Heaps by Aditya Verma

27 August 2023 01:31 PM

Heap is represented same as stack in strcture of some size k

Identification of heap??
Always look for 2 keywords,
K and smallest/largest
If these 2 words come in combination means it's a heap question

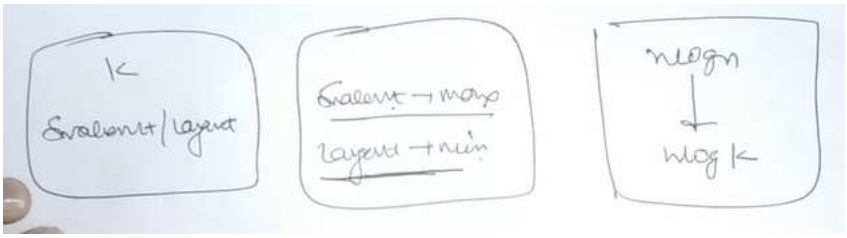
There are 2 types of heap
Min heap
Max heap

If given:
K + smallest = max heap of size k
K + largest = min heap of size k

Why we take min heap with kth largest?
Because min heap stores all minimum elements and when we remove elements from top till size k. We reach kth largest element in min heap
Sameway we use max heap in kth smallest element

Heap has top(), push(), pop()

Why k is given in heap questions?
All heap questions are based on sorting which takes nlogn
If k is given we make a heap of size k, sorting complexity reduces to nlogk



Question: kth smallest element
Arr: [7,10,4,3,20,15]
K = 3

Find 3rd smallest element?
3rd smallest means heap will be used
Smallest -> max heap

We sort array 3,4,7,10,15,20 now get arr[3] element in nlogn but
We have k in given part so why to sort whole array

Arr: [7,10,4,3,20,15]

K = 3
Max heap has top element as maximum
We take max heap of size k and sort k elements at a time

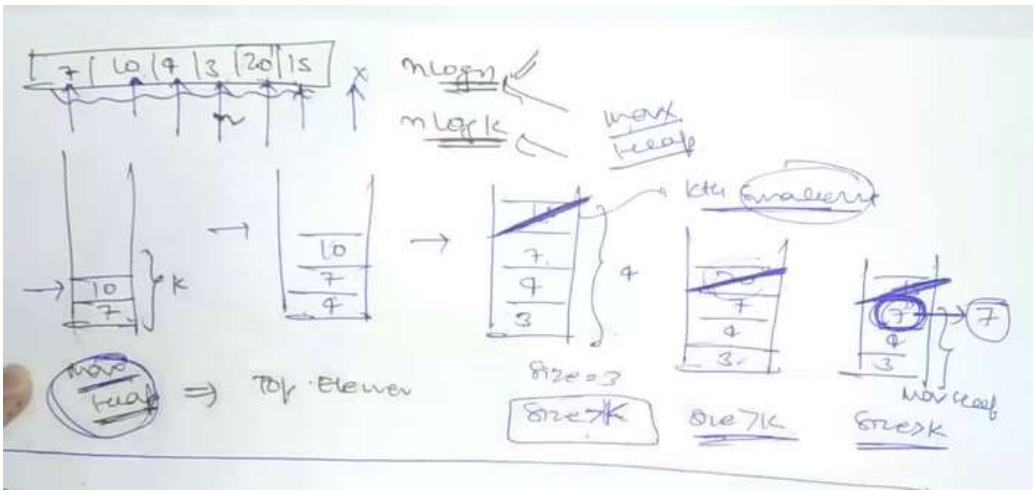
Push 7
Push 10 , 7 , 10
Push 4, 4,7,10

Push 3, 3,4,7,10
As size>k, we pop so pop(10)
Heap looks like 3,4,7

Push 20, 3,4,7,20
Size>k so pop(20)

Push 15, 3,4,7,15
Size>k so pop(15)

