**1. strStr()**

public int strStr(String haystack, String needle) {

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

for (int j = 0; ; j++) {

if (j == needle.length()) return i;

if (i + j == haystack.length()) return -1;

if (haystack.charAt(i + j) != needle.charAt(j)) break;

}

}

}

**2. Top K frequent elements**

/\*

// solution 1: use min heap. time complexity: O(NlogK)

public class Solution {

public List<Integer> topKFrequent(int[] nums, int k) {

Map<Integer, Integer> map = new HashMap<>();

for (int num : nums) {

if (!map.containsKey(num)) {

map.put(num, 0);

}

map.put(num, map.get(num) + 1);

}

PriorityQueue<Map.Entry<Integer, Integer>> queue = new PriorityQueue<Map.Entry<Integer, Integer>>(

new Comparator<Map.Entry<Integer, Integer>>() {

public int compare(Map.Entry<Integer, Integer> e1, Map.Entry<Integer, Integer> e2) {

return e1.getValue() - e2.getValue();

}

}

);

for (Map.Entry<Integer, Integer> entry: map.entrySet()) {

if (queue.size() < k) {

queue.offer(entry);

} else if (entry.getValue() > queue.peek().getValue()) {

queue.poll();

queue.offer(entry);

}

}

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

while (!queue.isEmpty()) {

list.add(queue.poll().getKey());

}

return list;

}

}

\*/

// solution 2: use buckets. time complexity: O(N)

public class Solution {

public List<Integer> topKFrequent(int[] nums, int k) {

Map<Integer, Integer> map = new HashMap<>();

for (int num : nums) {

if (!map.containsKey(num)) {

map.put(num, 0);

}

map.put(num, map.get(num) + 1);

}

List[] buckets = new List[nums.length + 1];

for (Integer key : map.keySet()) {

int times = map.get(key);

if (buckets[times] == null) {

buckets[times] = new ArrayList<Integer>();

}

buckets[times].add(key);

}

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

for (int i = buckets.length - 1; i >= 0 && list.size() < k; i--) {

if (buckets[i] != null)

list.addAll(buckets[i]);

}

return list;

}

}

knowledge about priority queue and min/max heap:

What is time complexity for offer, poll and peek methods in priority queue?

**Answer:** Time complexity for the methods offer & poll is O(log(n)) and for the peek() it is Constant time O(1).

Notes:

* In Java, Priority Queue is implemented using Heap Data Structure and Heap has O(log(n)) time complexity to insert and delete element.
* Offer() and add() methods are used to insert the element in the queue.
* Poll() and remove() is used to delete the element from the queue.
* Element retrieval methods i.e. peek() and element(), that are used to retrieve elements from the head of the queue is constant time i.e. O(1).
* contains(Object)method that is used to check if a particular element is present in the queue, have leaner time complexity i.e. O(n).

The default PriorityQueue is implemented with Min-Heap, that is the top element is the minimum one in the heap.

In order to implement a max-heap, you can create your own Comparator:

import java.util.Comparator;

public class MyComparator implements Comparator<Integer>

{

public int compare( Integer x, Integer y )

{

return y - x;

}

}

So, you can create a min-heap and max-heap in the following way:

PriorityQueue minHeap=new PriorityQueue();

PriorityQueue maxHeap=new PriorityQueue(size, new MyComparator());

 A heap can be classified further as either a "**max heap**" or a "**min heap**". In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node. In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node.

**3. parse JSON －－－－－－－－－－－－这题注意——————————————————————————**

**题目：http://www.1point3acres.com/bbs/thread-174445-1-1.html**

h是树的高度，N是结果个数:

time complexity: O(h\*N)

space complexity: O(h)

**public** **class** SolutionForJSON {

List<HashMap<String, String>> convertJSON(JSON apiData, String[] columns) {

List<HashMap<String, String>> list = **new** ArrayList<>();

Map<String, String> map = **new** HashMap<>();

helper(apiData, columns, 0, list, map);

**return** list;

}

**void** helper(JSON apiData, String[] columns, **int** level, List<HashMap<String, String>> list, Map<String, String> map) {

String type = apiData.type();

**if** (type.equals("JSONMapping")) {

String[] keys = apiData.keys();

**for** (String key : keys) {

map.put(columns[level], key);

helper(apiData.get(key), columns, level + 1, list, map);

map.remove(columns[level]);

}

} **else** {

map.put(columns[level], apiData.value());

list.add(**new** HashMap<String, String>(map));

map.remove(columns[level]);

}

}

}

**4. Ternary Expression Parser,**

**——输出树**

public static TreeNode findDup(String s) {  
 if (s == null || s.length() == 0) return null;  
 char[] arr = s.toCharArray();  
 Deque<TreeNode> stack = new ArrayDeque();  
 for (int i = arr.length - 1; i >= 0; i--) {  
 if (arr[i] == ':') continue;  
 if (arr[i] != '?') {  
 stack.push(new TreeNode(arr[i]));  
 } else { // if it's ?, arr[i - 1] == root, arr[i + 1] == left,  
 TreeNode node = new TreeNode(arr[i - 1]);  
 node.left = stack.pop();  
 node.right = stack.pop();  
 stack.push(node);  
 i--;  
 }  
 }  
 return stack.pop();  
}

