Binary Heap Implementation

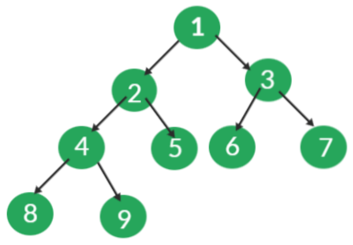
Binary Heap:

A Binary Heap is a Binary Tree that satisfies the following conditions.

- It should be a Complete Binary Tree.
- It should satisfy the Heap property.

Complete Binary Tree:

The tree in Which all the levels are completely filled except the last level and last level is filled in such a way that all the keys are as left as possible.

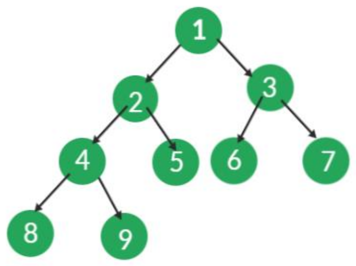


Complete Binary Tree

Heap Property:

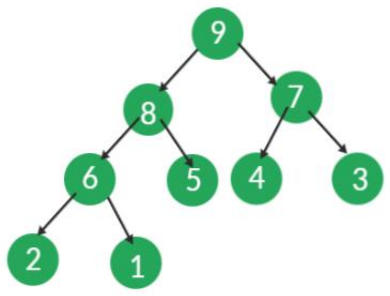
Binary Heap is either a Min Heap or Max Heap. Property of the Binary Heap decides whether it is Min Heap or Max Heap.

- **Min Heap property:** For every node in a binary heap, if the node value is less than its right and left child's value then Binary Heap is known as Min Heap. The property of Node's value less than its children's value is known as Min Heap property. In Min Heap, the root value is always the Minimum value in Heap.



Min Heap

- **Max Heap property:** For every node in a binary heap, if the node value is greater than its right and left child's value then Binary Heap is known as Max Heap. The property of being Node's value greater than its children's value is known as Max Heap property. In Max Heap, the root value is always the maximum value in Heap.



Max Heap

Representation of the Binary Heap:

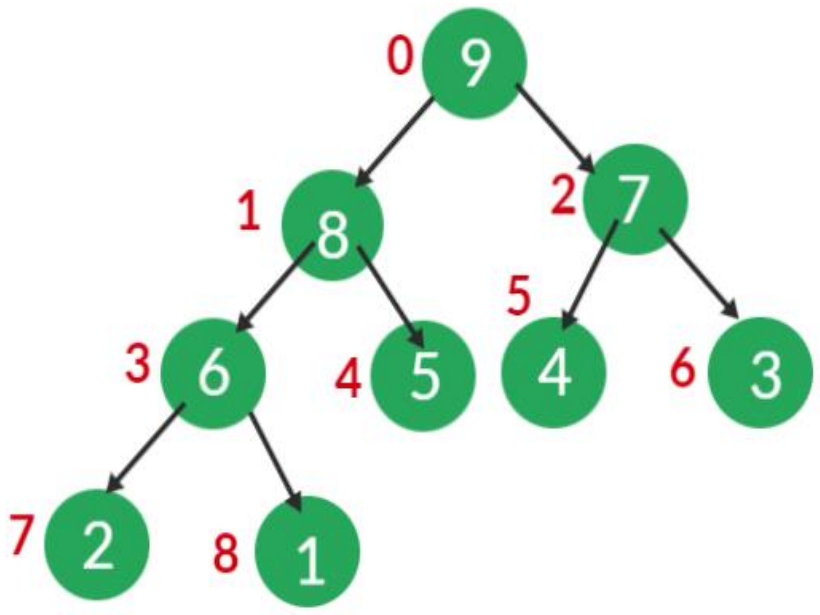
A Binary Heap is represented as an array.

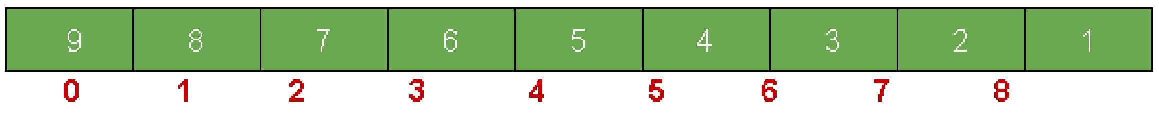
- The root value is at arr[0]
- The below table summarizes how the node and its children are stored in an array.

