-th element, the first *i* elements of the array are sorted. The insertion sort doesn’t depend on future data to sort data input till that point. In other words, insertion sort considers data sorted so far while inserting the next element. This is the key part of insertion sort that makes it an online algorithm.  
However, insertion sort takes O(n2) time to sort *n* elements. Perhaps we can use *binary search* on *insertion sort* to find the location of the next element in O(log n) time. Yet, we can’t do data movement in O(log n) time. No matter how efficient the implementation is, it takes polynomial time in case of insertion sort.  
Interested readers can try the implementation of Method 1.

// This code is contributed by Anjali Saxena

#include <bits/stdc++.h>

using namespace std;

// Function to find position to insert current element of

// stream using binary search

int binarySearch(int arr[], int item, int low, int high)

{

if (low >= high) {

return (item > arr[low]) ? (low + 1) : low;

}

int mid = (low + high) / 2;

if (item == arr[mid])

return mid + 1;

if (item > arr[mid])

return binarySearch(arr, item, mid + 1, high);

return binarySearch(arr, item, low, mid - 1);

}

// Function to print median of stream of integers

void printMedian(int arr[], int n)

{

int i, j, pos, num;

int count = 1;

cout << "Median after reading 1"

<< " element is " << arr[0] << "\n";

for (i = 1; i < n; i++) {

float median;

j = i - 1;

num = arr[i];

// find position to insert current element in sorted

// part of array

pos = binarySearch(arr, num, 0, j);

// move elements to right to create space to insert

// the current element

while (j >= pos) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = num;

// increment count of sorted elements in array

count++;

// if odd number of integers are read from stream

// then middle element in sorted order is median

// else average of middle elements is median

if (count % 2 != 0) {

median = arr[count / 2];

}

else {

median = (arr[(count / 2) - 1] + arr[count / 2])

/ 2;

}

cout << "Median after reading " << i + 1

<< " elements is " << median << "\n";

}

}

// Driver Code

int main()

{

int arr[] = { 5, 15, 1, 3, 2, 8, 7, 9, 10, 6, 11, 4 };

int n = sizeof(arr) / sizeof(arr[0]);

printMedian(arr, n);

return 0;

}

**Output**

Median after reading 1 element is 5

Median after reading 2 elements is 10

Median after reading 3 elements is 5

Median after reading 4 elements is 4

Median after reading 5 elements is 3

Median after reading 6 elements is 4

Median after reading 7 elements is 5

Median after reading 8 elements is 6

Median after reading 9 elements is 7

Median after reading 10 elements is 6

Median after reading 11 elements is 7

Median after reading 12 elements is 6

**Time Complexity:** O(n2)

**Space Complexity:** O(1)

**Method 2:** Augmented self-balanced binary search tree (AVL, RB, etc…)  
At every node of BST, maintain a number of elements in the subtree rooted at that node. We can use a node as the root of a simple binary tree, whose left child is self-balancing BST with elements less than root and right child is self-balancing BST with elements greater than root. The root element always holds *effective median*.  
If the left and right subtrees contain a same number of elements, the root node holds the average of left and right subtree root data. Otherwise, the root contains the same data as the root of subtree which is having more elements. After processing an incoming element, the left and right subtrees (BST) are differed utmost by 1.  
Self-balancing BST is costly in managing the balancing factor of BST. However, they provide sorted data which we don’t need. We need median only. The next method makes use of Heaps to trace the median.

**Method 3:** Heaps  
Similar to balancing BST in Method 2 above, we can use a max heap on the left side to represent elements that are less than *effective median*, and a min-heap on the right side to represent elements that are greater than *effective median*.  
After processing an incoming element, the number of elements in heaps differs utmost by 1 element. When both heaps contain the same number of elements, we pick the average of heaps root data as *effective median*. When the heaps are not balanced, we select *effective median* from the root of the heap containing more elements.  
Given below is the implementation of the above method.