Array finish
Heap looks like 3,4,7
Kth smallest element is top() = 7



Complexity reduces to nlogk from nlogn

How to make Heap? (use STL)

Max heap
Priority_queue<int> maxh;

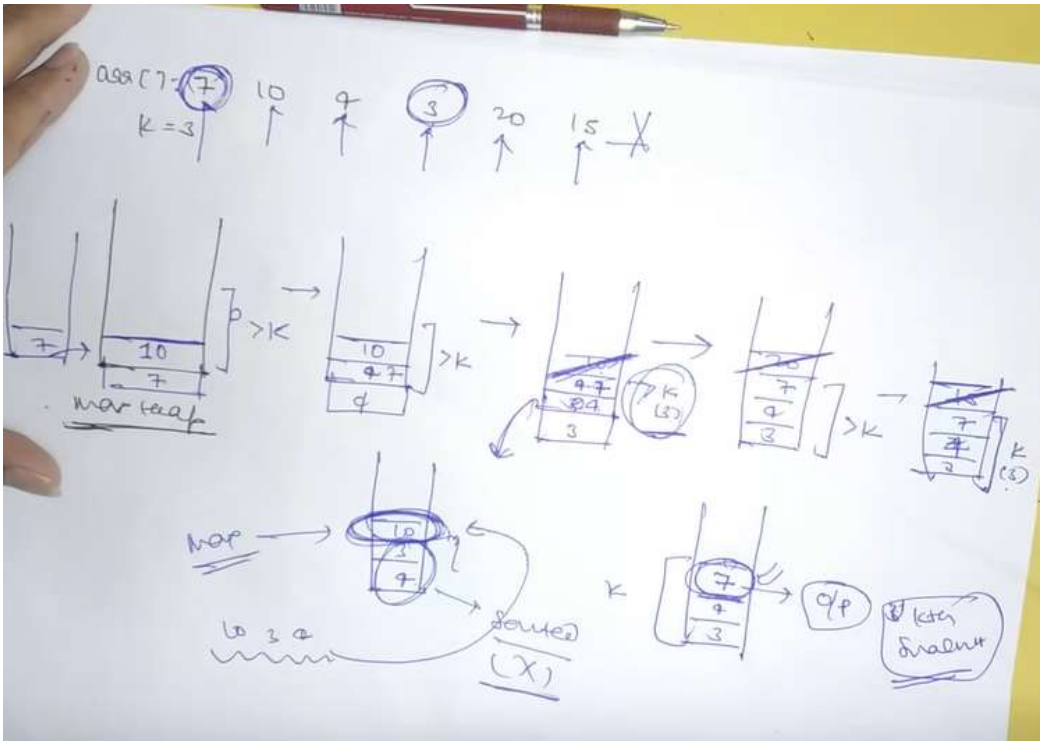
Min heap
Priority_queue<int,vector<int>,greater<int>> minh;