——lc原题

public String parseTernary(String expression) {  
 Deque<Character> stack = new ArrayDeque();  
 int len = expression.length();  
 for (int i = len - 1; i >= 0; i--) {  
 char c = expression.charAt(i);  
 if (c == ':') continue;  
 if (c != '?') { // digits, T, F  
 stack.push(c);  
 } else { // ?  
 if (expression.charAt(i - 1) == 'T') { // if T, keep stack.peek()  
 char keep = stack.pop();  
 stack.pop(); // keep left, remove right  
 stack.push(keep);  
 } else { // if F  
 stack.pop();  
 }  
 i--; // skip ? mark  
 }  
 }  
 return String.*valueOf*(stack.peek());  
}

**5. Find the Duplicate Number (leetcode)**

如果有时间再看看Linked List Cycle II的方法

// solution 1

// space complexity: O(1), time complexity: O(NlogN)

public class Solution {

public int findDuplicate(int[] nums) {

int lo = 0;

int hi = nums.length - 1;

while (lo < hi) {

int mid = lo + (hi - lo) / 2;

int count = countLess(nums, mid);

if (count > mid) {

hi = mid;

} else {

lo = mid + 1;

}

}

return lo;

}

int countLess(int[] nums, int target) {

int count = 0;

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

if (nums[i] <= target) {

count++;

}

}

return count;

}

}

// solution 2:

public int findDuplicate(int[] nums) {

int fast = nums[nums[0]], slow = nums[0];

while (fast != slow) {

fast = nums[nums[fast]];

slow = nums[slow];

}

fast = 0;

while (fast != slow) {

fast = nums[fast];

slow = nums[slow];

}

return slow;

}

**6. First Missing Positive**

/\*

// solution 1: O(n)

public class Solution {

public int firstMissingPositive(int[] nums) {

int len = nums.length;

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

while (nums[i] <= len && nums[i] > 0 && nums[nums[i] - 1] != nums[i]) {

swap(nums, nums[i] - 1, i);

}

}

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

if (nums[i] != i + 1) return i + 1;

}

return len + 1;

}

private void swap(int[] nums, int i, int j) {

if (nums[i] == nums[j]) return;

nums[i] ^= nums[j];

nums[j] ^= nums[i];

nums[i] ^= nums[j];

}

}

\*/

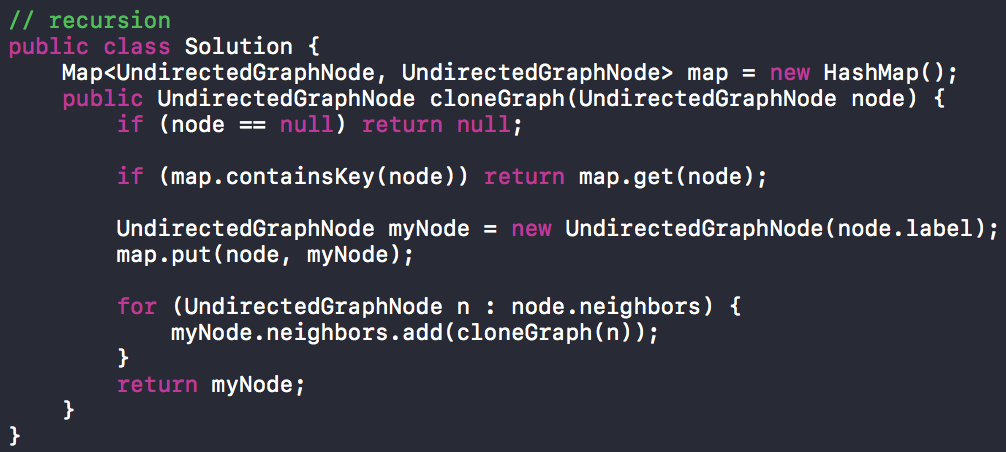
// solution2 : k = partition(nums, 0, n - 1), left side all positive numbers, k is the count of positive numbers.

**7. clone graph -------------------------------------- !!!!!- ----------------------------------------------------------------------------**