• **Node index**	• **Left child Index**	• **Right Child Index**
• i	• 2*i+1	• 2*i+2

- If the child is at index **i**, then its parent index can be found using the below formula.

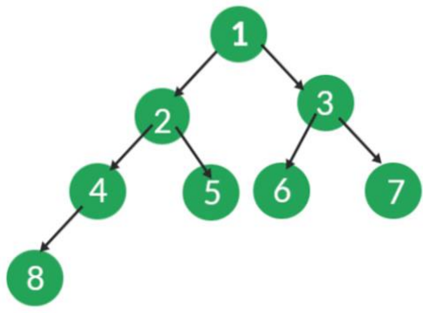
• **Child's Index**	• **Parent's Index**
• i	• (i-1)/2

- 

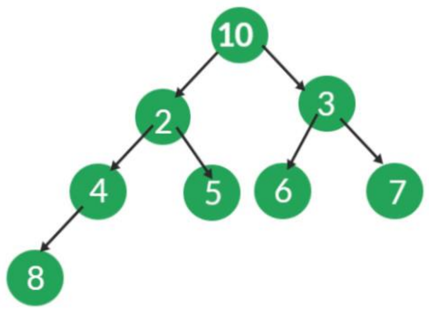


Array Representation of Binary Heap.

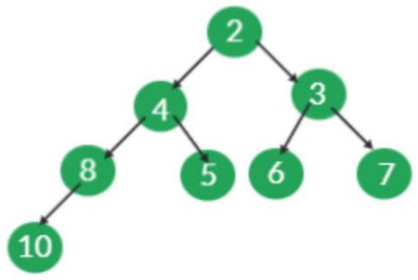
-
- **Operations Associated with Min Heap:**
- Insert()
- Heapify()
- getMin()
- ExtractMin()
- Delete()
- Decreasekey()
- Let's see how these operations are done Using Min Heap.
- **Note:** The Binary heap should always be a complete binary tree and should satisfy the corresponding heap property (Min / Max). If any of the two conditions are disturbed we should make necessary modifications in a heap to satisfy the two conditions.
- **Insert():**
- Insert operation inserts a new key in the Binary Heap.
- **Steps Followed for inserting the key in Binary Heap:**
- First Insert the key at the first vacant position from the left on the last level of the heap. IF the last level is completely filled, then insert the key as the left-most element in the next level.
- Inserting a new key at the first vacant position in the last level preserves the Complete binary tree property, The Min heap property may get affected we need to check for it.
- If the inserted key parent is less than the key, then the Min heap property is also not violated.
- If the parent is greater than the inserted key value, then swap the values. Now the heap property may get violated at the parent node. So repeat the same process until the heap property is satisfied.
- Inserting an element takes O(logN) Time Complexity. Below shows the animation of how -1 is inserted into a Binary heap by following above described steps.



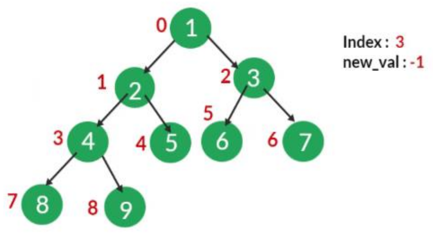
- **Heapify():**
- Suppose there exists a Node at some index i, where the Minheap property is Violated.
- We assume that all the subtrees of the tree rooted at index i are valid binary heaps.
- The Heapify function is used to restore the Minheap, by fixing the violation.
- **Steps to be followed for Heapify:**
- First find out the Minimum among the Violated Node, left, and right child of Violated Node.
- If the Minimum among them is the left child, then swap the Violated Node value with the Left child value and recursively call the function on the left Child.
- If the Minimum among them is the right child, then swap the Violated Node value with the right child value and recursively call the function right Child.
- Recursive call stops when the heap property is not violated.



