**1. strStr()**

time complexity: O(MN)

an example of worst time complexity: aaaaaab, ab

public class Solution {

public int strStr(String haystack, String needle) {

return haystack.indexOf(needle);

}

}

public class Solution {

public int strStr(String haystack, String needle) {

if (needle.length() == 0) {

return 0;

}

for (int i = 0; i < haystack.length() - needle.length() + 1; i++) {

int k = i;

int j = 0;

while (j < needle.length()) {

if (haystack.charAt(k) == needle.charAt(j)) {

k++;

j++;

} else {

break;

}

}

if (j == needle.length()) {

return i;

}

}

return -1;

}

}

**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**

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);

}

} **else** {

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

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

}

}

}

**4. Ternary Expression Parser**

public class Solution {

public String parseTernary(String expression) {

Stack<Character> stack = new Stack<>();

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

char c = expression.charAt(i);

if (!stack.isEmpty() && stack.peek() == '?') {

stack.pop(); // pop '?'

char left = stack.pop();

stack.pop(); // pop ':'

char right = stack.pop();

if (c == 'T') {

stack.push(left);

} else {

stack.push(right);

}

} else {

stack.push(c);

}

}

return String.valueOf(stack.pop());

}

}

**5. Find the Duplicate Number**

如果有时间再看看Linked List Cycle II的方法

/\*

// solution 1: intuitively use set.

// space complexity: O(N), time complexity: O(N)

public class Solution {

public int findDuplicate(int[] nums) {

Set<Integer> set = new HashSet<>();

for (int num : nums) {

if (set.contains(num)) {

return num;

}

set.add(num);

}

return 0;

}

}

\*/

// solution 2

// 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;

}

}

**6. First Missing Positive**

/\*

// solution 1: naive solution. time complexity: O(NlogN)

public class Solution {

public int firstMissingPositive(int[] nums) {

Arrays.sort(nums);

int i = 0;

while (i < nums.length && nums[i] <= 0) {

i++;

}

if (i == nums.length || nums[i] != 1) {

return 1;

}

while (i < nums.length - 1) {

if (nums[i + 1] == nums[i] || nums[i + 1] == nums[i] + 1) {

i++;

} else {

return nums[i] + 1;

}

}

return nums[nums.length - 1] + 1;

}

}

\*/

// solution 2: time complexity: O(N)

public class Solution {

public int firstMissingPositive(int[] nums) {

int len = nums.length;

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

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

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

}

}

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

if (nums[i] != i + 1) {

return i + 1;

}

}

return len + 1;

}

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

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

**7. clone graph**

// BFS

public class Solution {

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null) {

return null;

}

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

Queue<UndirectedGraphNode> queue = new LinkedList<>();

UndirectedGraphNode cloned = new UndirectedGraphNode(node.label);

queue.offer(node);

map.put(cloned.label, cloned);

while (!queue.isEmpty()) {

UndirectedGraphNode cur = queue.poll();

for (UndirectedGraphNode neighbor : cur.neighbors) {

if (!map.containsKey(neighbor.label)) {

map.put(neighbor.label, new UndirectedGraphNode(neighbor.label));

queue.offer(neighbor);

}

map.get(cur.label).neighbors.add(map.get(neighbor.label));

}

}

return cloned;

}

}

follow up: 不用node.label怎么办

在hashmap里面改成<node,node>就好了。

public class Solution {

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null) {

return null;

}

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

Queue<UndirectedGraphNode> queue = new LinkedList<>();

UndirectedGraphNode cloned = new UndirectedGraphNode(node.label);

map.put(node, cloned); // notice!

queue.offer(node);

while (!queue.isEmpty()) {

UndirectedGraphNode cur = queue.poll();

for (UndirectedGraphNode neighbor : cur.neighbors) {

if (!map.containsKey(neighbor)) {

map.put(neighbor, new UndirectedGraphNode(neighbor.label));

queue.offer(neighbor);

}

map.get(cur).neighbors.add(map.get(neighbor));

}

}

return cloned;

}

}

还有如果两个thread同时call 这个方法会怎么样？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.

