HEAP NOTES

ADITYA VERMA

MAX Heap

priority_queue <int> maxHeap;

Example:

```
#include <bits/stdc++.h>
using namespace std;
 int main ()
{
   // Creates a max heap
   priority_queue <int> pq;
   pq.push(5);
   pq.push(1);
   pq.push(10);
   pq.push(30);
   pq.push(20);

   // One by one extract items from max heap
   while (pq.empty() == false)
   {
      cout << pq.top() << " ";
      pq.pop();
   }

   return 0;
}
```

MIN Heap

priority_queue <int, vector<int>, greater<int> > minHeap;

Example:

```
#include <bits/stdc++.h>
using namespace std;

int main ()
{
   // Creates a min heap
   priority_queue <int, vector<int>, greater<int> > pq;
   pq.push(5);
   pq.push(1);
   pq.push(10);
   pq.push(30);
   pq.push(20);

   // One by one extract items from min heap
   while (pq.empty() == false)
   {
      cout << pq.top() << " ";
      pq.pop();
   }
   return 0;
}
```

Heap most used functions:

1. pq.pop()
2. pq.push()
3. pq.top()

Where **pq** is min/max heap

Pairs used in heap

priority_queue <int> maxHeap;

priority_queue <int, vector<int>, greater<int> > minHeap;

replace int with "pair<int, int>"

Like -> take an eg of max heap where max element will be on top

But we have to get its index also -> **(element, index)** as pair is stored below

Max function of heap will only apply to key or first element.

(10, 3)
(8, 1)
(2, 0)
(1, 2)

_____

CODE:

```
// C++ program to create a priority queue of pairs.
// By default a max heap is created ordered
// by first element of pair.
#include <bits/stdc++.h>
using namespace std;

int main()
{
   // By default a max heap is created ordered
   // by first element of pair.
   priority_queue<pair<int, int> > pq;

   pq.push(make_pair(10, 200));
   pq.push(make_pair(20, 100));
   pq.push(make_pair(15, 400));

   pair<int, int> top = pq.top();
   cout << top.first << " " << top.second;
   return 0;
}
```

Output:

20 100

Pairs used in MIN HEAP:

priority_queue <int> maxHeap;

priority_queue <int, vector<int>, greater<int> > minHeap;

Replace int with **pair<int, int>**

Suppose we have to create min Heap

priority_queue < pair<int, int>, vector< pair<int, int>>, greater< pair<int, int>> > minHeap;

CODE:

```
// C++ program to create a priority queue of pairs.
// We can create a min heap by passing adding two
// parameters, vector and greater().
#include <bits/stdc++.h>
using namespace std;

typedef pair<int, int> pi;

int main()
{
   // By default a min heap is created ordered by first element of pair.
   priority_queue<pi, vector<pi>, greater<pi> > pq;

   pq.push(make_pair(10, 200));
   pq.push(make_pair(20, 100));
```

```
    pq.push(make_pair(15, 400));

    pair<int, int> top = pq.top();
    cout << top.first << " " << top.second;
    return 0;
}
```

Output:

10 200

<u>**Heap Introduction and Identification**</u>

Identification – **k** and **smallest** / **largest**

But first, if possible give sorting approach then try to use heap

If k + smallest element – MAX HEAP

If k + largest element – MIN HEAP

MAX HEAP – Largest element is always at top

MIN HEAP – Smallest element is always at top

Use like a stack with k size

Sorting time complexity – **nlogn**

Heap time complexity – **nlogk**

Max Heap Working: arr[ ] = {2, 3, 34, 21, 1} , n=5,   let k=3

```
|       |        |       |       |        |       |        |    | 34   |
|       |        |       |       |        |       |  34    |    | 21   |
|       |  ->    |       |  ->   |  34    |  ->   |  21    | -> |  3   |
|       |        |   3   |       |   3    |       |   3    |    |  2   |
|   2   |        |   2   |       |   2    |       |   2    |    |  1   |
_______          _______         ________         ________      _______
```

QUESTIONS:

1. Kth Smallest Element.
2. Return K largest Elements in array.
3. Sort a K Sorted Array or Sort Nearly Sorted Array.
4. K Closest Numbers
5. Top K Frequent Numbers
6. Frequency sort
7. K Closest Points to Origin
8. Connect Ropes to Minimise the Cost
9. Sum of Elements between k1 smallest and k2 smallest

Kth Smallest Element

10 Given an array and a number k where k is smaller than the size of the array, we need to find the k'th smallest element in the given array. It is given that all array elements are distinct.

Examples:

Input: arr[] = {7, 10, 4, 3, 20, 15}, k = 3

Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}, k = 4

Output: 10

Method 1 (Simple Sorting Solution)

A simple solution is to sort the given array using an O(N log N) sorting algorithm like Merge Sort, Heap Sort, etc, and return the element at index k-1 in the sorted array.