// C++ program to find med in

// stream of running integers

#include<bits/stdc++.h>

using namespace std;

// function to calculate med of stream

void printMedians(double arr[], int n)

{

// max heap to store the smaller half elements

priority\_queue<double> s;

// min heap to store the greater half elements

priority\_queue<double,vector<double>,greater<double> > g;

double med = arr[0];

s.push(arr[0]);

cout << med << endl;

// reading elements of stream one by one

/\* At any time we try to make heaps balanced and

their sizes differ by at-most 1. If heaps are

balanced,then we declare median as average of

min\_heap\_right.top() and max\_heap\_left.top()

If heaps are unbalanced,then median is defined

as the top element of heap of larger size \*/

for (int i=1; i < n; i++)

{

double x = arr[i];

// case1(left side heap has more elements)

if (s.size() > g.size())

{

if (x < med)

{

g.push(s.top());

s.pop();

s.push(x);

}

else

g.push(x);

med = (s.top() + g.top())/2.0;

}

// case2(both heaps are balanced)

else if (s.size()==g.size())

{

if (x < med)

{

s.push(x);

med = (double)s.top();

}

else

{

g.push(x);

med = (double)g.top();

}

}

// case3(right side heap has more elements)

else

{

if (x > med)

{

s.push(g.top());

g.pop();

g.push(x);

}

else

s.push(x);

med = (s.top() + g.top())/2.0;

}

cout << med << endl;

}

}

// Driver program to test above functions

int main()

{

// stream of integers

double arr[] = {5, 15, 10, 20, 3};

int n = sizeof(arr)/sizeof(arr[0]);

printMedians(arr, n);

return 0;

}

**Output:**

5

10

10

12.5

10

**Complexity Analysis:** 

* **Time Complexity:** O(n Log n).   
  Time Complexity to insert element in min heap is log n. So to insert n element is O( n log n).
* **Auxiliary Space :** O(n).   
  The Space required to store the elements in Heap is O(n).

**My Implementation of above approach:**

// C++ program to find med in

// stream of running integers

#include<bits/stdc++.h>

using namespace std;

// function to calculate med of stream

void printMedians(double arr[], int n)

{

// max heap to store the smaller half elements

priority\_queue<double> s;

// min heap to store the greater half elements

priority\_queue<double,vector<double>,greater<double> > g;

double med = arr[0];

s.push(arr[0]);

cout << med << endl;

// reading elements of stream one by one

/\* At any time we try to make heaps balanced and

their sizes differ by at-most 1. If heaps are

balanced,then we declare median as average of

min\_heap\_right.top() and max\_heap\_left.top()

If heaps are unbalanced,then median is defined

as the top element of heap of larger size \*/

for (int i=1; i < n; i++)

{

double x = arr[i];

// case1(left side heap has more elements)

if (s.size() > g.size())

{

if (x < med)

{

g.push(s.top());

s.pop();

s.push(x);

}

else

g.push(x);

med = (s.top() + g.top())/2.0;

}

// case2(both heaps are balanced)

else if (s.size()==g.size())

{

if (x < med)

{

s.push(x);

med = (double)s.top();

}

else

{

g.push(x);

med = (double)g.top();

}

}

// case3(right side heap has more elements)

else

{

if (x > med)

{

s.push(g.top());

g.pop();

g.push(x);

}

else

s.push(x);

med = (s.top() + g.top())/2.0;

}

cout << med << endl;

}

}

// Driver program to test above functions

int main()

{

// stream of integers

double arr[] = {5, 15, 10, 20, 3};

int n = sizeof(arr)/sizeof(arr[0]);

printMedians(arr, n);

return 0;

}

**Output:**

5

10

10

12.5

10

**Complexity Analysis:** 

* **Time Complexity:** O(n Log n).   
  Time Complexity to insert element in min heap is log n. So to insert n element is O( n log n).
* **Auxiliary Space :** O(n).   
  The Space required to store the elements in Heap is O(n).

# 323. Check if a Binary Tree is Heap

Given a binary tree. The task is to check whether the given tree follows the **max heap** property or not.  
**Note:**Properties of a tree to be a max heap - Completeness and Value of node greater than or equal to its child.

**Example 1:**

**Input:**

      5

   / \

   2 3

**Output:** 1

**Explanation:** The given tree follows **max-heap** property since 5,

is root and it is greater than both its children.

**Example 2:**

**Input:**

       10

    /   \

   20   30

  /   \

40   60

**Output:** 0

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **isHeap()** which takes the root of Binary Tree as parameter returns **True** if the given binary tree is a **heap** else returns **False**.  
  