Kth Smallest Numbers

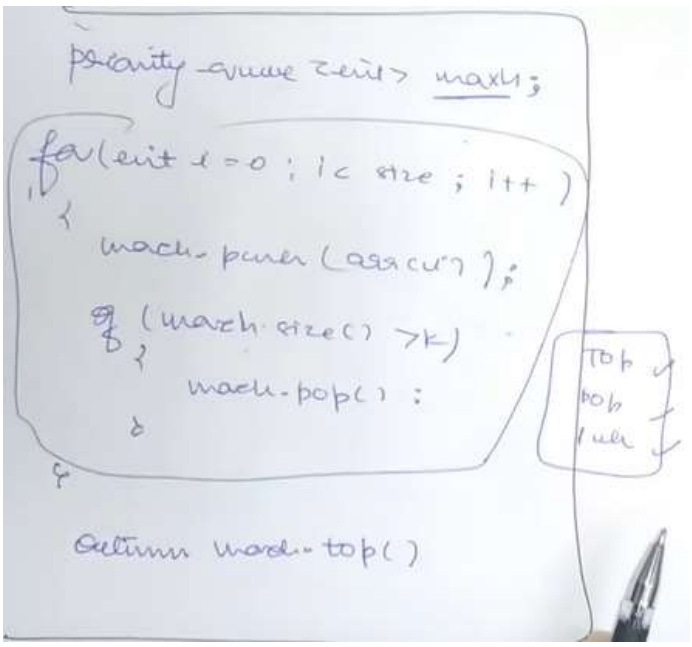
Arr[] = [7,10,4,3,20,15]
K = 3
O/p = 7

Sort() and return arr[k-1], TC: nlogn
We see, heap can be used here so

Kth smallest: we make max heap of size = k
Heap does not store elements in sorted order, it only guarantees that top element will be max in a max heap



Code



Kth Largest element

Given an integer array nums and an integer k, return *the* k[th] *largest element in the array*.

Note that it is the kth largest element in the sorted order, not the kth distinct element.
Can you solve it without sorting?

Example 1:
Input: nums = [3,2,1,5,6,4], k = 2
Output: 5

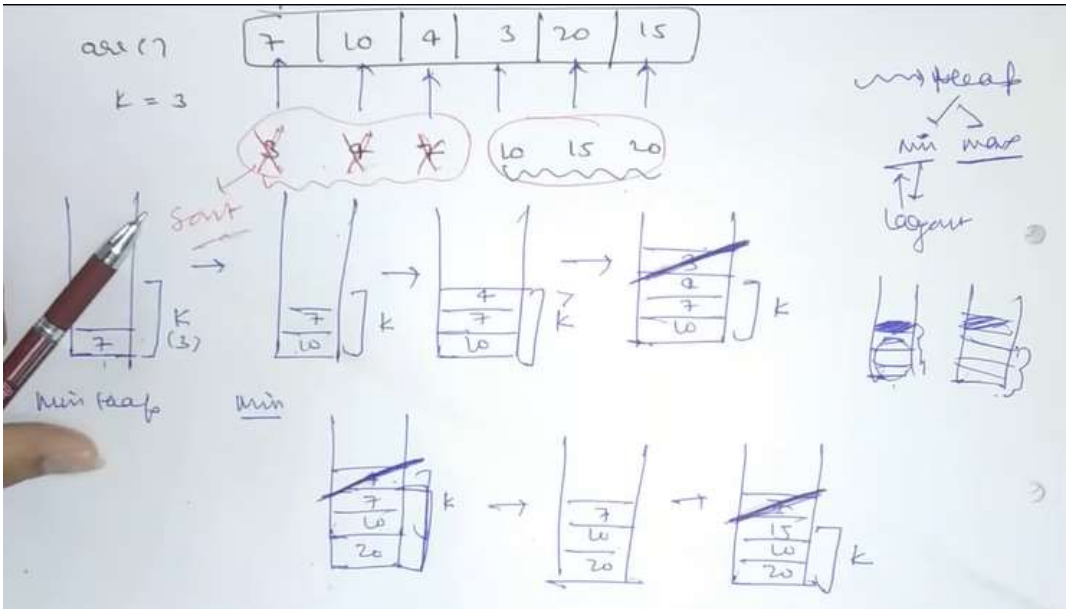
```
int findKthLargest(vector<int>& nums, int k) {
      priority_queue<int,vector<int>,greater<int>> pq;
      int n = nums.size();
      for(int i =0;i<n;i++)
      {
         pq.push(nums[i]);
         if(pq.size()>k)
         {
            pq.pop();
         }
      }

      return pq.top();
   }
```