// BFS



// dfs



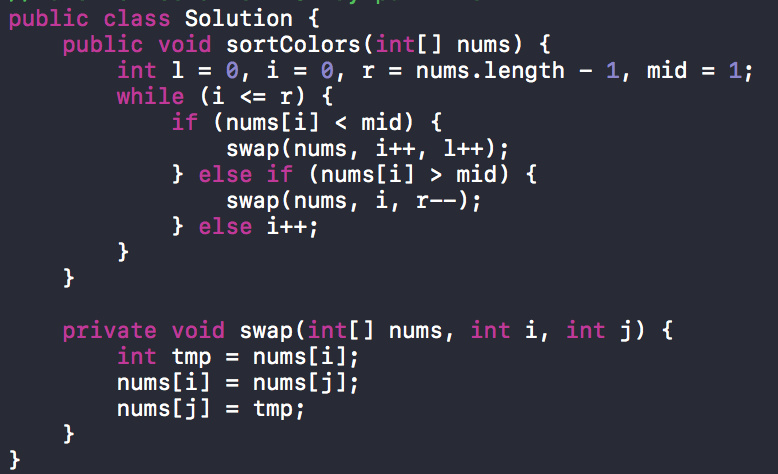
还有如果两个thread同时call 这个方法会怎么样？

1. Use Hashtable instead
2. Collections.synchronizedMap
3. ConcurrentHashMap

lz这里卡了半天，最后经过英国小哥提点才发现是hashmap的问题，hashmap在多线程里就不安全。问我改进方法，说了改成concurrenthashmap

HashMap is not synchronized, therefore it is not Thread Safe and it shouldn’t be used in multi Threaded applications. We can use ConcurrenthHashMap instead.

**8. sort color**



follow up:

4种颜色(0, 1, 2, 3)：

solution 1:

扫一遍：先把0，3分别放最前面和最后面(方法不变)

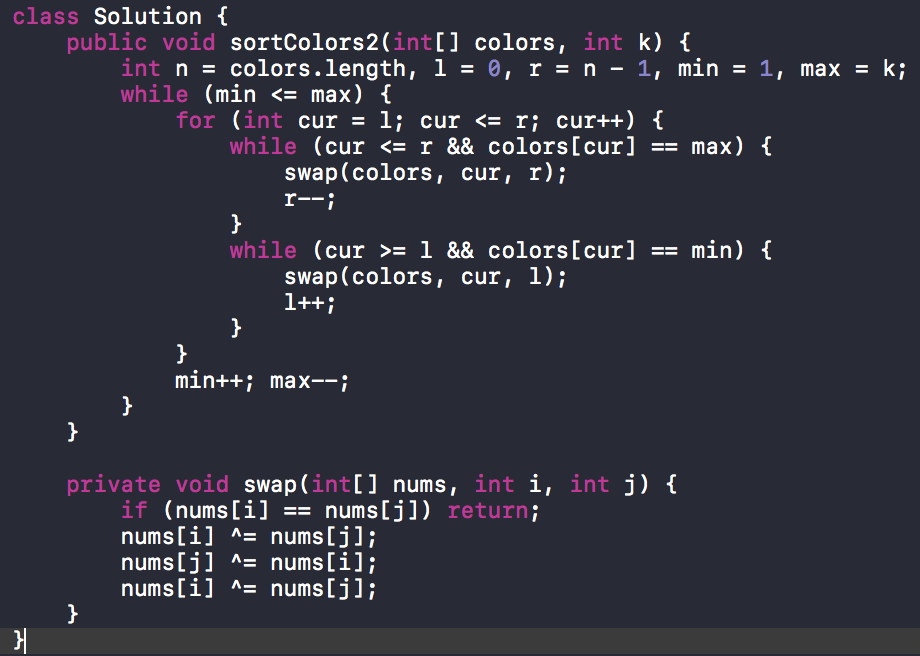
扫第二遍： 再整理1，2

solution2:

扫一遍：计数分别4种颜色分别有多少个, 换到相应的位置，用**负数**计数

扫第二遍：重新构造数组

下面是solution1



**9. Inorder Successor in BST**

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

if (root == null) return null;

// left, root, “right”

if (p.val >= root.val) {

return inorderSuccessor(root.right, p);

} else {

TreeNode minRight = inorderSuccessor(root.left, p);

return minRight == null ? root : minRight;

}

}

**follow up: use parent pointer**

**time complexity O(h)**

**1)** If right subtree of *node*is not *NULL*, then *succ*lies in right subtree. Do following.  
Go to right subtree and return the node with minimum key value in right subtree.  
**2)**If right sbtree of *node*is NULL, then *succ*is one of the ancestors. Do following.  
Travel up using the parent pointer until you see a node which is left child of it’s parent. The parent of such a node is the *succ*.

Node inOrderSuccessor(Node root, Node n) {

if (n.right != null) {

return minValue(n.right);

}

Node p = n.parent;

while (p != null && n == p.right) {

n = p;

p = p.parent;

}

return p;

}

Node minValue(Node node) {

Node current = node;

while (current.left != null) {

current = current.left;

}

return current;

}

**10. RPG Game**

把东西换成价值放到PriorityQueue(max heap) 那我每次只要O(lgn)，然后另外一个Map就维护剩下的物品，比如8个宝石 一个宝石价值5 一个slot最多放5个宝石 我就把 （宝石，25）放到PriorityQueue，然后Map变成（宝石，3个）这意思，（宝石，25）poll出来了再把（宝石，15）放进去

static class Item\_info {  
 int val, stack\_size;  
 public Item\_info(int val, int stack\_size) {  
 this.val = val;  
 this.stack\_size = stack\_size;  
 }  
}  
  
public static int getMaxValue(int n, String[] items, HashMap<String, String>[] item\_infos) {  
 Map<String, Item\_info> itemInfoMap = new HashMap(); // name:info(val, size)  
  