**Expected Time Complexity:**O(N)  
**Expected Space Complexity:**O(N)

**Constraints:**  
1 ≤ Number of nodes ≤ 100  
1 ≤ Data of a node ≤ 1000

## Solution:

We check each of the above condition separately, for checking completeness isComplete and for checking heap isHeapUtil function are written.   
Detail about isComplete function can be found [here](https://www.geeksforgeeks.org/check-whether-binary-tree-complete-not-set-2-recursive-solution/).  
isHeapUtil function is written considering the following things –

1. Every Node can have 2 children, 0 child (last level nodes) or 1 child (there can be at most one such node).
2. If Node has No child then it’s a leaf node and returns true (Base case)
3. If Node has one child (it must be left child because it is a complete tree) then we need to compare this node with its single child only.
4. If the Node has both child then check heap property at Node at recur for both subtrees.   
   Complete code.

Below is the implementation of the above approach:

/\* C++ program to checks if a

binary tree is max heap or not \*/

#include <bits/stdc++.h>

using namespace std;

/\* Tree node structure \*/

struct Node

{

int key;

struct Node \*left;

struct Node \*right;

};

/\* Helper function that

allocates a new node \*/

struct Node \*newNode(int k)

{

struct Node \*node = new Node;

node->key = k;

node->right = node->left = NULL;

return node;

}

/\* This function counts the

number of nodes in a binary tree \*/

unsigned int countNodes(struct Node\* root)

{

if (root == NULL)

return (0);

return (1 + countNodes(root->left)

+ countNodes(root->right));

}

/\* This function checks if the

binary tree is complete or not \*/

bool isCompleteUtil (struct Node\* root,

unsigned int index,

unsigned int number\_nodes)

{

// An empty tree is complete

if (root == NULL)

return (true);

// If index assigned to

// current node is more than

// number of nodes in tree,

// then tree is not complete

if (index >= number\_nodes)

return (false);

// Recur for left and right subtrees

return (isCompleteUtil(root->left, 2\*index + 1,

number\_nodes) &&

isCompleteUtil(root->right, 2\*index + 2,

number\_nodes));

}

// This Function checks the

// heap property in the tree.

bool isHeapUtil(struct Node\* root)

{

// Base case : single

// node satisfies property

if (root->left == NULL && root->right == NULL)

return (true);

// node will be in

// second last level

if (root->right == NULL)

{

// check heap property at Node

// No recursive call ,

// because no need to check last level

return (root->key >= root->left->key);

}

else

{

// Check heap property at Node and

// Recursive check heap

// property at left and right subtree

if (root->key >= root->left->key &&

root->key >= root->right->key)

return ((isHeapUtil(root->left)) &&

(isHeapUtil(root->right)));

else

return (false);

}

}

// Function to check binary

// tree is a Heap or Not.

bool isHeap(struct Node\* root)

{

// These two are used

// in isCompleteUtil()

unsigned int node\_count = countNodes(root);

unsigned int index = 0;

if (isCompleteUtil(root, index,

node\_count)

&& isHeapUtil(root))

return true;

return false;

}

// Driver code

int main()

{

struct Node\* root = NULL;

root = newNode(10);

root->left = newNode(9);

root->right = newNode(8);

root->left->left = newNode(7);

root->left->right = newNode(6);

root->right->left = newNode(5);

root->right->right = newNode(4);

root->left->left->left = newNode(3);

root->left->left->right = newNode(2);

root->left->right->left = newNode(1);

// Function call

if (isHeap(root))

cout << "Given binary tree is a Heap\n";

else

cout << "Given binary tree is not a Heap\n";

return 0;

}

**Output**

Given binary tree is a Heap

**Method 2:** (Iterative approach using level order traversal)

// C++ program to checks if a

// binary tree is max heap or not

#include <bits/stdc++.h>

using namespace std;

// Tree node structure

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// To add a new node

struct Node\* newNode(int k)

{

struct Node\* node = new Node;

node->data = k;

node->right = node->left = NULL;

return node;

}

bool isHeap(Node\* root)

{

// Your code here

queue<Node\*> q;

q.push(root);

bool nullish = false;

while (!q.empty())

