

一、刷题时间表

- [2021.10.27-difficult:301.移除无效的括号](#)
- [2021.10.28-mid:869. 重新排序得到 2 的幂](#)
- [2021.10.29-difficult:335.路径交叉](#)
- [2021.10.30-mid:260.只出现一次的数字3](#)
- [2021.10.31-easy: 500.键盘行](#)
- [2021.11.1-easy:575.分糖果](#)
- [2021.11.2 - easy: 237.删除链表中的节点](#)
- [2021.11.3-difficult: 407.接雨水2](#)
- [2021.11.4-easy:367有效的完全平方](#)
- [2021.11.5-mid: 1218.最长定差子序列](#)
- [2021.11.6-easy: 268.只出现一次的数字](#)
- [2021.11.7-easy: 598.范围求和](#)
- [2021.11.8-mid: 299.猜数字游戏](#)
- [2021.11.9-difficult: 488 ZumaGame](#)
- [2021.11.10-easy: 495.TeemoAttacking](#)
- [2021.11.11-difficult: 629.KInversePairs](#)
- [2021.11.12-mid: 375.Guess number higher or lower2](#)
- [2021.11.13-easy: 520.DetectCapital](#)
- [2021.11.14-mid: 677. Map Sum Pairs](#)
- [2021.11.15-mid: 319.bulb-switcher](#)
- [2021.11.16-difficult:391.PerfectRectangle](#)
- [2021.11.17-mid: 318.MaxmumProductOfWordLengths](#)

