K Smallest In Unsorted Array (Medium)

Find the K smallest numbers in an unsorted integer array A. The returned numbers should be in ascending order.

Assumptions

- A is not null
- K is >= 0 and smaller than or equal to size of A

Return

- an array with size K containing the K smallest numbers in ascending order

Examples

- A = {3, 4, 1, 2, 5}, K = 3, the 3 smallest numbers are {1, 2, 3}

```
// return smallest k elements from an unsorted array
vector<int> kSmallest(vector<int> arr, int k){
    // define a min heap
    priority_queue <int, vector<int>, greater<int>> pq;
    vector<int> result;
    // build a min heap of n elements, it takes O(n) time
    for(int i = 0; i < arr.size(); i++)
        pq.push(arr[i]);
    // output the top element and pop it, it takes O(klogn) time
    for(int i = 0; i < k; i++){
        result.push_back(pq.top());
        pq.pop();
    }
    return result;
}
```

Time Complexity: O(n + klogn).

//Ok, no problem!

Top K Frequent Words (Medium)

Given a composition with different kinds of words, return a list of the top K most frequent words in the composition.

Assumptions

- the composition is not null and is not guaranteed to be sorted
- K >= 1 and K could be larger than the number of distinct words in the composition, in this case, just return all the distinct words

Return

- a list of words ordered from most frequent one to least frequent one (the list could be of size K or smaller than K)

Examples

- Composition = ["a", "a", "b", "b", "b", "b", "c", "c", "c", "d"], top 2 frequent words are [“b”, “c”]
- Composition = ["a", "a", "b", "b", "b", "b", "c", "c", "c", "d"], top 4 frequent words are [“b”, “c”, "a", "d"]
- Composition = ["a", "a", "b", "b", "b", "b", "c", "c", "c", "d"], top 5 frequent words are [“b”, “c”, "a", "d"]

```
// return a list of the top K most frequent words
vector<string> kFrequentWords(vector<string> words, int k){
    vector<string> result;
    unordered_map<string, int> wordFreq;
    // store the frequency of words, it takes O(n) time
    for(int i = 0; i < words.size(); i++){
        wordFreq[words[i]]++;
    }
    // define the comparasion rules in the heap
    struct MyCmp {
        bool operator() (const pair<string, int>& a, const pair<string, int>& b){
            if(a.second != b.second)
                return a.second < b.second;
            else
                return a.first > b.first;
        }
    };
    priority_queue<pair<string, int>, vector<pair<string, int>>, MyCmp> pq;
    // store words and their freqency in heap, it takes O(n) time
    for(auto& it : wordFreq){
        pq.push(make_pair(it.first, it.second));
```

```
    }
    // output the top element and pop it, it takes O(klogn) time
  for(int i = 0; i < k; i++){
      //if (pq.empty()) break;
        result.push_back(pq.top().first);
        pq.pop();
    }
    return result;
}
```

Time Complexity: O(n + n + klogn) = O(n + klogn)
//Please read the statement carefully! The second assumption. The method is right!

Median Tracker (Medium)

Given an unlimited flow of numbers, keep track of the median of all elements seen so far.

You will have to implement the following two methods for the class

- read(int value) - read one value from the flow
- median() - return the median at any time, return null if there is no value read so far

Examples

- read(1), median is 1
- read(2), median is 1.5
- read(3), median is 2
- read(10), median is 2.5
- ……

```
// two heap to keep tracking median
class MedianFinder{
    priority_queue<int> lo; // max heap
    priority_queue<int, vector<int>, greater<int>> hi; // min heap
public:
    void read(int value){
        lo.push(value);
        hi.push(lo.top());
        lo.pop();
        if(lo.size() < hi.size()){
            lo.push(hi.top());
            hi.pop();
        }
    }

   double median(){
       //if (lo.empty() && hi.empty()) return NULL;
        if(lo.size() == hi.size())
            return (lo.top() + hi.top()) * 0.5;
        else
            return lo.top();
    }
};
```

Time Complexity: O(logn) for read(), since both push and pop operation take O(logn) time; O(1) for median().

//Ok, please read the statement carefully!

Walls and gates (Medium)

You are given a *m x n* 2D grid initialized with these three possible values.

1. −1 - A wall or an obstacle.
2. 0 - A gate.
3. INF - Infinity means an empty room. We use the value $2^{31} - 1 = 2147483647$ to represent INF as you may assume that the distance to a gate is less than 2147483647.

Fill each empty room with the distance to its *nearest* gate. If it is impossible to reach a gate, it should be filled with INF.