- **getMin():**
- It returns the minimum value in the Binary Heap.
- As we all know, the root Node is the Minimum value in Min Heap. Simply return the arr[0].
- **ExtractMin():**
- This removes the Minimum element from the heap.
- **Steps to be followed to Remove Minimum value/root Node:**

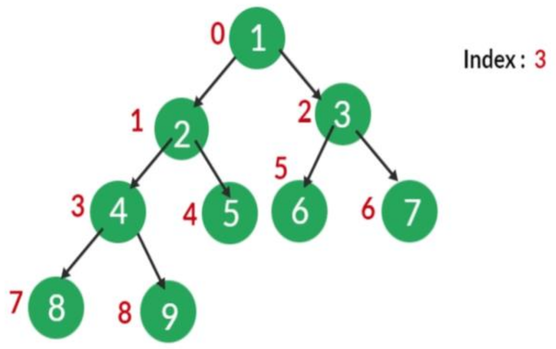
- Copy the last Node value to the Root Node, and decrease the size, this means that the root value is now deleted from the heap, and the size is decreased by 1.
- By doing the above step we ensure that the Complete binary tree property is not violated, as we are copying the last node value to the root node value, the Min Heap property gets violated.
- Call the Heapify function to convert it into a valid heap.
- 

- **Decreasekey():**
- Given an index and a value, we need to update the value at the index with the given value. We assume that the given value is less than the existing value at that index.
- **Steps to be followed for Decreasekey():**
- Let's the index be **i** and the value be **new_val.** Update existing value at index i with new_val i.e **arr[i] = new_val.**
- By performing the above step, the Complete binary tree property is not violated, but the Min heap property may get violated.
- As the new_val we are inserting is less than the previously existing value, the min-heap property is not violated in subtrees of this rooted tree.
- It may get violated in its ancestors, so as we do in insert operation, check the value of a current node with its parent node, if it violates the min-heap property
- Swap the nodes and recursively do the same.



-
- **Delete() :**
- Given an index, delete the value at that index from the min-heap.

- **Steps to be followed for Delete operation():**
- First, update the value at the index that needs to be deleted with INT_MIN.
- Now call the Decreasekey() function at the index which is need to be deleted. As the value at the index is the least, it reaches the top.
- Now call the ExtractMin() operation which deletes the root node in Minheap.
- In this way, the desired index value is deleted from the Minheap.



-

```
#include <bits/stdc++.h>


using namespace std;


class BinaryHeap {
 public:
  int capacity; /*Maximum elements that can be stored in heap*/
 int size; /*Current no of elements in heap*/
 int * arr; /*array for storing the keys*/


 BinaryHeap(int cap) {
  capacity = cap; /*Assigning the capacity*/
  size = 0; /*Intailly size of hepa is zero*/
  arr = new int[capacity]; /*Creating a array*/
```

```
}

/*returns the parent of ith Node*/
int parent(int i) {
  return (i - 1) / 2;
}
/*returns the left child of ith Node*/
int left(int i) {
  return 2 * i + 1;
}
/*Returns the right child of the ith Node*/
int right(int i) {
  return 2 * i + 2;
}

/*Insert a new key x*/
void Insert(int x) {
  if (size == capacity) {
    cout << "Binary Heap Overflwon" << endl;
    return;
  }
  arr[size] = x; /*Insert new element at end*/
  int k = size; /*store the index ,for checking heap property*/
  size++; /*Increase the size*/

  /*Fix the min heap property*/
  while (k != 0 && arr[parent(k)] > arr[k]) {
    swap( & arr[parent(k)], & arr[k]);
    k = parent(k);
```

```
 }
}

void Heapify(int ind) {
 int ri = right(ind); /*right child*/
 int li = left(ind); /*left child*/
 int smallest = ind; /*intially assume violated value in Min value*/
 if (li < size && arr[li] < arr[smallest])
  smallest = li;
 if (ri < size && arr[ri] < arr[smallest])
  smallest = ri;
 /*smallest will store the minvalue index*/
 /*If the Minimum among the three nodes is the parent itself,
 that is Heap property satisfied so stop else call function recursively on Minvalue node*/
 if (smallest != ind) {
  swap( & arr[ind], & arr[smallest]);
  Heapify(smallest);
 }
}
int getMin() {
 return arr[0];
}
int ExtractMin() {
 if (size <= 0)
  return INT_MAX;
 if (size == 1) {
  size--;
  return arr[0];
 }
```

```
  int mini = arr[0];
  arr[0] = arr[size - 1]; /*Copy last Node value to root Node value*/
  size--;
  Heapify(0); /*Call heapify on root node*/
  return mini;
 }
 void Decreasekey(int i, int val) {
  arr[i] = val; /*Updating the new_val*/
  while (i != 0 && arr[parent(i)] > arr[i]) /*Fixing the Min heap*/ {
   swap( & arr[parent(i)], & arr[i]);
   i = parent(i);
  }
 }
 void Delete(int i) {
  Decreasekey(i, INT_MIN);
  ExtractMin();
 }
 void swap(int * x, int * y) {
  int temp = * x;
  * x = * y;
  * y = temp;
 }
 void print() {
  for (int i = 0; i < size; i++)
   cout << arr[i] << " ";
  cout << endl;
 }
};
int main()
```

```
{
 BinaryHeap h(20);
 h.Insert(4);
 h.Insert(1);
 h.Insert(2);
 h.Insert(6);
 h.Insert(7);
 h.Insert(3);
 h.Insert(8);
 h.Insert(5);
 cout << "Min value is " << h.getMin() << endl;
 h.Insert(-1);
 cout << "Min value is " << h.getMin() << endl;
 h.Decreasekey(3, -2);
 cout << "Min value is " << h.getMin() << endl;
 h.ExtractMin();
 cout << "Min value is " << h.getMin() << endl;
 h.Delete(0);
 cout << "Min value is " << h.getMin() << endl;
 return 0;
}
```