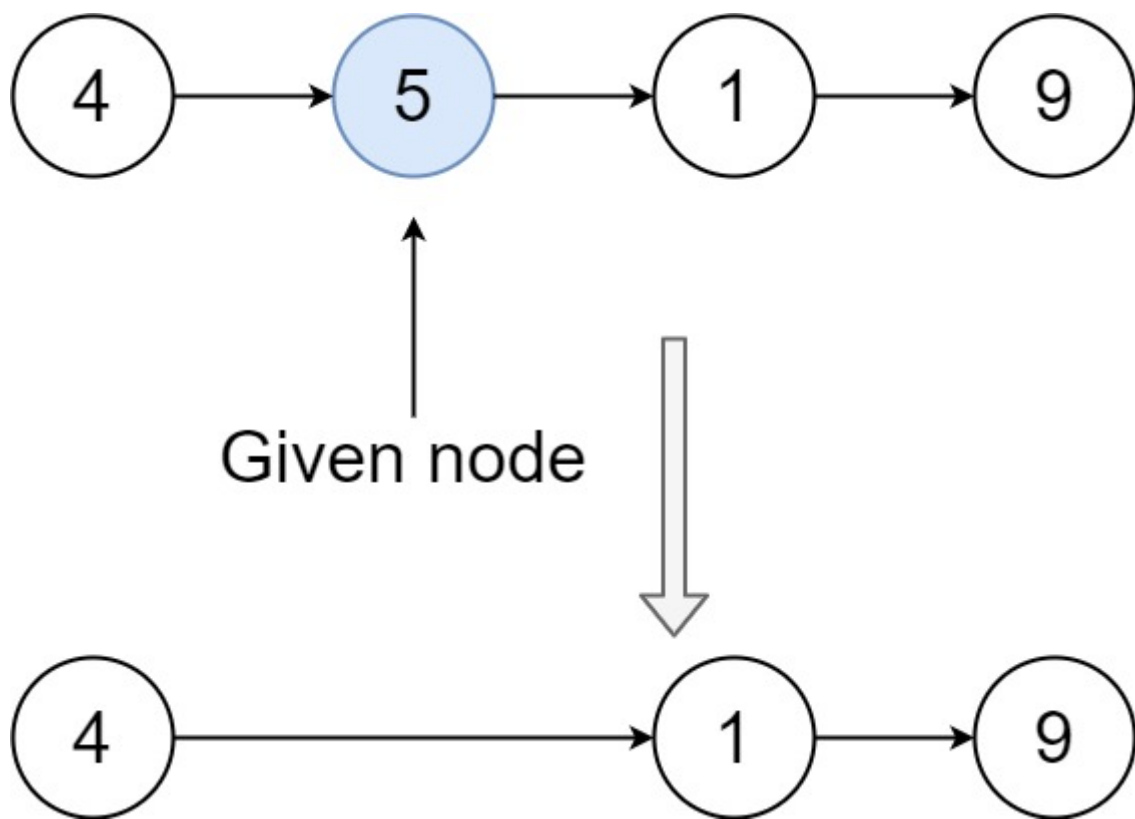
二、逐题题解

237.删除链表中的节点 easy

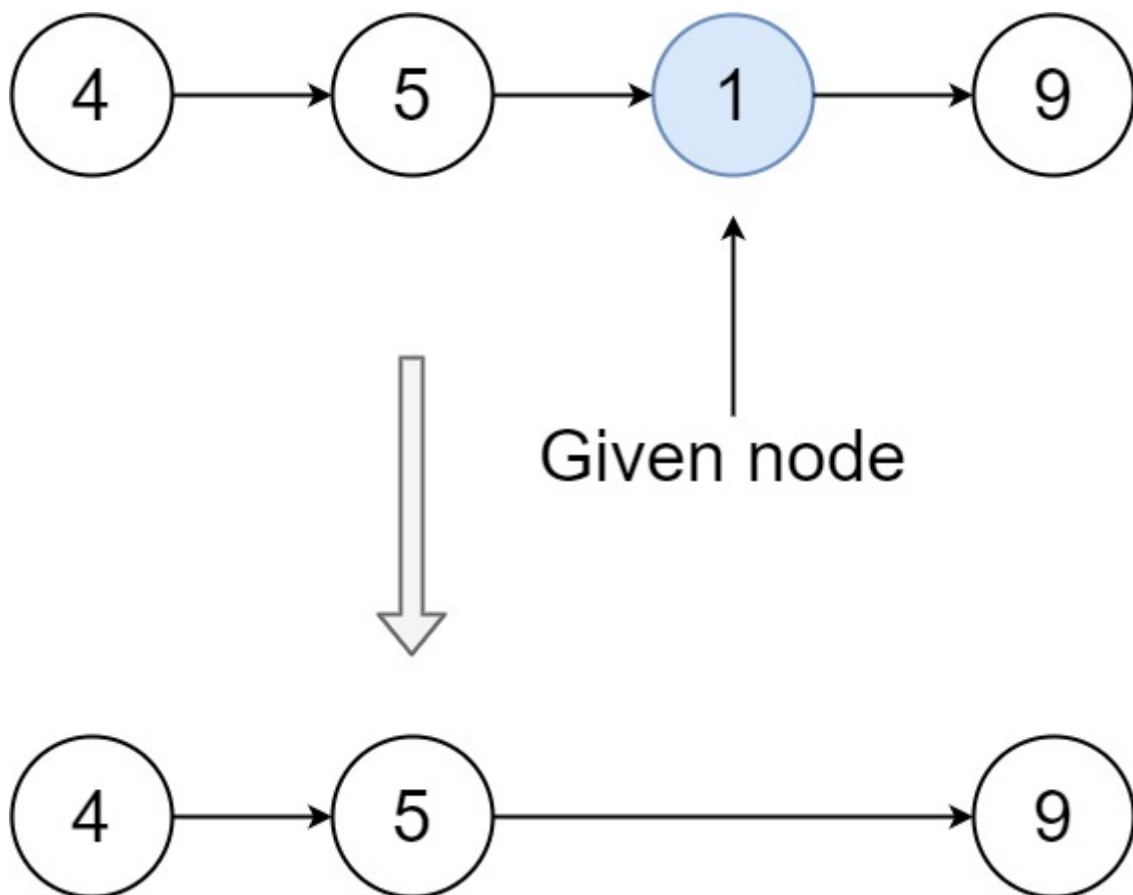
题目描述

请编写一个函数，用于 删除单链表中某个特定节点。在设计函数时需要注意，你无法访问链表的头节点 head，只能直接访问 要被删除的节点。

题目数据保证需要删除的节点 不是末尾节点。



- 1 示例1:
- 2 输入: head = [4,5,1,9], node = 5
- 3 输出: [4,1,9]
- 4 解释: 指定链表中值为 5 的第二个节点, 那么在调用了你的函数之后, 该链表应变为 4 -> 1 -> 9