For example, given the 2D grid:

```
INF  −1  0  INF
INF INF INF  −1
INF  −1 INF  −1
  0  −1 INF INF
```

After running your function, the 2D grid should be:

```
  3  −1   0   1
  2   2   1  −1
  1  −1   2  −1
  0  −1   3   4
```

```cpp
const int GATE = 0;
const int EMPTY = 2147483647;
const int WALL = -1;
vector< vector<int> > directions = {{0, 1}, {0, -1}, {-1, 0}, {1, 0}}; // up, down, left, right
void wallsAndGates(vector< vector<int> >& rooms){
    queue< vector<int> > q;
    // find all the gate locations in mxn matrix, and push to the queue
    int m = rooms.size();
    if(m == 0) return;
    int n = rooms[0].size();
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            if(rooms[i][j] == GATE)
                q.push({i ,j});
        }
    }
    while(q.size()){
        vector<int> first = q.front();
```

```
        q.pop();
        int row = first[0];
        int col = first[1];
        for(auto& direction : directions){
            int r = row + direction[0];
            int c = col + direction[1];
            if(r < 0 || r >= m || c < 0 || c >= n || rooms[r][c] != EMPTY)
                continue;
            rooms[r][c] = rooms[row][col] + 1;
            q.push({r, c});
        }
    }
}
```

Time Complexity: O(mn), same as BFS algorithm, we traverse all the elements in the 2d grid except Wall (-1).
//OK, great!

Place To Put The Chair I (Hard)

Given a gym with k pieces of equipment and some obstacles. We bought a chair and wanted to put this chair into the gym such that the sum of the shortest path cost from the chair to the k pieces of equipment is minimal. The gym is represented by a char matrix, ‘E ’denotes a cell with equipment, ‘O ’denotes a cell with an obstacle, 'C' denotes a cell without any equipment or obstacle. You can only move to neighboring cells (left, right, up, down) if the neighboring cell is not an obstacle. The cost of moving from one cell to its neighbor is 1. You can not put the chair on a cell with equipment or obstacle.

Assumptions

- There is at least one equipment in the gym
- The given gym is represented by a char matrix of size M * N, where M >= 1 and N >= 1, it is guaranteed to be not null
- It is guaranteed that each 'C' cell is reachable from all 'E' cells.
- If there does not exist such place to put the chair, just return {-1, -1}

Examples

```
{ { 'E', 'O', 'C' },
  {  'C', 'E',  'C' },
  {  'C',  'C',  'C' } }
```

we should put the chair at (1, 0), so that the sum of cost from the chair to the two equipment is 1 + 1 = 2, which is minimal.

```
// compute sum of two 2d matrix, store the result in a, it takes O(mn) time.
void vectorSum(vector< vector<int> >& a, vector< vector<int> > b) {
    int m = a.size();
    int n = a[0].size();
    vector< vector<int> > result(m, vector<int> (n, 0));
    for(int r = 0; r < m; r++){
        for(int c = 0; c < n; c++){
            a[r][c] += b[r][c];
        }
    }
}

// return the location of minimum value in 2d matrix, it takes O(mn) time.
vector<int> vectorMin(vector< vector<int> > a){
    int m = a.size();
    int n = a[0].size();
    int row = -1;
    int col = -1;
    int min = 10000000;//int min = 0x3f3f3f3f;
    for(int r = 0; r < m; r++){
        for(int c = 0; c < n; c++){
            if(a[r][c] > 0 && a[r][c] <= min){
                min = a[r][c];
                row = r;
                col = c;
            }
        }
    }
    return {row, col};
}

// return distance from a 'E' to each 'C' in the matrix, it takes O(mn) time.
vector< vector<int> > distanceFromEquipment(vector< vector<char> > gym,
vector<int> eLoc){
    int m = gym.size();
    int n = gym[0].size();
    vector< vector<int> > count(m, vector<int> (n, 0)); // initialize a mxn zero
matrix
    vector< vector<int> > test(m, vector<int> (n, 0));
    queue< vector<int> > q;
    q.push(eLoc);
    while(!q.empty()){
        vector<int> first = q.front();
        q.pop();
        int row = first[0];
        int col = first[1];
        for(auto& direction : directions){
            int r = row + direction[0];
            int c = col + direction[1];
            if(r < 0 || r >= m || c < 0 || c >= n || gym[r][c] != 'C' ||
count[r][c] != 0)
```

```
                continue;
            count[r][c] = count[row][col] + 1;
            q.push({r, c});
        }
    }
    return count;
}

vector<int> putChair(vector< vector<char> > gym){
    vector< vector<int> > eLocation;
    int m = gym.size();
    int n = gym[0].size();
    vector< vector<int> > countSum(m, vector<int> (n, 0));
    // find the location of all E
    for(int r = 0; r < m; r++){
        for(int c = 0; c < n; c++){
            if(gym[r][c] == 'E') {
                vector<vector<int> > count = distanceFromEquipment(gym, {r, c});
                vectorSum(countSum, count);
            }
        }
    }
    return vectorMin(countSum);
}
```

Time Complexity: O(kmn), k is the number of equipment in the gym.

//Ok, very good you can solve this problem! I have some suggestions, first is to use a pair<int, int> to store a 2d index. The aim is to save space and save time(vector is a little slow). The second is to use 0x3f3f3f3f to imply a infinite value. It's very large but not exceed max int. This is a common skill in competitive programming.