{

Node\* temp = q.front();

q.pop();

if (temp->left)

{

if (nullish

|| temp->left->data

>= temp->data)

{

return false;

}

q.push(temp->left);

}

else {

nullish = true;

}

if (temp->right)

{

if (nullish

|| temp->right->data

>= temp->data)

{

return false;

}

q.push(temp->right);

}

else

{

nullish = true;

}

}

return true;

}

// Driver code

int main()

{

struct Node\* root = NULL;

root = newNode(10);

root->left = newNode(9);

root->right = newNode(8);

root->left->left = newNode(7);

root->left->right = newNode(6);

root->right->left = newNode(5);

root->right->right = newNode(4);

root->left->left->left = newNode(3);

root->left->left->right = newNode(2);

root->left->right->left = newNode(1);

// Function call

if (isHeap(root))

cout << "Given binary tree is a Heap\n";

else

cout << "Given binary tree is not a Heap\n";

return 0;

}

**Output**

Given binary tree is a Heap

**My Implementation:**

class Solution {

public:

bool isComplete(Node\* tree){

int last\_level = -1;

queue<Node\*> q;

q.push(tree);

int level = 0;

while(!q.empty()){

int size = q.size();

level++;

while(size--){

Node\* tp = q.front();

q.pop();

if(tp->left){

if(level==last\_level-1)

return false;

q.push(tp->left);

}

else if(last\_level == -1)

last\_level = level +1;

if(tp->right){

if(level==last\_level-1)

return false;

q.push(tp->right);

}

else if(last\_level==-1)

last\_level = level +1;

if(!(tp->left) && !(tp->right) && level!=last\_level && level!=(last\_level-1)){

return false;

}

}

}

return true;

}

bool fun(Node\* tree){

if(!tree)

return true;

if(tree->left && tree->left->data > tree->data)

return false;

if(tree->right && tree->right->data > tree->data)

return false;

return fun(tree->left) && fun(tree->right);

}

bool isHeap(struct Node\* tree) {

return isComplete(tree) && fun(tree);

}

};

# 324. Connect “n” ropes with minimum cost

There are given **N** ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. The task is to connect the ropes with minimum cost.

**Example 1:**

**Input:**

n = 4

arr[] = {4, 3, 2, 6}

**Output:**

29

**Explanation:**

For example if we are given 4

ropes of lengths 4, 3, 2 and 6. We can

connect the ropes in following ways.

1) First connect ropes of lengths 2 and 3.

Now we have three ropes of lengths 4, 6

and 5.

2) Now connect ropes of lengths 4 and 5.

Now we have two ropes of lengths 6 and 9.

3) Finally connect the two ropes and all

ropes have connected.

Total cost for connecting all ropes is 5

+ 9 + 15 = 29. This is the optimized cost

for connecting ropes. Other ways of

connecting ropes would always have same

or more cost. For example, if we connect

4 and 6 first (we get three strings of 3,

2 and 10), then connect 10 and 3 (we get

two strings of 13 and 2). Finally we

connect 13 and 2. Total cost in this way

is 10 + 13 + 15 = 38.

**Example 2:**

**Input:**

n = 5

arr[] = {4, 2, 7, 6, 9}

**Output:**

62

**Explanation:**

First, connect ropes 4 and 2, which makes

the array {6,7,6,9}. Next, add ropes 6 and

6, which results in {12,7,9}. Then, add 7

and 9, which makes the array {12,16}. And

finally add these two which gives {28}.

Hence, the total cost is 6 + 12 + 16 +

28 = 62.

**Your Task:**  
You don't need to read input or print anything. Your task isto complete the function **minCost()** which takes 2 arguments and returns the minimum cost.

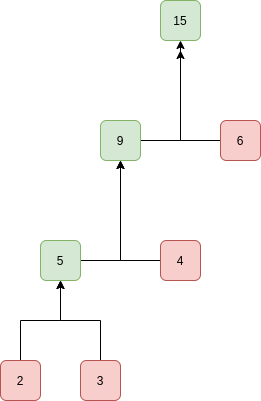
**Expected Time Complexity** : O(nlogn)  
**Expected Auxilliary Space** : O(n)

**Constraints:**  
1 ≤ N ≤ 100000  
1 ≤ arr[i] ≤ 106

## Solution:

**Solution:**   
If we observe the above problem closely, we can notice that the lengths of the ropes which are picked first are included more than once in total cost. Therefore, the idea is to connect the smallest two ropes first and recur for the remaining ropes. This approach is similar to [Huffman Coding](https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/). We put the smallest ropes down the tree so that they can be repeated multiple times rather than the longer ropes.