 // item info map  
 for (Map<String, String> info : item\_infos) {  
 int val = Integer.*valueOf*(info.get("value"));  
 int size = Integer.*valueOf*(info.get("maximum\_stack\_size"));  
 Item\_info tmp = new Item\_info(val, size);  
 itemInfoMap.put(info.get("name"), tmp);  
 }  
  
 PriorityQueue<Integer> q = new PriorityQueue<>((a, b) -> (b - a));  
 Map<String, Integer> map = new HashMap(); // item name : count  
  
 // count each item  
 for (String name : items) {  
 map.put(name, map.getOrDefault(name, 0) + 1);  
 }  
  
 // put values into priorityqueue  
 for (String item : map.keySet()) {  
 int maxSize = itemInfoMap.get(item).stack\_size;  
 int val = itemInfoMap.get(item).val;  
 while (map.get(item) > 0) {  
 int count = map.get(item);  
 int curVal = (count > maxSize ? maxSize : count) \* val;  
 map.put(item, count > maxSize ? count - maxSize : 0);  
 q.offer(curVal);  
 }  
 }  
  
 // fetch the top n values  
 int res = 0;  
 while (!q.isEmpty() && n > 0) {  
 res += q.poll();  
 n--;  
 }  
  
 return res;  
}

**11. Find path in 2-d grid**

题目： http://www.1point3acres.com/bbs/thread-199100-1-1.html

class Point {  
 int rowNum;  
 int colNum;  
 int keys;  
 Point parent;  
 Point(int rowNum, int colNum, int keys, Point parent) {  
 this.rowNum = rowNum;  
 this.colNum = colNum;  
 this.keys = keys;  
 this.parent = parent;  
 }  
  
 public String print() {  
 return "Point coordinate: "+ this.colNum + "," + this.rowNum;  
 }  
}  
  
 public List<Point> findShortestPath(String[] inputs) {  
 if (inputs == null || inputs.length == 0 || inputs[0] == null || inputs[0].length() == 0) {  
 return new ArrayList<>();  
 }  
 int row = inputs.length;  
 int col = inputs[0].length();  
  
 Point start = null, end = null;  
  
 // find the start and end point  
 for (int i = 0; i < row; i++) {  
 for (int j = 0; j < col; j++) {  
 if (inputs[i].charAt(j) == '2') {  
 start = new Point(i, j, 0, null);  
 } else if (inputs[i].charAt(j) == '3') {  
 end = new Point(i, j, 0, null);  
 }  
 }  
 }  
 int[][] visited = new int[row][col];  
 Point path = null;  
 Queue<Point> queue = new ArrayDeque(); // bfs  
 queue.offer(start);  
  
 int[][] dirs = {{1, 0}, {-1, 0}, {0, -1}, {0, 1}};  
  
 // if he has a key in his bag, he can traverse the same point at most twice  
 // if he has no key, he can only traverse the same point once  
 while (!queue.isEmpty()) {  
 Point cur = queue.poll();  
 // if it's water, continue  
 if (inputs[cur.rowNum].charAt(cur.colNum) == '0') {  
 continue;  
 }  
 // if reach the end  
 if (cur.rowNum == end.rowNum && cur.colNum == end.colNum) {  
 path = cur;  
 break;  
 }  
 // if the point is visited twice at the same status  
 // same status means twice both without key  
 // or twice both with key  
 if ((visited[cur.rowNum][cur.colNum] & 1) == 1 && ((visited[cur.rowNum][cur.colNum] >> 1) ^ cur.keys) == 0) {  
 continue;  
 }  
 // if it's a key  
 if (inputs[cur.rowNum].charAt(cur.colNum) >= 'a' && inputs[cur.rowNum].charAt(cur.colNum) <= 'z') {  
 int key = inputs[cur.rowNum].charAt(cur.colNum) - 'a';  
 cur.keys ^= (1 << key); // 0~25  
 }  
 // if it's a door  
 if (inputs[cur.rowNum].charAt(cur.colNum) >= 'A' && inputs[cur.rowNum].charAt(cur.colNum) <= 'Z') {  
 int door = inputs[cur.rowNum].charAt(cur.colNum) - 'A';  
 // there is no key for this door  
 if ((cur.keys & (1 << (door))) == 0) {  
 continue;  
 } else {  
 // there is a key  
 cur.keys ^= (1 << (door));  
 }  
 }  
 // mark as visited with key value  
 visited[cur.rowNum][cur.colNum] = (cur.keys << 1) + 1;  
  
 // try 4 directions  
 for (int[] dir : dirs) {  
 int x = cur.rowNum + dir[0];  
 int y = cur.colNum + dir[1];  
 if (x >= 0 && y >= 0 && x < row && y < col) {  
 queue.offer(new Point(x, y, cur.keys, cur));  
 }  
 }  
 }  
   
 // get the final path  
 LinkedList<Point> list = new LinkedList<>();  
 while (path != null) {  
 list.addFirst(path);  
 path = path.parent;  
 }  
 return list;  
 }