Output:

```
Min value is 1
Min value is -1
Min value is -2
Min value is -1
Min value is 1
```

Time Complexities :

Function	Time Complexity
Insert() :	**O(logN)**
Heapify()	**O(logN)**
getMin()	**O(1)**
ExtractMin()	**O(logN)**
Decreasekey()	**O(logN)**
Delete()	**O(logN)**

----

1) Top K Frequent Elements: 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]

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    //Store it in a unordered map;
    unordered_map<int, int>mp;
    vector<int> ans;
    for(int i=0;i<nums.size();i++){
      mp[nums[i]]++;
    }
    priority_queue<pair<int, int>>pq;
    for(auto i:mp){
      pq.push({i.second, i.first});
    }
```

```
  while(k!=0){
   ans.push_back(pq.top().second);
   pq.pop();
   k--;
  }
return ans;
}
```

- Time complexity: It will be O(nlogn) becuase the operations in a max heap takes this much time.
- Space complexity: Worst case : O(n) if all the elements are present only one time.

  ----

  2) **Kth Largest Element in an Array**: 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

  Example 2:

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

  Output: 4

```
  int findKthLargest(vector<int>& a, int k) {
    priority_queue<int> p;
    for (int i = 0; i < a.size(); i++)
      p.push(a[i]);
    while (k != 1) {
      k--;
      p.pop();
    }
    return p.top();
  }
```

----

3) Ugly Number II: An **ugly number** is a positive integer whose prime factors are limited to 2, 3, and 5.

Given an integer n, return *the* nth ***ugly number***.

Example 1:

Input: n = 10

Output: 12

Explanation: [1, 2, 3, 4, 5, 6, 8, 9, 10, 12] is the sequence of the first 10 ugly numbers.

Example 2:

Input: n = 1

Output: 1

Explanation: 1 has no prime factors, therefore all of its prime factors are limited to 2, 3, and 5.

```
  int nthUglyNumber(int n) {
    priority_queue<long, vector<long>, greater<long>> pq;
    unordered_set<int> vis;
    pq.push(1);
    vis.insert(1);
    vector<int> factors = {2,3,5};
   long curr_ugly = 1;
    for(int i = 0; i < n; i++){
      curr_ugly = pq.top();
      pq.pop();