*So it forms a structure like a tree:*



The sum contains sum of depth of each value. For array (2, 3, 4, 6) the sum is equal to 2 \* 3 + 3 \* 3 + 4 \* 2 + 6 \* 1 = 29 (According to the diagram).

**Algorithm:**

1. Create a min-heap and insert all lengths into the min-heap.
2. Do following while the number of elements in min-heap is not one.
   1. Extract the minimum and second minimum from min-heap
   2. Add the above two extracted values and insert the added value to the min-heap.
   3. Maintain a variable for total cost and keep incrementing it by the sum of extracted values.
3. Return the value of this total cost.

Following is the implementation of the above algorithm.

#include <bits/stdc++.h>

using namespace std;

int minCost(int arr[], int n)

{

// Create a priority queue

// https:// www.geeksforgeeks.org/priority-queue-in-cpp-stl/

// By default 'less' is used which is for decreasing order

// and 'greater' is used for increasing order

priority\_queue<int, vector<int>, greater<int> > pq(arr, arr + n);

// Initialize result

int res = 0;

// While size of priority queue is more than 1

while (pq.size() > 1) {

// Extract shortest two ropes from pq

int first = pq.top();

pq.pop();

int second = pq.top();

pq.pop();

// Connect the ropes: update result and

// insert the new rope to pq

res += first + second;

pq.push(first + second);

}

return res;

}

// Driver program to test above function

int main()

{

int len[] = { 4, 3, 2, 6 };

int size = sizeof(len) / sizeof(len[0]);

cout << "Total cost for connecting ropes is " << minCost(len, size);

return 0;

}

**Output:**

Total cost for connecting ropes is 29

**Complexity Analysis:**

* **Time Complexity:**O(nLogn), assuming that we use a O(nLogn) sorting algorithm.   
  Note that heap operations like insert and extract take O(Logn) time.
* **Auxiliary Complexity:** O(n).   
  The space required to store the values in min heap

# 325. Convert Complete BST to Min Heap with the condition that all the values in the left subtree of each node should be less than all the values in the right subtree of the node.

Given a binary search tree which is also a complete binary tree. The problem is to convert the given BST into a Min Heap with the condition that all the values in the left subtree of a node should be less than all the values in the right subtree of the node. This condition is applied on all the nodes in the so converted Min Heap.   
**Examples:** 

Input : 4

/ \

2 6

/ \ / \

1 3 5 7

Output : 1

/ \

2 5

/ \ / \

3 4 6 7

The given **BST** has been transformed into a

**Min Heap.**

All the nodes in the Min Heap satisfies the given

condition, that is, values in the left subtree of

a node should be less than the values in the right

subtree of the node.

## Solution:

1. Create an array **arr[]** of size **n**, where n is the number of nodes in the given BST.
2. Perform the inorder traversal of the BST and copy the node values in the **arr[]** in sorted order.
3. Now perform the preorder traversal of the tree.
4. While traversing the root during the preorder traversal, one by one copy the values from the array **arr[]** to the nodes.

// C++ implementation to convert the given

// BST to Min Heap

#include <bits/stdc++.h>

using namespace std;

// structure of a node of BST

struct Node

{

int data;

Node \*left, \*right;

};

/\* Helper function that allocates a new node

with the given data and NULL left and right

pointers. \*/

struct Node\* getNode(int data)

{

struct Node \*newNode = new Node;

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// function prototype for preorder traversal

// of the given tree

void preorderTraversal(Node\*);

// function for the inorder traversal of the tree

// so as to store the node values in 'arr' in

// sorted order

void inorderTraversal(Node \*root, vector<int>& arr)

{

if (root == NULL)

return;

// first recur on left subtree

inorderTraversal(root->left, arr);