// DFS

public class Solution {

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

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null) {

return null;

}

if (map.containsKey(node.label)) {

return map.get(node.label);

}

UndirectedGraphNode cloned = new UndirectedGraphNode(node.label);

map.put(cloned.label, cloned);

for (UndirectedGraphNode neighbor : node.neighbors) {

cloned.neighbors.add(cloneGraph(neighbor));

}

return cloned;

}

}

**8. sort color**

public class Solution {

public void sortColors(int[] nums) {

int lo = 0;

int hi = nums.length - 1;

int i = 0;

while (i <= hi) {

if (nums[i] == 0) {

swap(nums, lo++, i++);

// 如果上面不是写的i++，而是写的i

// Last executed input: [0]

// Runtime Error Message: Line 21: java.lang.ArrayIndexOutOfBoundsException: 1

// i 不变，lo变为nums.length了

} else if (nums[i] == 1) {

i++;

} else {

swap(nums, hi--, i);

// 如果上面写的是i++

// Input: [1,2,0]; Output:[1,0,2]; Expected:[0,1,2]

// 从高位换来的0就没有办法处理

}

}

}

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

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

}

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

扫第二遍：重新构造数组

**9. Inorder Successor in BST**

public class Solution {

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

while (root != null && root.val <= p.val) {

root = root.right;

}

if (root == null) {

return null;

}

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

if (left != null) {

return left;

} else {

return root;

}

}

}

**follow up: use parent pointer**

**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) {

// step 1 of the above algorithm

if (n.right != null) {

return minValue(n.right);

}

// step 2 of the above algorithm

Node p = n.parent;

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

n = p;

p = p.parent;

}

return p;

}

Node minValue(Node node) {

Node current = node;

/\* loop down to find the leftmost leaf \*/

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

public class MaxValue {

static int getMaxValue(int n, String[] items, item\_infos) {

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

// max heap

PriorityQueue<Integer> queue = new PriorityQueue<Integer>(

new Comparater<Integer>() {

public int compare(Integer i1, Integer i2) {

return i2 - i1;

}

}

);

for (String item : items) {

if (!map.containsKey(item)) {

map.put(item, 0);

}

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

}

for (String item : map.keySet()) {

while (map.get(item) > 0) {

if (map.get(item) >= item\_infos.get(item).max\_stack\_size) {

queue.offer(max\_stack\_size \* item\_infos.get(item).value);

map.put(item, map.get(item) - item\_infos.get(item).max\_stack\_size);

} else {

queue.offer(map.get(item) \* item\_infos.get(item).value);

map.put(item, 0);

}

}

}

int res = 0;

while (!queue.isEmpty() && n > 0) {

res += queue.poll();

n--;

}

return res;

}

}

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

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;

}

@Override

public String toString() {

return "Point coordinate: "+ this.colNum + "," + this.rowNum;

}

}

public class FindPath {

static List<Point> findShortestPath(String[] inputs) {

int row = inputs.length;

if (row == 0) {

return new ArrayList<>();

}

int col = inputs[0].length();

Point start = null, end = null;

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 LinkedList<>();

queue.offer(start);

while (!queue.isEmpty()) {

Point cur = queue.poll();

// System.out.println(cur.toString());

if (inputs[cur.rowNum].charAt(cur.colNum) == '0') {

continue;

}

if (cur.rowNum == end.rowNum && cur.colNum == end.colNum) {

path = cur;

break;

}

if ((visited[cur.rowNum][cur.colNum] & 1) == 1 && ((visited[cur.rowNum][cur.colNum] >> 1) ^ cur.keys) == 0) {

continue;

}

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);

}

if (inputs[cur.rowNum].charAt(cur.colNum) >= 'A' && inputs[cur.rowNum].charAt(cur.colNum) <= 'Z') {

int key = inputs[cur.rowNum].charAt(cur.colNum) - 'A';

if ((cur.keys & (1 << (key))) == 0) {

continue;

} else {

cur.keys -= (1 << (key));

}

}

visited[cur.rowNum][cur.colNum] = (cur.keys << 1) + 1;

if (cur.rowNum - 1 >= 0) {

queue.offer(new Point(cur.rowNum - 1, cur.colNum, cur.keys, cur));

}

if (cur.rowNum + 1 < row) {

queue.offer(new Point(cur.rowNum + 1, cur.colNum, cur.keys, cur));

}

if (cur.colNum - 1 >= 0) {

queue.offer(new Point(cur.rowNum, cur.colNum - 1, cur.keys, cur));

}

if (cur.colNum + 1 < col) {

queue.offer(new Point(cur.rowNum, cur.colNum + 1, cur.keys, cur));

}

}

LinkedList<Point> list = new LinkedList<>();

while (path != null) {

list.addFirst(path);

path = path.parent;

}

return list;

}

public static void main(String[] args) {

FindPath ob = new FindPath();

String[] inputs1 = new String[]{"02111", "01001", "01003", "01001", "01111"};

List<Point> list1 = ob.findShortestPath(inputs1);