The Time Complexity of this solution is **O(N log N)**

```
// Simple C++ program to find k'th smallest element
#include <bits/stdc++.h>
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: Heap

```
priority_queue <int> maxheap;
for(int i=0; i<n; i++)
{
        maxheap.push( arr[ i ] );

        if (maxheap.size() > k )
        {
                maxheap.pop();
        }
}
```

Time: O(nlogk)

Space: O(k+1)

Return K largest Elements in array.

Given an 1D integer array A of size N you have to find and return the B largest elements of the array A.

Input 1:

A = [11, 3, 4]

B = 2

Output 1:

[11, 4]

Input 2:

A = [11, 3, 4, 6]

B = 3

Output 2:

[4, 6, 11]

1. **Use sorting**

   Time: O(nlogn), then traverse accordingly and print it.

2. **Use heap**

```
#include <bits/stdc++.h>
using namespace std;

int main() {

   int k = 3;
   vector <int> arr = {11, 3, 4, 6};
   // op should be -> 11 6 4
   int size = arr.size();
```

```
    // Make a min heap
    priority_queue<int,vector<int>,greater<int>> minHeap;

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

       minHeap.push(arr[i]);

       if( minHeap.size() > k){
          minHeap.pop();
       }
    }

    // Now only k largest element are in heap

    vector<int> ans;

    while(minHeap.size() > 0 ){

       ans.push_back(minHeap.top());

       minHeap.pop();
    }

    for(auto it:ans){
       cout << it << " ";
    }

    return 0;
  }
```

Output:

4 6 11

Sort a K Sorted Array or Sort Nearly Sorted Array.

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. For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Example:

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

k = 3

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

1. Use **sort** function -> O(nlogn) time complexity

2. **Use Heap**

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
   int k = 3;
   vector <int> arr = {6, 5, 3, 2, 8, 10, 9};
   // op-> {2, 3, 5, 6, 8, 9, 10}

   int size = arr.size();
   vector <int> ans;
```

```
    // craete a min heap
    priority_queue <int, vector<int>, greater<int>> minHeap;

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

        minHeap.push(arr[i]);

        if(minHeap.size() > k){
            ans.push_back(minHeap.top());
            minHeap.pop();
        }
    }
    // since some elements may remain in heap so we have to traverse that also
    while(minHeap.size() > 0){
        ans.push_back(minHeap.top());
        minHeap.pop();
    }

    for(auto it:ans){
        cout << it << " ";
    }

    return 0;
}
```

Output:

2 3 5 6 8 9 10

K Closest Numbers

Given an unsorted array and two numbers x and k, find k closest values to x.

Input : arr[] = {10, 2, 14, 4, 7, 6}, x = 5, k = 3 .

Output: {7, 6, 4}

K is given so heap -> don't insert array values directly in heap

	10	2	14	4	7	6	
---	5	5	5	5	5	5	
Abs->	5	3	9	**1**	**2**	**1**	-> key to insert in heap

So return 4,7,6

1. **Problem statement**
   X=5 and k=3
   Three numbers which are closest to 5 are 6, 7, 8

   Similar to no of k largest elements in array
   But in this we need smallest

2. **How to insert values**

   Key, arr[i]

   Pair<int, int>

3. **Which Heap?**

   Closest -> take keys which are minimum -> take abs as 0, 1,2 rather than 8
   MAX HEAP

4. **Code**

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
   int x = 5, k=3;
   vector <int> arr = {10, 2, 14, 4, 7, 6};
   // op-> {4,7,6}

   int size = arr.size();
   vector <int> ans;

   // craete a max heap with pair
   priority_queue <pair<int, int>> maxHeap;

   for(int i=0; i<size; i++){
      maxHeap.push({abs(arr[i]-x), arr[i]});

      if(maxHeap.size() > k){
         maxHeap.pop();
      }
```

```
    }

    while (maxHeap.size() > 0){
        ans.push_back(maxHeap.top().second);
        maxHeap.pop();
    }

    for(auto it:ans){
        cout << it << " ";
    }

    return 0;
}
```

Output: {7, 6, 4}

Top K Frequent Numbers

Given an array of n numbers. Your task is to read numbers from the array and keep at-most K numbers at the top (According to their decreasing frequency) every time a new number is read. We basically need to print top k numbers sorted by frequency when input stream has included k distinct elements, else need to print all distinct elements sorted by frequency.

Example:

Input : arr[] = {1, 1, 1, 2, 2, 3}

k = 2

Output : {1, 2}

create a map;

elem	freq
1	3
2	2
3	1

1. **Which Heap**

   Since we need max freq so we will use **min heap** so that min element is on top and can be popped

2. **Insert in heap**

   We will insert freq of array elements

3. **Hashing**
   To insert freq of element we first need to create a map
   create a map;

elem	freq
1	3
2	2
3	1

CODE:

```
typedef pair<int, int> pii;
class Solution {
public:
   vector<int> topKFrequent(vector<int>& nums, int k) {
      int n = nums.size();
      vector <int> ans;

      unordered_map<int, int> mpp;

      for(int i=0; i<n; i++){
         mpp[nums[i]]++;
      }

      // craete a min heap with pair
      priority_queue <pii, vector<pii>, greater<pii>> minHeap;

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

```
            minHeap.push({it->second, it->first});

            if(minHeap.size() > k){
                minHeap.pop();
            }
        }

        while(minHeap.size() > 0){
            pair<int, int> tempPair = minHeap.top();
            ans.push_back(tempPair.second);
            minHeap.pop();
        }
        return ans;
    }
};
```

Frequency Sort

Print the elements of an array in the increasing frequency if 2 numbers have same frequency then print the one which came first.