// then copy the data of the node

arr.push\_back(root->data);

// now recur for right subtree

inorderTraversal(root->right, arr);

}

// function to convert the given BST to MIN HEAP

// performs preorder traversal of the tree

void BSTToMinHeap(Node \*root, vector<int> arr, int \*i)

{

if (root == NULL)

return;

// first copy data at index 'i' of 'arr' to

// the node

root->data = arr[++\*i];

// then recur on left subtree

BSTToMinHeap(root->left, arr, i);

// now recur on right subtree

BSTToMinHeap(root->right, arr, i);

}

// utility function to convert the given BST to

// MIN HEAP

void convertToMinHeapUtil(Node \*root)

{

// vector to store the data of all the

// nodes of the BST

vector<int> arr;

int i = -1;

// inorder traversal to populate 'arr'

inorderTraversal(root, arr);

// BST to MIN HEAP conversion

BSTToMinHeap(root, arr, &i);

}

// function for the preorder traversal of the tree

void preorderTraversal(Node \*root)

{

if (!root)

return;

// first print the root's data

cout << root->data << " ";

// then recur on left subtree

preorderTraversal(root->left);

// now recur on right subtree

preorderTraversal(root->right);

}

// Driver program to test above

int main()

{

// BST formation

struct Node \*root = getNode(4);

root->left = getNode(2);

root->right = getNode(6);

root->left->left = getNode(1);

root->left->right = getNode(3);

root->right->left = getNode(5);

root->right->right = getNode(7);

convertToMinHeapUtil(root);

cout << "Preorder Traversal:" << endl;

preorderTraversal(root);

return 0;

}

**Output:**

Preorder Traversal:

1 2 3 4 5 6 7

**Time Complexity:** O(n)   
**Auxiliary Space:** O(n)

# Convert BST into a Min-Heap without using array

Given a Binary Search Tree, convert it into a Min-Heap containing the same elements in O(n) time. Do this in-place. 

Input: Binary Search Tree

8

/ \

4 12

/ \ / \

2 6 10 14

Output - Min Heap

2

/ \

4 6

/ \ / \

8 10 12 14

[Or any other tree that follows Min Heap

properties and has same keys]

## Solution:

If we are allowed to use extra space, we can perform inorder traversal of the tree and store the keys in an auxiliary array. As we’re doing inorder traversal on a BST, array will be sorted. Finally, we construct a complete binary tree from the sorted array. We construct the binary tree level by level and from left to right by taking next minimum element from sorted array. The constructed binary tree will be a min-Heap. This solution works in O(n) time, but is not in-place.  
**How to do it in-place?**   
The idea is to convert the binary search tree into a sorted linked list first. We can do this by traversing the BST in inorder fashion. We add nodes at the beginning of current linked list and update head of the list using pointer to head pointer. Since we insert at the beginning, to maintain sorted order, we first traverse the right subtree before the left subtree. i.e. do a reverse inorder traversal.  
Finally we convert the sorted linked list into a min-Heap by setting the left and right pointers appropriately. We can do this by doing a Level order traversal of the partially built Min-Heap Tree using queue and traversing the linked list at the same time. At every step, we take the parent node from queue, make next two nodes of linked list as children of the parent node, and enqueue the next two nodes to queue. As the linked list is sorted, the min-heap property is maintained.  
Below is the implementation of above idea –

// Program to convert a BST into a Min-Heap

// in O(n) time and in-place

#include <bits/stdc++.h>

using namespace std;

// Node for BST/Min-Heap

struct Node

{

int data;

Node \*left, \*right;

};

// Utility function for allocating node for BST

Node\* newNode(int data)

{

Node\* node = new Node;

node->data = data;

node->left = node->right = NULL;

return node;

}

// Utility function to print Min-heap level by level

void printLevelOrder(Node \*root)

{

// Base Case

if (root == NULL) return;

// Create an empty queue for level order traversal

queue<Node \*> q;

q.push(root);

while (!q.empty())

{

int nodeCount = q.size();

while (nodeCount > 0)

{

Node \*node = q.front();

cout << node->data << " ";

q.pop();

if (node->left)

q.push(node->left);

if (node->right)

q.push(node->right);

nodeCount--;

}

cout << endl;