      for(int f: factors){
        long next_ugly = curr_ugly * f;
```

```
                if(vis.find(next_ugly) == vis.end()){
                    pq.push(next_ugly);
                    vis.insert(next_ugly);
                }
            }
        }
        return (int)curr_ugly;
    }
```

Time complexity: O(N*logn)

Space complexity: O(N)

----

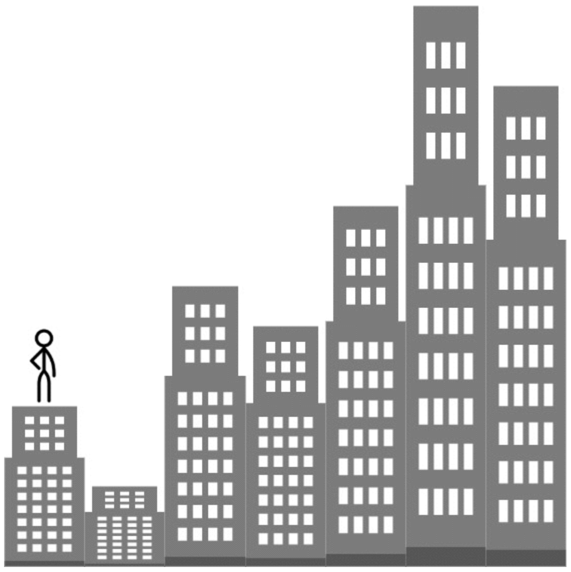
4) **Furthest Building You Can Reach**: You are given an integer array heights representing the heights of buildings, some bricks, and some ladders.

You start your journey from building 0 and move to the next building by possibly using bricks or ladders.

While moving from building i to building i+1 (**0-indexed**),

- If the current building's height is **greater than or equal** to the next building's height, you do **not** need a ladder or bricks.
- If the current building's height is **less than** the next building's height, you can either use **one ladder** or (h[i+1] - h[i]) **bricks**.

Return the furthest building index (0-indexed) you can reach if you use the given ladders and bricks optimally.



Input: heights = [4,2,7,6,9,14,12], bricks = 5, ladders = 1

Output: 4

Explanation: Starting at building 0, you can follow these steps:

- Go to building 1 without using ladders nor bricks since 4 >= 2.

- Go to building 2 using 5 bricks. You must use either bricks or ladders because 2 < 7.

- Go to building 3 without using ladders nor bricks since 7 >= 6.

- Go to building 4 using your only ladder. You must use either bricks or ladders because 6 < 9.

It is impossible to go beyond building 4 because you do not have any more bricks or ladders.

Example 2:

Input: heights = [4,12,2,7,3,18,20,3,19], bricks = 10, ladders = 2

Output: 7

Example 3:

Input: heights = [14,3,19,3], bricks = 17, ladders = 0

Output: 3

```
int furthestBuilding(vector<int>& heights, int bricks, int ladders) {
    priority_queue<int, vector<int>, greater<int>> pq;
    for (int i = 0; i < heights.size() - 1; ++i) {
        if (heights[i] < heights[i + 1]) {
            pq.push(heights[i + 1] - heights[i]);
            if (pq.size() > ladders) {
                int smallest = pq.top();
                pq.pop();
                if (bricks < smallest) {
                    return i;
                }
                bricks -= smallest;
            }
        }
    }
    return heights.size() - 1;
}
```

- **Time complexity: *O*(*nlogn*)**
- **Space complexity: *O*(*ladders*)**

--

5) Kth Smallest Element in a Sorted Matrix:

Given an n x n matrix where each of the rows and columns is sorted in ascending order, return *the* kth *smallest element in the matrix*.

Note that it is the kth smallest element in the sorted order, not the kth distinct element.

You must find a solution with a memory complexity better than O(n^2).

Example 1:

Input: matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8

Output: 13

Explanation: The elements in the matrix are [1,5,9,10,11,12,13,13,15], and the 8th smallest number is 13

Example 2:

Input: matrix = [[-5]], k = 1

Output: -5

```
int kthSmallest(vector<vector<int>>& matrix, int k) {
    priority_queue <int, vector<int>, greater<int>> pq ;
    for(int i=0;i<matrix.size();i++){
        for(int j=0;j<matrix.size();j++){
            pq.push(matrix[i][j]);
        }
    }
    while(k>1){
        pq.pop();
        k--;
    }
    return pq.top();
}
```

---

6) Reorganize String: 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: ""

```
string reorganizeString(string s) {
    priority_queue<pair<int,char>> pq; //max 26 size
    unordered_map<char,int> m;  //max 26 size
    for(auto it : s){
            m[it]++;
        }
        for(auto it : m){
            pq.push({it.second,it.first});
        }
        string ans = "";
        while(pq.size() > 1){
            auto f = pq.top();
            pq.pop();
            auto s = pq.top();
            pq.pop();
            ans+=f.second;
            ans+=s.second;
            if(--f.first > 0){
                pq.push(f);
            }
            if(--s.first > 0){
                pq.push(s);
            }
        }