Return kth largest elements in Array

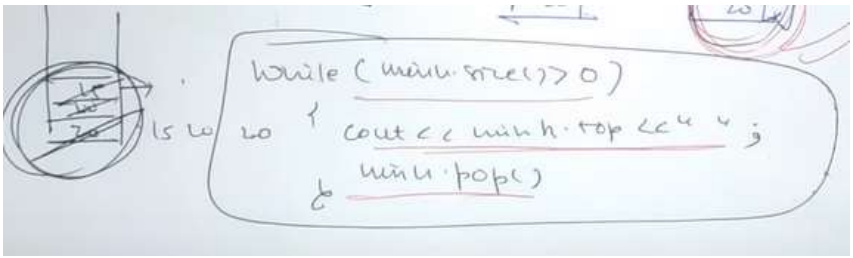
Arr[] = [7,10,4,3,20,15]
K = 3

Approach 1: Sort the array and move from arr[size-1] -> arr[size-k]

Approach 2: Use Heap as its kth largest so use min_heap (top element in the smallest)



Print kth largest elements



Sort k sorted array or nearly sorted array

Arr[] = [6,5,3,2,8,10,9]
K = 3
K sorted means sorted in group of 3 already

Given an array of **N** elements, where each element is at most K away from its target position, devise an algorithm that sorts in O(N log K) time.

Examples:

Input: *arr[] = {6, 5, 3, 2, 8, 10, 9}, K = 3*
Output: *arr[] = {2, 3, 5, 6, 8, 9, 10}*

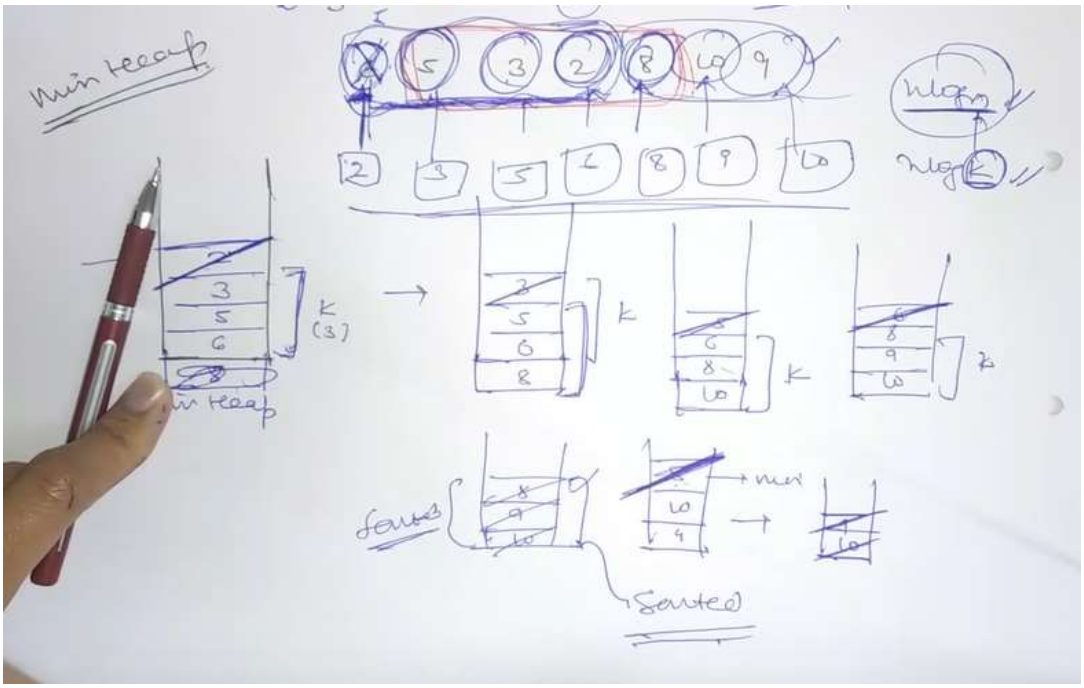
Input: *arr[] = {10, 9, 8, 7, 4, 70, 60, 50}, k = 4*
Output: *arr[] = {4, 7, 8, 9, 10, 50, 60, 70}*

Approach 1: sort whole array

Approach 2: Making use of k sorted arrays condition, To get element at 0th position we can check 0 + kth positions, for 1st position, check 1 + kth position and so on..