- 1 示例2
- 2 输入: head = [4,5,1,9], node = 1
- 3 输出: [4,5,9]

```
4  解释：指定链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9
5  示例 3：
6
7  输入：head = [1,2,3,4]，node = 3
8  输出：[1,2,4]
9  示例 4：
10
11 输入：head = [0,1]，node = 0
12 输出：[1]
13 示例 5：
14
15 输入：head = [-3,5,-99]，node = -3
16 输出：[5,-99]
```

提示：

链表中节点的数目范围是 [2, 1000]

$-1000 \leq \text{Node.val} \leq 1000$

链表中每个节点的值都是唯一的

需要删除的节点 node 是链表中的一个有效节点，且不是末尾节点

思路

用当前节点的后一个节点覆盖当前节点的值，依次，最后删掉最后一个节点。

代码

```
1  package cn.edu.csust.leetcode;
2  /**
3   * Definition for singly-linked list.
4   * public class ListNode {
5   *     int val;
6   *     ListNode next;
7   *     ListNode(int x) { val = x; }
8   * }
9   */
10 class ListNode {
11     int val;
12     ListNode next;
13     ListNode(int x) { val = x; }
14 }
15 class Solution {
16     public void deleteNode1(ListNode node) {
17         while(node.next.next != null){
18             node.val = node.next.val;
19             node = node.next;
20         }
21         node.val = node.next.val;
22         node.next = null;
23     }
24     public void deleteNode2(ListNode node) {
25         while(node.next.next != null){
26             node.val = node.next.val;
27             node = node.next;
28         }
29         node.val = node.next.val;
30         node.next = node.next.next;
```

```
31     }  
32 }
```

提交结果成绩

执行结果: 通过 [显示详情](#)

[添加备注](#)

执行用时: **0 ms** , 在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗: **37.8 MB** , 在所有 Java 提交中击败了 **59.19%** 的用户

通过测试用例: **41 / 41**

炫耀一下:



[写题解, 分享我的解题思路](#)

提交结果	执行用时	内存消耗	语言	提交时间	备注
通过	0 ms	37.8 MB	Java	2021/11/02 08:23	添加备注

260.SingleNumber3

题目描述

给定一个整数数组 `nums`，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。你可以按 任意顺序 返回答案。

```
1  示例 1:  
2  输入: nums = [1,2,1,3,2,5]  
3  输出: [3,5]  
4  解释: [5, 3] 也是有效的答案。  
5  示例 2:  
6  
7  输入: nums = [-1,0]  
8  输出: [-1,0]  
9  示例 3:  
10  
11 输入: nums = [0,1]  
12 输出: [1,0]
```

提示:

$2 \leq \text{nums.length} \leq 3 \times 10^4$
 $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

解题思路

遍历第一遍，把所有的数都进行一个异或，得到最后结果的最高位，
再次遍历数组，与最高位进行与操作，如果不为0的，那么这些数必定存在一个单数，
如果为0，那么这些数也存在另一个单数，分成两组遍历

代码

```
1 class Solution{
2     public int[] singleNumber(int [] nums){
3         if(nums.length == 2){
4             return nums;
5         }
6         int sum = 0;
7         int [] res = new int[2];
8         for (int num: nums) {
9             sum ^= num;
10        }
11        int len = 1;
12        while((sum & 1) == 0) {
13            sum >>= 1;
14            len <<= 1;
15        }
16        for (int num: nums) {
17            if((num & len) != 0){
18                res[0] ^= num;
19            }
20            else{
21                res[1] ^= num;
22            }
23        }
24        return res;
25    }
26 }
```

```
1 | 时间beats:100%
2 | 空间beats:38%
```

268. Missing Number

题目描述

给定一个包含 $[0, n]$ 中 n 个数的数组 `nums`，找出 $[0, n]$ 这个范围内没有出现在数组中的那个数。

示例输入

```
1 | 示例 1:
2 |
3 | 输入: nums = [3,0,1]
4 | 输出: 2
5 | 解释: n = 3, 因为有 3 个数字, 所以所有的数字都在范围 [0,3] 内。2 是丢失的数字, 因为它没有出现在 nums 中。
6 | 示例 2:
7 |
8 | 输入: nums = [0,1]
9 | 输出: 2
10 | 解释: n = 2, 因为有 2 个数字, 所以所有的数字都在范围 [0,2] 内。2 是丢失的数字, 因为它没有出现在 nums 中。
11 | 示例 3:
12 |
13 | 输入: nums = [9,6,4,2,3,5,7,0,1]
```

```
14 输出: 8
15 解释: n = 9, 因为有 9 个数字, 所以所有的数字都在范围 [0,9] 内。8 是丢失的数字, 因为它没有出现在 nums 中。
16 示例 4:
17
18 输入: nums = [0]
19 输出: 1
20 解释: n = 1, 因为有 1 个数字, 所以所有的数字都在范围 [0,1] 内。1 是丢失的数字, 因为它没有出现在 nums 中。
```