System.out.println("-------------------------------------------");

System.out.println("outputs1:");

for (Point p : list1) {

System.out.println(p.toString());

}

System.out.println("-------------------------------------------");

String[] inputs2 = new String[]{"02a11", "0100A", "01003", "01001", "01111"};

List<Point> list2 = ob.findShortestPath(inputs2);

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

calculator with +-\*/:

public class Solution {

public int calculate(String s) {

s = s.replaceAll(" ", "");

if (s.length() == 0) {

return 0;

}

int num = 0;

char sign = '+';

Stack<Integer> stack = new Stack<>();

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

char c = s.charAt(i);

if (Character.isDigit(c)) {

num = num \* 10 + c - '0';

}

if (!Character.isDigit(c) || i == s.length() - 1) {

if (sign == '+') {

stack.push(num);

} else if (sign == '-') {

stack.push(-num);

} else if (sign == '\*') {

stack.push(stack.pop() \* num);

} else {

stack.push(stack.pop() / num);

}

sign = c;

num = 0;

}

}

int res = 0;

while (!stack.isEmpty()) {

res += stack.pop();

}

return res;

}

}

followup1: ^ followup2: ()

public class Find\_Expression\_Value {

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<>();

char sign = '+';

opeLvl.put('+', 1);

opeLvl.put('-', 1);

opeLvl.put('\*', 2);

opeLvl.put('/', 2);

opeLvl.put('^', 3);

int curNum = 0;

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

char c = expression.charAt(i);

if (Character.isDigit(c)){

curNum = curNum \* 10 + (c - '0');

} else {

if (c == '(') {

s2.push(c);

continue;

}

// 有负数的运算才需要加这一部分

//遇到符号要清零,这个注意一下,就是这里加和减都表示正和负。

if (c == '+' || c == '-'){

if (i == 0){

sign = c;

continue;

}

char pre = expression.charAt(i-1);

// 处理这种情况：2+3\*-4+5

if (pre != '(' && pre != ')' && !Character.isDigit(pre)){

sign = c;

continue;

}

}

//遇到其他符号,先把数字压进去。

if (sign == '-') {

curNum \*= -1;

sign = '+'; //恢复

}

s1.push(curNum);

curNum = 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。

curNum = s1.pop();

continue;

}

while (!s2.isEmpty() && s2.peek() != '(' && opeLvl.get(c) <= opeLvl.get(s2.peek())){

int a = s1.pop();

int b = s1.pop();

int temp = operation(a, b, 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(curNum);

//System.out.println(expression); 这里就是看一下replaceAll这个办法的返回值

while (!s2.isEmpty()){

int a = s1.pop();

int b = s1.pop();

s1.push(operation(a, b, 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 a, int b, char c){

int res = 0;

if (c == '+'){

res = a + b;

}

else if (c == '-'){

return b - a;

}

else if (c == '\*'){

res = a \* b;

}

else if (c == '^'){

res = (int)Math.pow(b, a);

}

return res;

}

**13. Find leaves of binary tree**

public class Solution {

public List<List<Integer>> findLeaves(TreeNode root) {

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

findLeavesHelper(list, root);

return list;

}

private int findLeavesHelper(List<List<Integer>> list, TreeNode root) {

if (root == null) {

return -1;

}

int leftLevel = findLeavesHelper(list, root.left);

int rightLevel = findLeavesHelper(list, root.right);

int level = Math.max(leftLevel, rightLevel) + 1;

if (list.size() == level) {

list.add(new ArrayList<>());

}

list.get(level).add(root.val);

return level;

}

}

**14. 等概率random**

最后是random从一个array里面选出来一个数放到另一个array里面去～ 然后问random函数如何写才能保证等概率的取到各个idx

int[] arr2 = Arrays.copyOf(arr1, arr1.length);

Random randomno = new Random();

for (int i = 0; i < arr2.length; i++) {

swap(arr2, i, i + randomno.nextInt(arr2.length - i));

}

// nextInt is uniformly distributed

**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 class Solution {

public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {

TreeSet<Integer> set = new TreeSet<>();

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

Integer floor = set.floor(nums[i] + t);

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];

        for (int i = 0; i < V; i++)

            visited[i] = false;

        // 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**

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[] count = new int[26];

int max = 0;

int start = 0;

for (int end = 0; end < s.length(); end++) {

max = Math.max(max, ++count[s.charAt(end) - 'A']);

if (max + k <= end - start) {

count[s.charAt(start) - 'A']--;

start++;

}

}

return s.length() - start;

}

}