We want min element first so we make a min heap

We push 6,5,3,2 so heap looks like 6,5,3,2 now size>k so we know arr[0] = 2 i.e top element of min heap
Now pop out
Push(8), heap looks like 8,6,5,3 so size>k
Arr[1] = 3 and pop
Push(10)



K Closest Number

Given an array and k and x
Get k numbers closest to x in the array

Arr[] = [5,6,7,8,9]
K = 3, x = 7

We can subtract x from each element in array and get the element with lowest result
So we need elements with lowest results at the top means we need min heap where top k elements will be our answer

We can take pair<int,int> for min heap where pair stores absolute difference and element itself in the min_heap

```
vector<int> findClosestElements(vector<int>& arr, int k, int x) {
    // we will make a min_heap of pair<int,int> to store difference and element
    priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;
    for(int i = 0;i<arr.size();i++)
    {
        // storing difference and element in heap
        pq.push({abs(x - arr[i]),arr[i]});
    }

    // It is a min_heap means smallest difference will be at the top
    vector<int> ans;
    int i = k;
    while(i--)
    {
        // get top k element and store in vector answer
        ans.push_back(pq.top().second);
        pq.pop();
```

```
    }

    // sort the answer vector
    sort(ans.begin(), ans.end());
    return ans;
  }
```

Top K frequent Numbers

Given an integer array nums and an integer k, return *the* k *most frequent elements*. You may return the answer in **any order**.

Example 1:
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]
Example 2:
Input: nums = [1], k = 1
Output: [1]
Hashing Solution

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    unordered_map<int,int>m;
    for(auto it:nums){
        m[it]++;
    }
    vector<pair<int,int> >v;
    for(auto it:m){
        v.push_back({it.second,it.first});
    }
    sort(v.begin(),v.end(),greater<pair<int,int>>());

    vector<int>ans;
    for(int i=0;i<k;i++){
        ans.push_back(v[i].second);
    }

    return ans;
  }
```

Heap Solution

Here our key is based upon frequency and number
<pair<int,int>> should store frequency and element
We need greater frequency at the top means we need max heap.
We used sorting in the above solution, we can use heap instead of sorting and reduce complexity from nlogn to nlogk

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    unordered_map<int,int> mpp;
    int n = nums.size();
    for(int i = 0;i<n;i++)
    {
        mpp[nums[i]]++;
```

```
    }

    // Making a max heap to store frequency and element
    priority_queue<pair<int,int>> maxH;
    for(auto it: mpp)
    {
        maxH.push({it.second,it.first});
    }

    int i = k;
    vector<int> ans;
    while(i--)
    {
        ans.push_back(maxH.top().second);
        maxH.pop();
    }

    return ans;
  }
```

Sort Array by Increasing Frequency

Given an array of integers nums, sort the array in **increasing** order based on the frequency of the values. If multiple values have the same frequency, sort them in **decreasing** order.
Return the *sorted array*.

Example 1:
Input: nums = [1,1,2,2,2,3]
Output: [3,1,1,2,2,2]
Explanation: '3' has a frequency of 1, '1' has a frequency of 2, and '2' has a frequency of 3.
Example 2:
Input: nums = [2,3,1,3,2]
Output: [1,3,3,2,2]
Explanation: '2' and '3' both have a frequency of 2, so they are sorted in decreasing order.
Example 3:
Input: nums = [-1,1,-6,4,5,-6,1,4,1]
Output: [5,-1,4,4,-6,-6,1,1,1]