提示:

$n == \text{nums.length}$
 $1 \leq n \leq 10^4$
 $0 \leq \text{nums}[i] \leq n$
nums 中的所有数字都独一无二

进阶: 你能否实现线性时间复杂度、仅使用额外常数空间的算法解决此问题?

思路

1.思路一:

开辟n + 1个空间, 用哈希, 如果哪一个空间没有被占用那么返回该值

2.思路二:

原地空间, 如果 $\text{num}[\text{num}[i]] \neq \text{num}[i]$,就交换

再次遍历查找如果 $\text{num}[i] \neq i$ 返回, 否则返回nums.length;

3.思路三:

遍历前首先做异或从0-n,做num[i]的异或, 返回异或结果

代码

```
1  package cn.edu.csust.leetcode.Array;
2
3  import java.util.Arrays;
4
5  class Solution {
6      // 开辟空间
7      // 执行结果: 通过
8      // 执行用时: 0 ms, 在所有 Java 提交中击败了100.00%的用户
9      // 内存消耗: 38.4 MB, 在所有 Java 提交中击败了96.43%的用户
10     public int missingNumber(int[] nums) {
11         int [] temp = new int [nums.length+1];
12         for (int num : nums) {
13             temp[num] = 1;
14         }
15         for (int i = 0; i <= nums.length; i++) {
16             if(temp[i] == 0)
17                 return i;
18         }
19         return 0;
20     }
21     // 原地空间
22     // 慢
```

```

23 // 时间beats:41.56%
24 // 空间beats:84.49%
25 public int missingNumber2(int[] nums) {
26     int n = nums.length;
27     for (int i = 0; i < n; i++) {
28         while(nums[i] < n && nums[nums[i]] != nums[i]){
29             int temp = nums[i];
30             nums[i] = nums[temp];
31             nums[temp] = temp;
32         }
33     }
34     for (int i = 0; i < nums.length; i++) {
35         if(nums[i] != i)
36             return i;
37     }
38     return n;
39 }
40 // 异或：常量空间
41
42 public int missingNumber3(int [] nums){
43     int sum = 0;
44     int n = nums.length;
45     for (int i = 0; i <= n; i++) {
46         sum ^= i;
47         if(i < n)
48             sum ^= nums[i];
49     }
50     return sum;
51 }
52 }
53 public class MissingNumber {
54     public static void main(String[] args) {
55         Solution solution = new Solution();
56         System.out.println(solution.missingNumber2(new int[]{3,0,1}));
57     }
58 }
59

```

执行结果： 通过 [显示详情](#)

[添加备注](#)

执行用时： 0 ms，在所有 Java 提交中击败了 100.00% 的用户

内存消耗： 38.4 MB，在所有 Java 提交中击败了 96.43% 的用户

通过测试用例： 122 / 122

炫耀一下：



[写题解，分享我的解题思路](#)

提交结果	执行用时	内存消耗	语言	提交时间	备注
通过	0 ms	38.4 MB	Java	2021/11/06 08:14	添加备注

299.BullsAndCows

题目描述:

你在和朋友一起玩 猜数字（Bulls and Cows）游戏，该游戏规则如下：

写出一个秘密数字，并请朋友猜这个数字是多少。朋友每猜测一次，你就会给他一个包含下述信息的提示：

猜测数字中有多少位属于数字和确切位置都猜对了（称为 "Bulls", 公牛），有多少位属于数字猜对了但是位置不对（称为 "Cows", 奶牛）。也就是说，这次猜测中有多少位非公牛数字可以通过重新排列转换成公牛数字。