        //If there are still elements present in pq then check if there is multiple elements of single
character
        if(!pq.empty()){
            if(pq.top().first > 1){
                return "";
            }
            else{
                ans+=pq.top().second;
```

```
    }
  }

  return ans;
}
```

- Time complexity: O(26*N)
- Space complexity: O(1)

----

7) **Find the Most Competitive Subsequence**: Given an integer array nums and a positive integer k, return *the most* ***competitive*** *subsequence of* nums *of size* k.

An array's subsequence is a resulting sequence obtained by erasing some (possibly zero) elements from the array.

We define that a subsequence a is more **competitive** than a subsequence b (of the same length) if in the first position where a and b differ, subsequence a has a number **less** than the corresponding number in b. For example, [1,3,4] is more competitive than [1,3,5] because the first position they differ is at the final number, and 4 is less than 5.

Example 1:

Input: nums = [3,5,2,6], k = 2

Output: [2,6]

Explanation: Among the set of every possible subsequence: {[3,5], [3,2], [3,6], [5,2], [5,6], [2,6]}, [2,6] is the most competitive.

Example 2:

Input: nums = [2,4,3,3,5,4,9,6], k = 4

Output: [2,3,3,4]

```
vector<int> mostCompetitive(vector<int>& nums, int k) {
  int n=nums.size();
  priority_queue<vector<int>,vector<vector<int>>,greater<vector<int>>>arr;
  int i=0;
  for(;i<n-k;i++){
    arr.push({nums[i],i});
  }
  // for(auto v:arr)cout<<v[0]<<"-"<<v[1]<<"  ";
  vector<int>ans;
  int ci=-1;
  for(;i<n;i++){
    arr.push({nums[i],i});
    while(arr.top()[1]<ci)arr.pop();
    ans.push_back(arr.top()[0]);
    ci=arr.top()[1];
    arr.pop();
  }
  return ans;
}
```

----

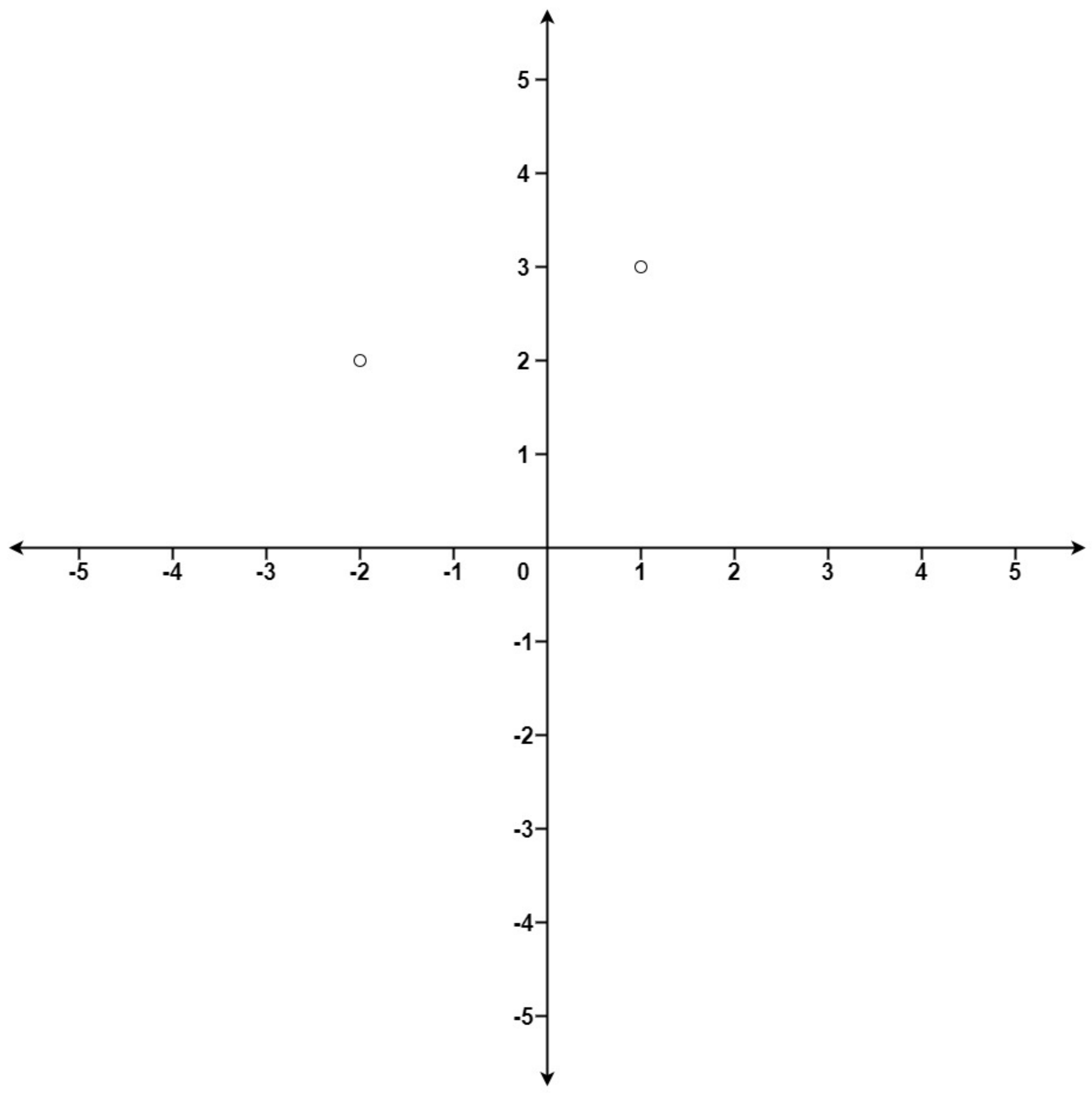
8) K Closest Points to Origin:

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) {
    //use maxheap here
    //pq->{dist,points[i]};
    priority_queue<pair<int,vector<int>>>maxh;
    for(int i=0;i<points.size();i++){
      int dist=points[i][0]*points[i][0]+points[i][1]*points[i][1];
      maxh.push({dist,points[i]});
      if(maxh.size()>k){
        maxh.pop();
      }
    }
    vector<vector<int>>ans;
    while(!maxh.empty()){
      pair<int,vector<int>>node=maxh.top();
      ans.push_back(node.second);
      maxh.pop();
    }
    return ans;


  }
```

