PriorityQueue

378. Kth Smallest Element in a Sorted Matrix

Given a n x n matrix where each of the rows and columns are sorted in ascending order, find the kth smallest element in the matrix.

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

matrix = [

[ 1, 5, 9],

[10, 11, 13],

[12, 13, 15]], k = 8, return 13.

**S1. PriorityQueue**

先把第一行的每个都加进去，然后再看哪个最小，就加那一列的下一个，

在第二个for循环中一共取出了k-1个最小的值，此时peek()就是kth smallest element.

public class Solution {

public int kthSmallest(int[][] matrix, int k) {

int n = matrix[0].length;

PriorityQueue<Tuple> pq = new PriorityQueue<Tuple>();

for(int j = 0; j <= n-1; j++) pq.offer(new Tuple(0, j, matrix[0][j]));

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

Tuple t = pq.poll();

if(t.x == n-1) continue;

pq.offer(new Tuple(t.x+1, t.y, matrix[t.x+1][t.y]));

}

return pq.poll().val;

}

}

class Tuple implements Comparable<Tuple> {

int x, y, val;

public Tuple (int x, int y, int val) {

this.x = x;

this.y = y;

this.val = val;

}

@Override

public int compareTo (Tuple that) {

return this.val - that.val;

}

}

Houzz – find min time

告诉你火车时刻表，有三个站，每个时刻

3

westwood， santamonica， lax

westwood santamonica 1000 1050

santamonica lax 1100 1230

westwood lax 1130 1250

告诉你 westwood 开始 lax 是到达站

要求 1.最早时间到达 2.如果同时到达的话 希望出发时间最晚

public static int[] findMinTime(String[] stations, String[] inputs) {

if (inputs == null || inputs.length == 0) return null;

int n = stations.length;

PriorityQueue<Element> pqueue = new PriorityQueue<>((a, b)->(a.end - b.end));

List<Element> list = new ArrayList<>();

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

String[] splits = inputs[i].split(" ");

Element element = new Element(i, splits[0], splits[1],

Integer.parseInt(splits[2]), Integer.parseInt(splits[3]));

if (element.endName.equals(stations[n-1])) pqueue.add(element);

if (element.endName.equals(stations[n-2])) list.add(element);

}

int[] res = new int[2];

int startTime = -1;

while (!pqueue.isEmpty()) {

Element cur = pqueue.poll();

res[1] = cur.end;

if (cur.startName.equals(stations[0])) { *// A->C directly*

startTime = cur.start;

res[0] = cur.start;

}

for (Element e:list) { *// check all A->B conditions*

if (e.end > cur.start) continue;

if (e.start > startTime) {

startTime = e.start;

res[0] = e.start;

}

}

if (startTime > 0 &&

(pqueue.isEmpty() || pqueue.peek().end > cur.end)) return res;

}

return null;

}

class Element {

int id;

String startName;

String endName;

int start;

int end;

Element (int id, String startName, String endName, int start, int end) {

this.id = id;

this.startName = startName;

this.endName = endName;

this.start = start;

this.end = end;

}

}

PocketGems – 口袋装宝石的经典题目

<http://www.1point3acres.com/bbs/thread-217589-1-1.html>

有n个口袋，每个口袋能装多少宝石要取决于宝石的种类；每种宝石有固定的价格。

输入：1.发现的宝石，比如[“diamond”, “ruby”, “armor”, “diamond”, “diamond”, “ruby”, “diamond”, “diamond”, “diamond”, “diamond”, “diamond” “armor”]

2. 宝石的信息：

item\_infos:[

{

name=”diamond”

value=10

maximum\_stack\_size=5. 一个口袋最多装5块

},

{. name=”ruby

value=5

maximum\_stack\_size=5.

}

{

name=”armor”

value=25

maximum\_stack\_size=1

}]

3.口袋的数目n

返回这些口袋可以装的下的宝石的最大的价值。

Segment Tree

介绍看这里：

<http://www.cnblogs.com/wuyudong/p/segment-tree.html>

常见的操作就是buildTree, update, 以及query。重点是update会同时涉及到其他点的改动

适合的场景：

* 查找在某个区间内min/max/count的值
* 频繁update – 可以做到O(logn)
* 频繁query – 可以做到O(logn)

307. Range Sum Query - Mutable

Given an integer array nums, find the sum of the elements between indices i and j (i ≤ j), inclusive.

The update(i, val) function modifies nums by updating the element at index i to val.

Example: Given nums = [1, 3, 5]

sumRange(0, 2) -> 9

update(1, 2)

sumRange(0, 2) -> 8

**S1. Array**

用数组存[0, i]的sum，然后sumRange的时候相减，O(1)

update()时更新所有相关项，O(n),

**S2. Array**

和S1 反过来，update时候只更新相关项，O(1)

sumRange的时候加一遍，O(n)

**S3. Segment tree**,

update, query都可以做到O(logn)

public class NumArray {

SegmentTreeNode root;

*// build a Segment tree*

public NumArray(int[] nums) {

root = buildTree(nums, 0, nums.length - 1);

}

private SegmentTreeNode buildTree(int[] nums, int start, int end) {

if (nums == null || nums.length == 0) return null;

if (start == end) return new SegmentTreeNode(start, end, nums[start]);

int mid = start + (end - start)/2;

SegmentTreeNode left = buildTree(nums, start, mid);

SegmentTreeNode right = buildTree(nums, mid + 1, end);

SegmentTreeNode root = new SegmentTreeNode(start, end, left.sum + right.sum);

root.left = left;

root.right = right;

return root;

}

*// O(log n)*

public void update(int i, int val) {

updateHelper(root, i, val);

}

private void updateHelper(SegmentTreeNode root, int i, int val) {

if (root == null || i < root.start || i > root.end) return;

if (root.start == i && root.end == i) {

root.sum = val;

return;

}

updateHelper(root.left, i, val);

updateHelper(root.right, i, val);

root.sum = root.left.sum + root.right.sum;

}

*// O(log n)*

public int sumRange(int i, int j) {

return sumHelper(root, i, j);

}

private int sumHelper(SegmentTreeNode root, int i, int j) {

*// 这一步只能保证[i, j] 与 [root.start, root.end]之间有重叠，保证部分valid*

*// 但仍有可能[i, j]并不是全部valid, 所以通过max/min提取出有效的部分*

if (root == null || root.start > j || root.end < i) return 0;

i = Math.max(i, root.start); *// to make sure range is valid*

j = Math.min(j, root.end);

if (root.start == i && root.end == j) return root.sum;

int left = sumHelper(root.left, i, j);

int right = sumHelper(root.right, i, j);

return left + right;

}

}

class SegmentTreeNode {

int start, end, sum;

SegmentTreeNode left, right;

public SegmentTreeNode(){};

public SegmentTreeNode(int start, int end, int sum) {

this.start = start;

this.end = end;

this.sum = sum;

this.left = this.right = null;

}

}