**12. calculator** calculator with +-\*/ followup1: ^ followup2: ()

calculator with +-\*/:

without stack:

public class Solution {  
 public int calculate(String s) {  
 s = s.replaceAll(" ", "");  
 int res = 0, pre = 0, opt = '+', i = 0;  
 while (i < s.length()) {  
 int cur = 0;  
 while (i < s.length() && s.charAt(i) >= '0' && s.charAt(i) <= '9') {  
 cur = cur \* 10 + s.charAt(i++) - '0';  
 }  
 if (opt == '+') {  
 res += pre;  
 pre = cur;  
 } else if (opt == '-') {  
 res += pre;  
 pre = -cur;  
 } else if (opt == '\*') {  
 pre \*= cur;  
 } else if (opt == '/') {  
 pre /= cur;  
 }  
 if (i < s.length())  
 opt = s.charAt(i++);  
 }  
 return res + pre;  
 }  
}

with stack:

public int calculate(String s) {  
 s = s.replaceAll(" ", "");  
 int cur = 0, sign = '+';  
 Deque<Integer> stack = new ArrayDeque();  
  
 for (int i = 0; i < s.length(); i++) {  
 if (Character.*isDigit*(s.charAt(i))) {  
 cur = cur \* 10 + s.charAt(i) - '0';  
 }  
 if (!Character.*isDigit*(s.charAt(i)) || i == s.length() - 1) {  
 if (sign == '+') {  
 stack.push(cur);  
 } else if (sign == '-') {  
 stack.push(-cur);  
 } else if (sign == '\*') {  
 stack.push(stack.pop() \* cur);  
 } else if (sign == '/') {  
 stack.push(stack.pop() / cur);  
 }  
 sign = s.charAt(i);  
 cur = 0;  
 }  
 }  
 int res = 0;  
 while (!stack.isEmpty()) {  
 res += stack.pop();  
 }  
 return res;  
}

followup1: ^

public int getValue(String expression){  
 expression = expression.replaceAll(" ", "");  
 Stack<Integer> s1 = new Stack<>();  
 Stack<Character> s2 = new Stack<>();  
 Map<Character, Integer> opeLvl = new HashMap<>();  
 opeLvl.put('+', 1);  
 opeLvl.put('-', 1);  
 opeLvl.put('\*', 2);  
 opeLvl.put('/', 2);  
 opeLvl.put('^', 3);  
  
 int cur = 0;  
 for (int i = 0; i < expression.length(); i++){  
 char c = expression.charAt(i);  
 if (Character.*isDigit*(c)){  
 cur = cur \* 10 + (c - '0');  
 } else {  
 s1.push(cur);  
 cur = 0;  
 while (!s2.isEmpty() && opeLvl.get(c) <= opeLvl.get(s2.peek())){  
 int temp = operation(s1.pop(), s1.pop(), s2.pop());  
 s1.push(temp);  
 }  
 s2.push(c);  
 }  
 }  
 //这个十分的重要,缺了这个就stack里面不够了。  
 s1.push(cur);  
 int res = 0;  
 while (!s2.isEmpty()){  
 s1.push(operation(s1.pop(), s1.pop(), s2.pop()));  
 }  
 res = s1.peek();  
 return res;  
}  
 //a is the first pop up。其实带了乘方,也无妨  
 private int operation(int b, int a, char c){  
 int res = 0;  
 if (c == '+'){  
 res = a + b;  
 }  
 else if (c == '-'){  
 return a - b;  
 }  
 else if (c == '\*'){  
 res = a \* b;  
 } else if (c == '/'){  
 res = a / b;  
 } else if (c == '^'){  
 res = (int)Math.*pow*(a, b);  
 }  
 return res;  
 }

followup2: ()

public int getValue(String expression){  
 int res = 0;  
 expression = expression.replaceAll(" ", "");  
 Stack<Integer> s1 = new Stack<>();  
 Stack<Character> s2 = new Stack<>();  
 Map<Character, Integer> opeLvl = new HashMap<>();  
 opeLvl.put('+', 1);  
 opeLvl.put('-', 1);  
 opeLvl.put('\*', 2);  
 opeLvl.put('/', 2);  
 opeLvl.put('^', 3);  
  
 int cur = 0;  
 for (int i = 0; i < expression.length(); i++){  
 char c = expression.charAt(i);  
 if (Character.*isDigit*(c)){  
 cur = cur \* 10 + (c - '0');  
 } else {  
 if (c == '(') {  
 s2.push(c);  
 continue;  
 }  
 s1.push(cur);  
 cur = 0;  
 if (c == ')'){  
 while (s2.peek() != '('){  
  
 int temp = operation(s1.pop(), s1.pop(), s2.pop());  
 s1.push(temp);  
 }  
 s2.pop();  
 //这里还单纯的开了个倒车,遇到括号这真的是不好写啊  
 //因为)+这样,会当做两次计算,所以括号相对特殊,占用curNum使得)没有计算的属性。  
 cur = s1.pop();  
 continue;  
 }  
 while (!s2.isEmpty() && s2.peek() != '(' && opeLvl.get(c) <= opeLvl.get(s2.peek())){  
 int temp = operation(s1.pop(), s1.pop(), s2.pop());  
 s1.push(temp);  
 }  
 s2.push(c);  
 }  
 }  
 //这个十分的重要,缺了这个就stack里面不够了。  
 s1.push(cur);  
 while (!s2.isEmpty()){  
 s1.push(operation(s1.pop(), s1.pop(), s2.pop()));  
 }  
 res = s1.peek();  
 return res;  
 }  
 //a is the first pop up。其实带了乘方,也无妨  
 private int operation(int b, int a, char c){  
 int res = 0;  
 if (c == '+'){  
 res = a + b;  
 }  
 else if (c == '-'){  
 return a - b;  
 }  
 else if (c == '\*'){  
 res = a \* b;  
 } else if (c == '/'){  
 res = a / b;  
 } else if (c == '^'){  
 res = (int)Math.*pow*(a, b);  
 }  
 return res;  
 }