Heap Solution

```
vector<int> frequencySort(vector<int>& nums) {
    // We can store the frequency in a map
    // We can use min_heap to get minimum frequency element at the top
    // We push element, freq number of times in the ans vector
    // We use pair<int,int> to store frequency and element
    unordered_map<int,int> mpp;
    int n = nums.size();
    for(int i = 0;i<n;i++)
    {
        mpp[nums[i]]++;
```

```
        }

        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;
        for(auto it: mpp)
        {
            pq.push({it.second,it.first});
        }

        // Now we get one element at a time from top of heap
        // and store frequency times in the array
        vector<int> ans;

        while(!pq.empty())
        {
            int k = pq.top().first;
            while(k--)
            {
                ans.push_back(pq.top().second);
            }
            pq.pop();
        }
        return ans;

    }
```

Actual solution using map

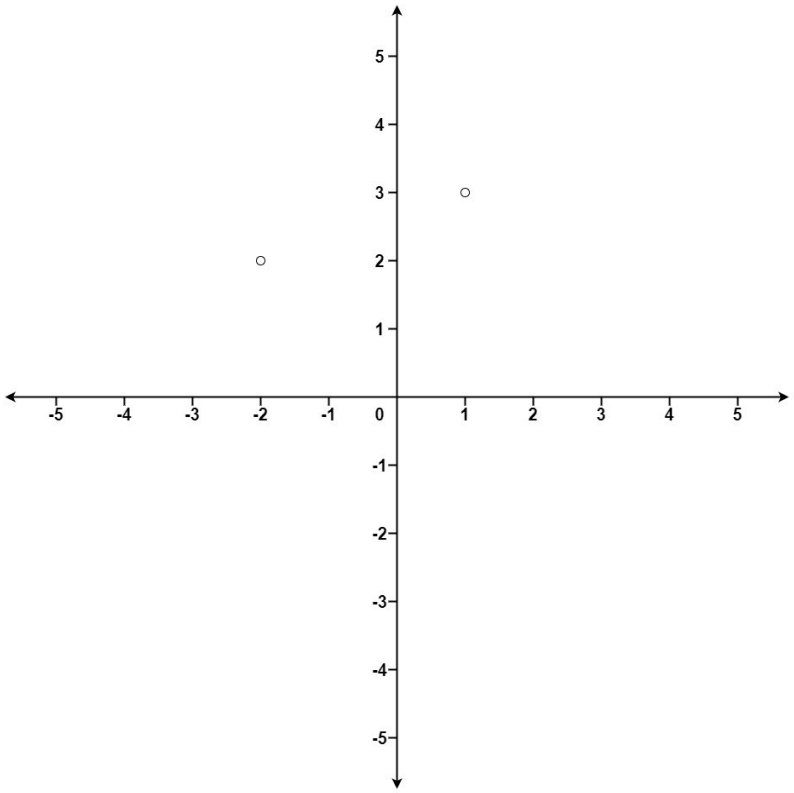
```
bool static comparator(pair<int,int>m,pair<int,int>n){
        if(m.second==n.second)  // if frequency same
            return m.first>n.first; // return whose element is bigger
        else
            return m.second<n.second; // return whose frequency is bigger
    }
    vector<int> frequencySort(vector<int>& nums) {
        // We can store the frequency in a map
        // We can use min_heap to get minimum frequency element at the top
        // We push element, freq number of times in the ans vector
        // We use pair<int,int> to store frequency and element
        unordered_map<int,int> mpp;
        int n = nums.size();
        for(int i = 0;i<n;i++)
        {
            mpp[nums[i]]++;
        }

        // Now we get one element at a time from top of heap
        // and store frequency times in the array
        vector<pair<int,int>> v1;
        for(auto k: mpp)
        {
            v1.push_back(k);
        }
```

```
    // We will make a vector of pair and store all frequency and elements in that
vector
    // now, we sort that vector on basis of comparator which says if two numbers
have same frequency then put the bigger one first then smaller one
    // say 3->2, 2->2 then in final array, 3 comes first then 2 comes
    sort(v1.begin(),v1.end(),comparator);

    vector<int> ans;
    // store all elements in ans vector
    for(auto y: v1)
    {
      // till frequency becomes 0, store the element
      while(y.second--)
      {
        ans.push_back(y.first);
      }
    }

    return ans;
  }
```

K Closest Points to Origin

Given an array of points where points[i] = [x_i, y_i] represents a point on the **X-Y** plane and an integer k, return the k closest points to the origin (0, 0).
The distance between two points on the **X-Y** plane is the Euclidean distance (i.e., $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$).
You may return the answer in **any order**. The answer is **guaranteed** to be **unique** (except for the order that it is in).