}

}

// A simple recursive function to convert a given

// Binary Search tree to Sorted Linked List

// root --> Root of Binary Search Tree

// head\_ref --> Pointer to head node of created

// linked list

void BSTToSortedLL(Node\* root, Node\*\* head\_ref)

{

// Base cases

if(root == NULL)

return;

// Recursively convert right subtree

BSTToSortedLL(root->right, head\_ref);

// insert root into linked list

root->right = \*head\_ref;

// Change left pointer of previous head

// to point to NULL

if (\*head\_ref != NULL)

(\*head\_ref)->left = NULL;

// Change head of linked list

\*head\_ref = root;

// Recursively convert left subtree

BSTToSortedLL(root->left, head\_ref);

}

// Function to convert a sorted Linked

// List to Min-Heap.

// root --> Root of Min-Heap

// head --> Pointer to head node of sorted

// linked list

void SortedLLToMinHeap(Node\* &root, Node\* head)

{

// Base Case

if (head == NULL)

return;

// queue to store the parent nodes

queue<Node \*> q;

// The first node is always the root node

root = head;

// advance the pointer to the next node

head = head->right;

// set right child to NULL

root->right = NULL;

// add first node to the queue

q.push(root);

// run until the end of linked list is reached

while (head)

{

// Take the parent node from the q and remove it from q

Node\* parent = q.front();

q.pop();

// Take next two nodes from the linked list and

// Add them as children of the current parent node

// Also in push them into the queue so that

// they will be parents to the future nodes

Node \*leftChild = head;

head = head->right; // advance linked list to next node

leftChild->right = NULL; // set its right child to NULL

q.push(leftChild);

// Assign the left child of parent

parent->left = leftChild;

if (head)

{

Node \*rightChild = head;

head = head->right; // advance linked list to next node

rightChild->right = NULL; // set its right child to NULL

q.push(rightChild);

// Assign the right child of parent

parent->right = rightChild;

}

}

}

// Function to convert BST into a Min-Heap

// without using any extra space

Node\* BSTToMinHeap(Node\* &root)

{

// head of Linked List

Node \*head = NULL;

// Convert a given BST to Sorted Linked List

BSTToSortedLL(root, &head);

// set root as NULL

root = NULL;

// Convert Sorted Linked List to Min-Heap

SortedLLToMinHeap(root, head);

}

// Driver code

int main()

{

/\* Constructing below tree

8

/ \

4 12

/ \ / \

2 6 10 14

\*/

Node\* root = newNode(8);

root->left = newNode(4);

root->right = newNode(12);

root->right->left = newNode(10);

root->right->right = newNode(14);

root->left->left = newNode(2);

root->left->right = newNode(6);

BSTToMinHeap(root);

/\* Output - Min Heap

2

/ \

4 6

/ \ / \

8 10 12 14

\*/

printLevelOrder(root);

return 0;

}

**Output :**

2

4 6

8 10 12 14

# 326. Convert min heap to max heap

Given array representation of min Heap, convert it to max Heap in O(n) time.   
**Example :** 

Input: arr[] = [3 5 9 6 8 20 10 12 18 9]

3

/ \

5 9

/ \ / \

6 8 20 10

/ \ /

12 18 9

Output: arr[] = [20 18 10 12 9 9 3 5 6 8] OR

[any Max Heap formed from input elements]

20

/ \

18 10

/ \ / \

12 9 9 3

/ \ /

5 6 8

## Solution:

The problem might look complex at first look. But our final goal is to only build the max heap. The idea is very simple – we simply build Max Heap without caring about the input. We start from the bottom-most and rightmost internal mode of min Heap and heapify all internal modes in the bottom-up way to build the Max heap.  
Below is its implementation

// A C++ program to convert min Heap to max Heap

#include<bits/stdc++.h>

using namespace std;

// to heapify a subtree with root at given index

void MaxHeapify(int arr[], int i, int n)

{

int l = 2\*i + 1;

int r = 2\*i + 2;

int largest = i;

if (l < n && arr[l] > arr[i])

largest = l;

if (r < n && arr[r] > arr[largest])

largest = r;

if (largest != i)

{

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

MaxHeapify(arr, largest, n);