**13. Find leaves of binary tree**

public class Solution {  
 public List<List<Integer>> findLeaves(TreeNode root) {  
 List<List<Integer>> res = new ArrayList();  
 helper(root, res);  
 return res;  
 }  
  
 private int helper(TreeNode root, List<List<Integer>> res) {  
 if (root == null) return -1;  
 int level = Math.*max*(helper(root.left, res), helper(root.right, res)) + 1;  
 if (res.size() <= level) res.add(new ArrayList());  
 res.get(level).add(root.val);  
 return level;  
 }  
}

**14. 等概率random**

解法： http://www.geeksforgeeks.org/reservoir-sampling/

void selectKItems(int stream[], int n, int k)  
{  
 Random rand = new Random();  
 int i; // index for elements in stream[]  
  
 // reservoir[] is the output array. Initialize it with  
 // first k elements from stream[]  
 int[] reservoir = new int[k];  
 for (i = 0; i < k; i++)  
 reservoir[i] = stream[i];  
  
 // Iterate from the (k+1)th element to nth element  
 for (; i < n; i++)  
 {  
 // Pick a random index from 0 to i.  
 int j = rand.nextInt(i+1);  
  
 // If the randomly picked index is smaller than k, then replace  
 // the element present at the index with new element from stream  
 if (j < k)  
 reservoir[j] = stream[i];  
 }  
  
 // print reservoir[]   
}

**15. contain duplicates**

Given an array of integers, find out whether there are two distinct indices i and j in the array such that the difference between nums[i] and nums[j] is at most t and the difference between i and j is at most k.

Treeset:

This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

floor(E e) :Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.

public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {  
 TreeSet<Integer> set = new TreeSet<>();  
 for (int i = 0; i < nums.length; i++) {  
 // a – b <= t   
 // a <= t + b  
 // b <= a  
 Integer floor = set.floor(nums[i] + t);  
 // b – a <= t  
 // b <= a + t  
 // b >= a  
 Integer ceiling = set.ceiling(nums[i] - t);  
 if ((floor != null && nums[i] <= floor) || (ceiling != null && nums[i] >= ceiling)) {  
 return true;  
 }  
 set.add(nums[i]);  
 if (i >= k) {  
 set.remove(nums[i - k]);  
 }  
 }  
 return false;  
}

**16. topological sort ================**

We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

void topologicalSort()

    {

        Stack stack = new Stack();

        // Mark all the vertices as not visited

        boolean visited[] = new boolean[V];

        // Call the recursive helper function to store

        // Topological Sort starting from all vertices

        // one by one

        for (int i = 0; i < V; i++)

            if (visited[i] == false)

                topologicalSortUtil(i, visited, stack);

        // Print contents of stack

        while (stack.empty()==false)

            System.out.print(stack.pop() + " ");

    }

void topologicalSortUtil(int v, boolean visited[],

                             Stack stack)

    {

        // Mark the current node as visited.

        visited[v] = true;

        Integer i;

        // Recur for all the vertices adjacent to this

        // vertex

        Iterator<Integer> it = adj[v].iterator();

        while (it.hasNext())

        {

            i = it.next();

            if (!visited[i])

                topologicalSortUtil(i, visited, stack);

        }

        // Push current vertex to stack which stores result

        stack.push(new Integer(v));

    }

    // The function to do Topological Sort. It uses

    // recursive topologicalSortUtil()

**17. same number of elements -----------------===================**

Input:

Array of numbers: A

number: x

Output: find 0 < k < A.size() (divide A into two non-empty parts), such that the number of elements in A[0,..., k - 1] == x is same as number of elements in A[k, ..., N - 1] != x.

Return -1 if not found

public class SameNumber {

public static int find(int[] arr, int x) {

if (arr.length < 2) {

return -1;

}

int count = 0;

for (int item : arr) {

if (item == x) {

count++;

}

}

int left = 0;

int right = arr.length - count;

int k = 1;

while (k < arr.length) {

if (arr[k - 1] == x) {

left++;

} else {

right--;

}

if (left == right) {

return k;

}

k++;

}

return -1;

}

public static void main(String[] args) {

SameNumber ob = new SameNumber();

int[] arr = new int[]{6, 9, 7, 6, 6, 3, 2, 6, 4};

int x = 6;

int k = ob.find(arr, x);

System.out.println(k);

}

}

**18. Longest Repeating Character Replacement**

Since we are only interested in the longest valid substring, our sliding windows need not shrink, even if a window may cover an invalid substring. We either grow the window by appending one char on the right, or shift the whole window to the right by one. And we only grow the window when the count of the new char exceeds the historical max count (from a previous window that covers a valid substring).