Example 1:



Input: points = [[1,3],[-2,2]], k = 1
Output: [[-2,2]]
Explanation:
The distance between (1, 3) and the origin is sqrt(10).
The distance between (-2, 2) and the origin is sqrt(8).
Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.
We only want the closest k = 1 points from the origin, so the answer is just [[-2,2]].
Example 2:
Input: points = [[3,3],[5,-1],[-2,4]], k = 2
Output: [[3,3],[-2,4]]
Explanation: The answer [[-2,4],[3,3]] would also be accepted.

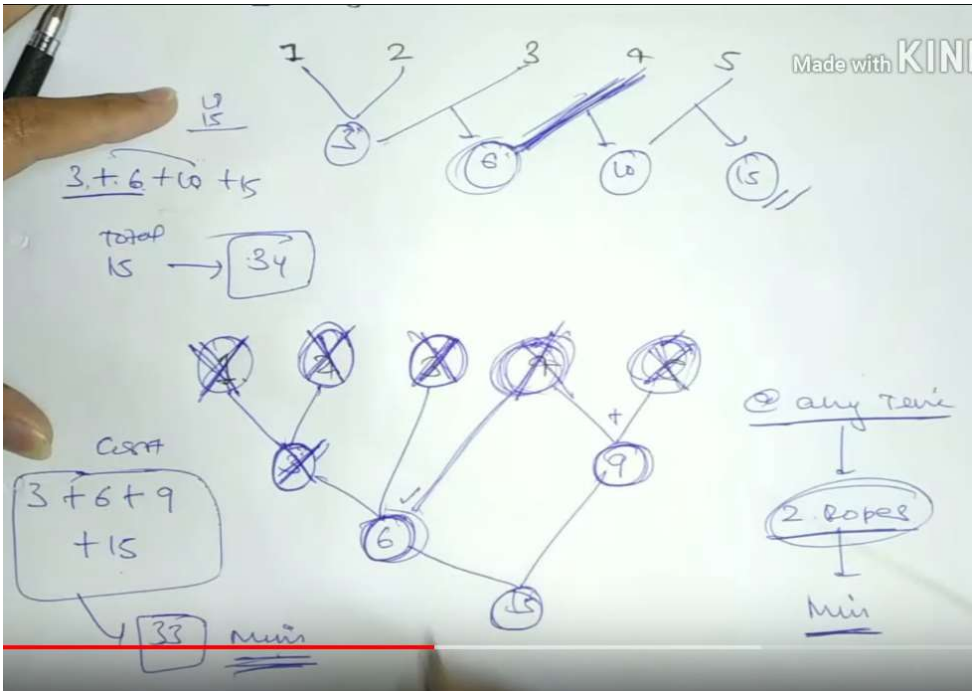
```
vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {
    // we can store distance and coordinate in vector
    // we need minimum distance for k elements means we need max heap
    // we pop if size>k this way we always maintain lower k values in the heap
    vector<vector<int>> result(k);
    //maxheap storage initialization
    priority_queue<vector<int>> maxHeap;
    //Construction of maxheap
    for (auto& p : points) {
        int x = p[0], y = p[1]; // as the other point is origin always so formula reduces
easily
       // we can store x^2+y^2 instead of sqrt(x^2+y^2) its one of the same thing
       // because values will be sorted on the basis of their values only
        maxHeap.push({x*x + y*y, x, y});
        if (maxHeap.size() > k) {
            maxHeap.pop();
        }
    }
```

```
    for (int i = 0; i < k; ++i) {
      // In the top vector we have {distance value, coordinate 1, coordinate 2}
      vector<int> top = maxHeap.top();
      maxHeap.pop();
      // top[1] means coordinate 1, top[2] means coordinate 2
      result[i] = {top[1], top[2]};
    }

    return result;
  }
```

Connect N Ropes With Minimum Cost

You have been 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. We need to connect the ropes with minimum cost.