给你一个秘密数字 `secret` 和朋友猜测的数字 `guess`，请你返回对朋友这次猜测的提示。

提示的格式为 "`xAyB`"，`x` 是公牛个数，`y` 是奶牛个数，`A` 表示公牛，`B` 表示奶牛。

请注意秘密数字和朋友猜测的数字都可能含有重复数字。

示例 1:

```
1  输入: secret = "1807", guess = "7810"
2  输出: "1A3B"
3  解释: 数字和位置都对（公牛）用 '|' 连接，数字猜对位置不对（奶牛）的采用斜体加粗标识。
4  "1807"
5      |
6  "7810"
```

示例 2:

```
1  输入: secret = "1123", guess = "0111"
2  输出: "1A1B"
3  解释: 数字和位置都对（公牛）用 '|' 连接，数字猜对位置不对（奶牛）的采用斜体加粗标识。
4  "1123"      "1123"
5      |         or      |
6  "0111"      "0111"
7  注意，两个不匹配的 1 中，只有一个会算作奶牛（数字猜对位置不对）。通过重新排列非公牛数字，其中仅有一个 1 可以成为公牛数字。
```

示例 3:

```
1  输入: secret = "1", guess = "0"
2  输出: "0A0B"
3  示例 4:
4
5  输入: secret = "1", guess = "1"
6  输出: "1A0B"
```

提示:

$1 \leq secret.length, guess.length \leq 1000$

$secret.length == guess.length$

`secret` 和 `guess` 仅由数字组成

思路:

- 相同的字符串统计countA,不同的字符统计 每个数字0-9在guess和secret中的最小次数, 即为位置不正确但是数字正确的数量

代码:

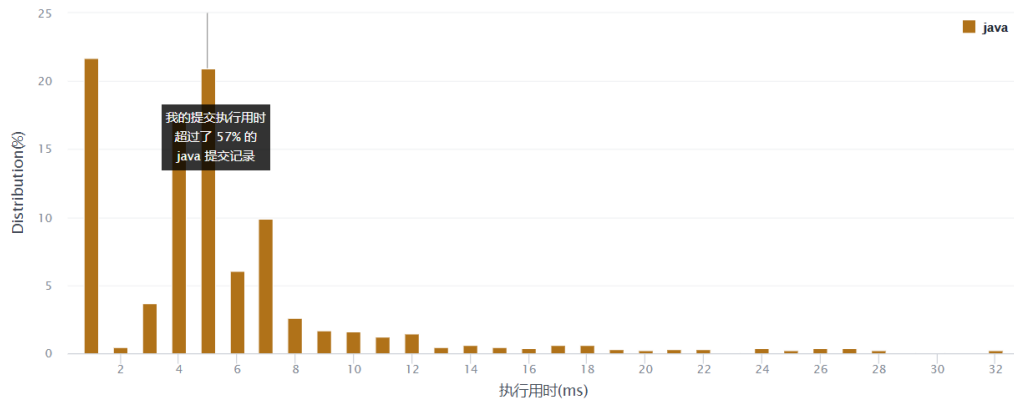
```
1 //思路:
2 class Solution {
3     public String getHint(String secret, String guess) {
4         int[] nums = new int[10];
5         int countA = 0, countB = 0;
6         for (int i = 0; i < secret.length(); i++) {
7             if(secret.charAt(i) == guess.charAt(i)) countA++;
8             else{
9                 if (nums[guess.charAt(i) - '0']-- > 0) countB++;
10                if(nums[secret.charAt(i) - '0']++ < 0) countB++;
11            }
12        }
13        return countA + "A" + countB + "B";
14    }
15    public String getHint(String secret, String guess) {
16        int countA = 0;
17        int countB = 0;
18        Map<Character, Integer> map = new HashMap<>();
19        List<Character> list = new ArrayList<>();
20        for (int i = 0; i < secret.length(); i++) {
21            char charSecret = secret.charAt(i);
22            char charGuess = guess.charAt(i);
23            if(charSecret == charGuess){
24                countA++;
25            }
26            else{
27                list.add(charGuess);
28                map.put(charSecret, map.getOrDefault(charSecret, 0) + 1);
29            }
30        }
31        for (int i = 0; i < list.size(); i++) {
32            char charGuess = list.get(i);
33            if(map.getOrDefault(charGuess, 0) > 0){
34                map.put(charGuess, map.get(charGuess) - 1);
35                if(map.get(charGuess) == 0){
36                    map.remove(charGuess);
37                }
38                countB++;
39            }
40        }
41        return countA + "A" + countB + "B";
42    }
43 }
```

提交记录

152 / 152 个通过测试用例
执行用时: 5 ms
内存消耗: 38.4 MB

状态: **通过**
提交时间: 24 分钟前

执行用时分布图表



301. Remove Invalid Parentheses

- 方法1: backtracking/dfs
- 易错点
- 方法2: bfs

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses (and).