That is, we do not need the accurate max count of the current window; we only care if the max count exceeds the historical max count; and that can only happen because of the new char.

public class Solution {  
 public int characterReplacement(String s, int k) {  
 int[] map = new int[128];  
 int start = 0, end = 0, maxCount = 0, res = 0;  
 while (end < s.length()) {  
 maxCount = Math.*max*(maxCount, ++map[s.charAt(end++)]);  
 while (end - start - maxCount > k) {  
 map[s.charAt(start++)]--;  
 }  
 res = Math.*max*(end - start, res);  
 }  
 return res;  
 }  
}

19

一个 int array,   每个元素 在 1 到 25000 之间， O(n) 复杂度 打印出 有 重复的 数字， 用了 bitmap， 空间复杂度 在 4 k byte 之内。  用 一个 长度 1000 的 int array， 每个数字 ／ 31  得到  是在第几个 int 里， ％ 31 表示 在这个 int 的第几位 index， 用 1 << index 来存。 随后 面试官 加问了 << 和 <<<的区别。

public static List<Integer> findDup(int[] nums) {  
 if (nums == null || nums.length == 0) return new ArrayList<>();  
 List<Integer> res = new ArrayList();  
 int n = nums.length;  
 int[] table = new int[1000];  
 for (int num : nums) {  
 int idx = num/31;  
 int pos = num%31;  
 if ((table[idx] >>> pos & 1) == 0) { // first time  
 table[idx] |= 1 << pos;  
 } else { // duplicate  
 res.add(num);  
 }  
 }  
 return res;  
}

20.

题目大概的意思就是给你很多个view，每个view有几个button，然后通过按button可以切换view，让设计一个方法，可以按遍所有的button。

21.

**A\* 算法**

public class Main {  
 public static void A\_star(int[][] graph, int start, int end) {  
 int V = graph.length;  
  
 // the set of nodes already evaluated  
 Set<Integer> closedSet = new HashSet<>();  
  
 // the set of currently discovered nodes that are already evaluated  
 // initially only the start node is known.  
 Set<Integer> openSet = new HashSet<>();  
 openSet.add(start);  
  
 // the most efficient previous step, i.e. the parent node  
 int[] cameFrom = new int[V];  
 for (int i = 0; i < V; i++) cameFrom[i] = i;  
  
 // for each node, the cost of getting from start node to that node  
 int[] gScore = new int[V];  
 Arrays.*fill*(gScore, Integer.*MAX\_VALUE*);  
 gScore[start] = 0;  
  
 // fScore = gScore + hScore  
 // hScore is the estimated cost to move from that node to the end  
 // The value is partly known, partly heuristic  
 int[] fScore = new int[V];  
 Arrays.*fill*(fScore, Integer.*MAX\_VALUE*);  
 // 3 ways to calculate "h"  
 // 1. Euclidean Distance (Exact Heuristics)  
 // when there are no blocked cells/obstacles  
 // h = sqrt ( (current\_cell.x – goal.x)2 + (current\_cell.y – goal.y)2 )  
  
 // 2.1 Manhattan Distance (Approximation Heuristics)  
 // When we are allowed to move only in four directions only  
 // h = abs (current\_cell.x – goal.x) + abs (current\_cell.y – goal.y)  
  
 // 2.2 Diagonal Distance (Approximation Heuristics)  
 // When we are allowed to move 8 directions only  
 // h = max { abs(current\_cell.x – goal.x), abs(current\_cell.y – goal.y) }  
  
 fScore[start] = *hScore*(start, end);  
  
 while (!openSet.isEmpty()) {  
 int cur = *minF*(openSet, fScore, V);  
  
 // reach the end point  
 if (cur == end) {  
 *constructPath*(cameFrom, cur, start);  
 }  
  
 // current node has been evaluated  
 openSet.remove(cur);  
 closedSet.add(cur);  
  
 for (int v = 0; v < V; v++) { // for each neighbor of cur  
 // not a neighbor, undirected graph  
 if (graph[cur][v] == 0) continue;  
 // has been evaluated  
 if (closedSet.contains(v)) continue;  
  
 // the dist from start to the neighbor  
 int tmp\_gScore = gScore[cur] + graph[cur][v];  
  
 if (!openSet.contains(v)) openSet.add(v); // Discover a new node  
 else if (tmp\_gScore >= gScore[v]) continue; // this is not a better path  
  