Sample Input 1:
4
4 3 2 6
Sample Output 1:
29
Explanation:
1) If we first connect ropes of lengths 2 and 3, we will left with three ropes of lengths 4, 6 and 5.
2) Now then, if we connect ropes of lengths 4 and 5, we will left with two ropes of lengths 6 and 9.
3) Finally, we connect the remaining two ropes and all ropes are now connected.
Total cost for connecting all ropes in this way is 5 + 9 + 15 = 29 which is the optimized cost.
Now there are other ways also for connecting ropes. For example, if we connect 4 and 6 first (we get three ropes of lengths 3, 2 and 10), then connect 10 and 3 (we get two ropes of length 13 and 2). Finally we connect 13 and 2. Total cost in this way is 10 + 13 + 15 = 38 which is not the optimal cost
Sample Input 2:
5
1 2 3 4 5
Sample Output 2:
33



At any time we will connect 2 rope which are minimum so that our cost reduces

```
long long connectRopes(int* arr, int n)
{
    // To get the minimum cost, we need to take 2 rope of minimum l
ength and add them
    // Everytime we need 2 minimum ropes means we can use a min hea
p
    // We will take top 2 elements, pop them from heap and add them
 and add that answer in total cost
    // Now we push that answer in heap again
    // Now we again repeat this process and we do this till heap.si
ze()>1
    // If only one element left in the heap, we have our answer, re
turn it
    priority_queue<int,vector<int>,greater<int>> pq;
    for(int i = 0;i<n;i++)
    {
        pq.push(arr[i]);
    }
    long long ans = 0;
    while(pq.size()>1)
    {
        long long first = pq.top();
        pq.pop();
        long long second = pq.top();
        pq.pop();
        ans += (first + second);
        pq.push(first+second);
    }
    return ans;
}
```

Sum of elements between k1'th and k2'th smallest elements

Given an array **A[]** of **N** positive integers and two positive integers K_1 and K_2. Find the sum of all elements between K_1^{th} and K_2^{th} smallest elements of the array. It may be assumed that (1 <= k1 < k2 <= n).

Example 1:

Input:
N = 7
A[] = {20, 8, 22, 4, 12, 10, 14}
K1 = 3, K2 = 6
Output:26
Explanation:
3rd smallest element is 10
6th smallest element is 20
Element between 10 and 20
12,14. Their sum = 26.

Example 2:

Input
N = 6
A[] = {10, 2, 50, 12, 48, 13}
K1= 2, K2 = 6
Output:
73

```
long long kthsmallest(long long A[], long long N, long long K)
 {
     // we need kth smallest so we take max heap
     priority_queue<int> pq;
     for(int i = 0;i<N;i++)
     {
        pq.push(A[i]);
        if(pq.size()>K)
        {
           pq.pop();
        }
     }
     return pq.top();
  }
  long long sumBetweenTwoKth( long long A[], long long N, long long K1, long long K2)
  {
     long long index1 = kthsmallest(A,N,K1);
     long long index2 = kthsmallest(A,N,K2);

     // Find the k1 smallest element, k2th smallest element
     // Add all numbers whose value>k1th && value<k2th

     long long sum = 0;
     for(int i = 0;i<N;i++)
     {
        if(A[i]>index1 && A[i]<index2)
```

```
        {
            sum += A[i];
        }
    }

    return sum;
}
```