Example:

Input : arr[] = {2, 5, 2, 8, 5, 6, 8, 8}

Output : arr[] = {8, 8, 8, 2, 2, 5, 5, 6}

Output : arr[] = {6, 5, 5, 2, 2, 8, 8, 8}

Put array elements in map with freq

Element	frequency
6	1
5	2
2	2
8	3

Traverse the map and put it in heap -> freq as key

Select min/max heap according to question

MIN heap

(freq, eement)

(1, 6)
(2, 5)
(2, 2)
(3, 8)
(freq, ele)

Now traverse the heap ans store **minheap.top().second** one by one in vector with while loop (freq>0)

CODE:

```
typedef pair<int, int> pii;


class Solution {
public:
   vector<int> topKFrequent(vector<int>& nums, int k) {
      int n = nums.size();
      vector <int> ans;

      unordered_map<int, int> mpp;

      for(int i=0; i<n; i++){
         mpp[nums[i]]++;
      }

      // craete a min heap with pair
      priority_queue <pii, vector<pii>, greater<pii>> minHeap;

      for(auto it=mpp.begin(); it!=mpp.end(); it++){
         minHeap.push({it->second, it->first});
      }

      while(minHeap.size() > 0){
               pair<int, int> tempPair;
               tempPair = minHeap.top();
               int element = tempPair.second;
```

```
                int frequency = tempPair.first;

                while(frequency > 0){
                        ans.push_back(element);
                         frequency--;
                } //end of inside while loop
                minHeap.pop();
        }  // end of main while loop

        return ans;
    }
};
```

K Closest Points to Origin

Given a list of points on the 2-D plane and an integer K. The task is to find K closest points to the origin and print them.

Note: The distance between two points on a plane is the Euclidean distance.

Example:

Input: point = [[3, 3], [5, -1], [-2, 4]], K = 2 .

Output: {}

Input: points = [[1,3],[-2,2]], k = 1

The way to judge whether a point is close or not is to find the euclidean distance.

we dont have to perform square root and make the code look complex moreover it's more prone to errors . we can simply just use `dist = x*x+y*y`

Now let's calculate the distance:

dist = 1*1 + 3*3 = 10

dist = -2*-2 + 2*2 = 8

We can see that dist of the 2nd element is smaller so we need to pushback that element into our result vector.

Algorithm:-

1. Initialize a priority queue for storing the maxheap data. let's name this as maxHeap and initialize a result vector .

2. let x and y be the co-ordinates of point p .

3. Main logic behind maxHeap is that, we will maintain a maxHeap of size k Thus after addding new points to our maxHeap we need to check the size of heap if it is greater than k or not .If the size is greater than k we will remove remove the root element to ensure the size of the heap is always k . Thus, the max heap is always maintain top K smallest elements from the first one to current one.

4. In short The maxheap will show true potential once the size of the heap is over its maximum capacity i.e it will exclude the maximum element in it as it can not be the proper candidate anymore.

CODE:

```
class Solution {
public:
   vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {
      vector<vector<int>> result(k);
      //maxheap
      priority_queue<vector<int>> maxHeap;
```

```
        //Construction of maxheap
        for (auto& p : points) {
            int x = p[0], y = p[1];
            maxHeap.push({x*x + y*y, x, y});
            if (maxHeap.size() > k) {
                maxHeap.pop();
            }
        }

        for (int i = 0; i < k; i++) {
            vector<int> top = maxHeap.top();
            maxHeap.pop();
            result[i] = {top[1], top[2]};
        }
        return result;
    }
};
```

Sorting:

/*We have a list of points on the plane. Find the K closest points to the origin (0, 0).

(Here, the distance between two points on a plane is the Euclidean distance.)

You may return the answer in any order. The answer is guaranteed to be unique (except for the order that it is in.) */

```
class Solution {
public:
   vector<vector<int>> kClosest(vector<vector<int>>& points, int K) {
      vector<pair<double,int>> vec;
      for(int i=0;i<points.size();i++){//for loop for calculating distances and pushing it back in the vector
         double distance = std::sqrt(std::pow(points[i][0], 2)+std::pow(points[i][1], 2));
         vec.push_back(make_pair(distance,i));
      }
      sort(vec.begin(), vec.end());//sorting the vector according to distances
      vector<vector<int>> f;//final points vector
      for(int i=0;i<K;i++){
         f.push_back(points[vec[i].second]);//pushing in the points accordingly
      }
      return f;
   }
};
```

Connect Ropes to Minimise the 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. We need to connect the ropes with minimum cost.

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. .

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.
    - Extract the minimum and second minimum from min-heap
    - Add the above two extracted values and insert the added value to the min-heap.
    - Maintain a variable for total cost and keep incrementing it by the sum of extracted values.
3. Return the value of this total cost.
4. Time: O(nlogn) , space: O(n)

CODE:

```
#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;
}
```