- Time complexity:
  O(NlogK)

----

9) Maximum Sum Combination:

Given two equally sized 1-D arrays **A, B** containing **N** integers each.

A **sum combination** is made by adding one element from array **A** and another element of array **B**.

Return the **maximum C valid sum combinations** from all the possible sum combinations.

Example Input

Input 1:

A = [3, 2]

B = [1, 4]

C = 2

Input 2:

A = [1, 4, 2, 3]

B = [2, 5, 1, 6]

C = 4

Example Output

Output 1:

[7, 6]

Output 1:

[10, 9, 9, 8]

Example Explanation

Explanation 1:

7   (A : 3) + (B : 4)

6   (A : 2) + (B : 4)

Explanation 2:

10   (A : 4) + (B : 6)

9   (A : 4) + (B : 5)

9   (A : 3) + (B : 6)

8   (A : 3) + (B : 5)

```
int* solve(int* A, int n1, int* B, int n2, int C, int *len1) {
```

```
sort(A.begin(), A.end(), greater<int>());

sort(B.begin(), B.end(), greater<int>());

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

int N = A.size();

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

{

  pq.push(A[i] + B[i]);

}

vector<int> ans(C);

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

{

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

  {

    if(i==j)

    continue;

    if(A[i] + B[j] > pq.top())

    {

      pq.pop();

      pq.push(A[i] + B[j]);

    }

    else

    break;

  }

}

for(int i=C-1; i>=0; i--)

{

  ans[i] = pq.top();

  pq.pop();

}
```

```
  return ans;
}
```

--------------------------Hard--

[Swim in Rising Water](#):

[Minimum Cost to Hire K Workers](#):

--

[Minimum Number of Refueling Stops](#):

[Merge k Sorted Lists](#):

[Find Median from Data Stream](#):