}

}

// This function basically builds max heap

void convertMaxHeap(int arr[], int n)

{

// Start from bottommost and rightmost

// internal mode and heapify all internal

// modes in bottom up way

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

MaxHeapify(arr, i, n);

}

// A utility function to print a given array

// of given size

void printArray(int\* arr, int size)

{

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

printf("%d ", arr[i]);

}

// Driver program to test above functions

int main()

{

// array representing Min Heap

int arr[] = {3, 5, 9, 6, 8, 20, 10, 12, 18, 9};

int n = sizeof(arr)/sizeof(arr[0]);

printf("Min Heap array : ");

printArray(arr, n);

convertMaxHeap(arr, n);

printf("\nMax Heap array : ");

printArray(arr, n);

return 0;

}

**Output :**

Min Heap array : 3 5 9 6 8 20 10 12 18 9

Max Heap array : 20 18 10 12 9 9 3 5 6 8

The complexity of above solution might looks like O(nLogn) but it is O(n).

# 327. Rearrange characters in a string such that no two adjacent are same.

## Same as ques 80 of strings.

# 328. Minimum sum of two numbers formed from digits of an array

Given an array Arr of size N such that each element is from the range 0 to 9. Find the minimum possible sum of two numbers formed using the elements of the array. All digits in the given array must be used to form the two numbers.

**Example 1:**

**Input:**

N = 6

Arr[] = {6, 8, 4, 5, 2, 3}

**Output:** 604

**Explanation:** The minimum sum is formed

by numbers 358 and 246.

**Example 2:**

**Input:**

N = 5

Arr[] = {5, 3, 0, 7, 4}

**Output:** 82

**Explanation:** The minimum sum is

formed by numbers 35 and 047.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **solve()** which takes **arr[]** and **n**as input parameters and returns the minimum possible sum. As the number can be large, return string presentation of the number without leading zeroes.

**Expected Time Complexity:** O(N\*logN)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ N ≤ 107  
0 ≤ Arri ≤ 9

## Solution:

A minimum number will be formed from set of digits when smallest digit appears at most significant position and next smallest digit appears at next most significant position ans so on..  
The idea is to sort the array in increasing order and build two numbers by alternating picking digits from the array. So first number is formed by digits present in odd positions in the array and second number is formed by digits from even positions in the array. Finally, we return the sum of first and second number.  
Below is the implementation of above idea.

// C++ program to find minimum sum of two numbers

// formed from digits of the array.

#include <bits/stdc++.h>

using namespace std;

// Function to find and return minimum sum of

// two numbers formed from digits of the array.

int solve(int arr[], int n)

{

// sort the array

sort(arr, arr + n);

// let two numbers be a and b

int a = 0, b = 0;

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

{

// fill a and b with every alternate digit

// of input array

if (i & 1)

a = a\*10 + arr[i];

else

b = b\*10 + arr[i];

}

// return the sum

return a + b;

}

// Driver code

int main()

{

int arr[] = {6, 8, 4, 5, 2, 3};

int n = sizeof(arr)/sizeof(arr[0]);

cout << "Sum is " << solve(arr, n);

return 0;

}

**Output :**

Sum is 604

**Method 2 (For Large Numbers)**

When we have to deal with very big numbers (as in the [PRACTICE](https://practice.geeksforgeeks.org/problems/minimum-sum4058/1) section of this question) the above approach will not work. The basic idea of approaching the question is the same as above, but instead of using numbers, we will use strings to handle sum.

To add two numbers given in form of the string, you can refer to [this](https://www.geeksforgeeks.org/sum-two-large-numbers/).

class Solution{

public:

string solve(int arr[], int n) {

sort(arr, arr+n);

string res = "";

int carry = 0, temp = 0;

for(int i=n-1; i>0; i-=2){

temp = carry + arr[i] + arr[i-1];

carry = temp/10;

res = to\_string(temp%10) + res;

}

if(n%2!=0){

temp = carry + arr[0];

carry = temp/10;

res = to\_string(temp%10) + res;

}

res = to\_string(carry) + res;

while(res[0]=='0')

res = res.substr(1, res.size());

return res;

}

};

**Time Complexity:** O(N\*logN)

**Auxiliary Space:** O(1)