 // else, this is a better path  
 cameFrom[v] = cur;  
 gScore[v] = tmp\_gScore;  
 fScore[v] = gScore[v] + *hScore*(v, end);  
 }  
 }  
 }  
  
 private static int hScore(int start, int end) {  
 // here we use Manhattan dist  
 return 2 \* Math.*abs*(end - start);  
 }  
  
 private static int minF(Set<Integer> openSet, int[] fScore, int V) {  
 int minVal = Integer.*MAX\_VALUE*, idx = -1;  
 for (int i = 0; i < V; i++) {  
 if (openSet.contains(i) && fScore[i] < minVal) {  
 minVal = fScore[i];  
 idx = i;  
 }  
 }  
 return idx;  
 }  
  
 private static void constructPath(int[] cameFrom, int cur, int start) {  
 // construct path  
 LinkedList<Integer> finalPath = new LinkedList<>();  
 finalPath.addFirst(cur + 1);  
 while (true) {  
 cur = cameFrom[cur];  
 finalPath.addFirst(cur + 1);  
 if (cur == start) break;  
 }  
 System.*out*.print(finalPath);  
 System.*out*.println();  
 }  
  
 public static void main(String[] args) {  
 int graph[][] = {  
 {0,0,0,1,0},  
 {0,0,0,1,1},  
 {0,0,0,0,1},  
 {1,1,0,0,1},  
 {0,1,1,1,0},  
 };  
 *A\_star*(graph, 0, 2);  
 }  
}

**22. Dijkstra’s shortest path algorithm**

public static void Dijkstra(int[][] graph, int start)  
{  
 int V = graph.length;  
 int[] dist = new int[V]; // distance from start to that node  
 boolean[] sptSet = new boolean[V]; // shortest path tree set  
  
 // initialize dist to infinite  
 for (int i = 0; i < V; i++) {  
 dist[i] = Integer.*MAX\_VALUE*;  
 }  
  
 dist[start] = 0;  
  
 // find shortest path for all vertices  
 for (int count = 0; count < V-1; count++) {  
 // pick the minimum distance vertex from vertices that are not in sptSet  
 int u = *minDist*(dist, sptSet, V);  
 // set the vertex as processed  
 sptSet[u] = true;  
 // update the adjacent vertices of the picked vertex  
 for (int v = 0; v < V; v++) {  
 if (!sptSet[v] && graph[u][v] != 0 && dist[u] != Integer.*MAX\_VALUE* && dist[u] + graph[u][v] < dist[v]) {  
 dist[v] = dist[u] + graph[u][v];  
 }  
 }  
 }  
  
 // print path  
 for (int i = 0; i < V; i++) {  
 System.*out*.println(i + "\t" + dist[i]);  
 }  
}  
  
public static int minDist(int[] dist, boolean[] sptSet, int V) {  
 int minDist = Integer.*MAX\_VALUE*, idx = -1;  
 for (int v = 0; v < V; v++) {  
 if (!sptSet[v] && dist[v] < minDist) {  
 minDist = dist[v];  
 idx = v;  
 }  
 }  
 return idx;  
}

**23. Floyd’s**

func floyd2D(\_ graph: [[Int]])  
 {  
 let n = graph.count  
 var dist = Array(repeating: Array(repeatElement(0, count: n)), count: n)  
 var path = Array(repeating: Array(repeatElement(0, count: n)), count: n)  
 for i in 0..<n {  
 for j in 0..<n {  
 if i == j {  
 dist[i][j] = 0  
 } else if graph[i][j] == 0 {  
 dist[i][j] = Int(Int32.max)  
 } else {  
 dist[i][j] = graph[i][j]  
 }  
 }  
 }  
 for k in 0..<n {  
 for i in 0..<n {  
 for j in 0..<n {  
 if dist[i][k] + dist[k][j] < dist[i][j] {  
 path[i][j] = k  
 dist[i][j] = dist[i][k] + dist[k][j]  
 }  
 }  
 }  
 }  
// return path  
 }

**24. Shortest Palindrome**

public class Solution {  
 public String shortestPalindrome(String s) {  
 String ss = s + "+" + new StringBuilder(s).reverse().toString();  
 int[] table = new int[ss.length()];  
 int j = 0;  
 for (int i = 1; i < ss.length(); i++) {  
 if (ss.charAt(j) == ss.charAt(i)) {  
 table[i] = j + 1;  
 j++;  
 } else {  
 while (j > 0 && ss.charAt(j) != ss.charAt(i)) {  
 j = table[j - 1];  
 }  
 if (ss.charAt(j) == ss.charAt(i)) {  
 j++;  
 }  
 table[i] = j;  
 }  
 }  
 return new StringBuilder(s.substring(table[table.length - 1])).reverse().toString() + s;  
 }  
}

O(n) solution:

public String shortestPalindrome(String s) {

int j = 0;

for (int i = s.length() - 1; i >= 0; i--) {

if (s.charAt(i) == s.charAt(j))

j++;

}

if (j == s.length()) return s;

String suffix = s.substring(j);

return new StringBuilder(suffix).reverse().toString() + shortestPalindrome(s.substring(0, j)) + suffix;

}

// 442. Find All Duplicates in an Array

public class Solution {  
 public List<Integer> findDuplicates(int[] nums) {  
 List<Integer> res = new ArrayList();  
 for (int i = 0; i < nums.length; i++) {  
 int idx = Math.*abs*(nums[i]);  
 if (nums[idx - 1] < 0) {  
 res.add(idx);  
 }  
 nums[idx - 1] = -nums[idx - 1];  
 }  
 return res;  
 }  
}