Example 1:

```
1 Input: "()())()"
2 Output: ["()()()", "(())()"]
```

Example 2:

```
1 Input: "(a)())()"
2 Output: ["(a)()()", "(a())()"]
```

Example 3:

```
1 Input: ")("
2 Output: [""]
```

方法1: dfs

思路:

遍历第一遍统计有多少个非法的“(”，以及非法的“)”，赋值为 l 和 r，也就是交给 dfs 需要删除的括号数量。

开始 dfs。在 dfs 的过程中，退出条件是 $l = 0 \ \&\& \ r == 0$ ，此时可以合法推入一个结果。如果还没有删完，需要遍历字符串，尝试删除一个“(或)”（看哪个还需要）。这个过程中可以用到技巧剪枝：1. 如果当前的不是“(或)”，可以跳过，2. 对于有多种删除选择但是导致的结果相同时，我们选择跳过。这种情况仅发生在类似“()”或“(”这种情况，也就是出现多个重复的左右括号。那么为了减掉这些可以通过 $\text{if } (i \neq \text{start} \ \&\& \ s[i] == s[i - 1]) \text{ continue}$ 。start 是每次 dfs 传递进来的 string 需要开始尝试删除的位置。为什么

记录这个呢？因为这个值之前的字符如果需要我们都已经尝试过删除了，再从头开始会产生很多重复计算。而且利用这个数字也可以来去重。删除的操作是`s.erase(i, 1)`去掉这个字符。注意这时的`start`就会移交给 `i`，因为之前的我们也在本循环内都递归删除过了。

code

```
1 package cn.edu.csust.leetcode.stack;
2
3 import java.util.*;
4 // DFS
5 //1.这个题可以用堆栈，
6 // 但是由于只有小括号，
7 // 所以堆栈有点多余，
8 // 直接用一个统计变量统计是否有不匹配的左右括号即可
9 class Solution {
10 // 检测右括号检测是否多余
11 char[] pair = new char[]{'(', ')'};
12 char[] rePair = new char[]{'}', '('};
13 public List<String> removeInvalidParentheses(String s) {
14     List<String> result = new ArrayList<>();
15     removeHelper(s, result, 0, 0, pair);
16     return result;
17 }
18 private void removeHelper(String s, List<String> result, int lastI, int
lastJ, char [] pa){
19 // count用于记录有多少个不合法括号，count>0,左括号不合法，count < 0右括号不合法
20     int count = 0;
21     for (int i = lastI; i < s.length(); i++) {
22         if(s.charAt(i) == pa[0]){
23             count++;
24         }
25         if(s.charAt(i) == pa[1]){
26             count--;
27         }
28         if(count<0){
29             for (int j = lastJ; j <= i; j++) {
30                 if(s.charAt(j) == pa[1] && (j == lastJ || s.charAt(j - 1)
!= pa[1])){
31                     String newStr = s.substring(0, j) + s.substring(j +
1);
32                     removeHelper(newStr, result, i, j, pa);
33                 }
34             }
35             return;
36         }
37     }
38     StringBuilder stringBuilder = new StringBuilder(s);
39     stringBuilder.reverse();
40     s = stringBuilder.toString();
41     if(pa[0] == '('){
42         removeHelper(s, result, 0, 0, rePair);
43     }
44     else{
45         result.add(s);
46     }
47 }
```

```

48 }
49 // 测试类
50 public class RemoveInvalidParentheses {
51     public static void main(String[] args) {
52         Solution solution = new Solution();
53         System.out.println(solution.removeInvalidParentheses(")("("));
54     }
55 }
56

```

易错点

- 统计非法括号的子函数易错：如果用下面列的第二种累计cnt的方法，cnt不应该出现< -1，因为)))*(，右边的（没有immunity。
- 统计合法的时候（两次都包括）不要粗暴的else if，因为还可能有其他字符，要检查确实是“(”或者”)”。
- 这里的dfs不要用&，或者像下面的方法一样用&但是单独拷贝出一个current，因为要定点再恢复current不太容易。
- start的传递：移交给 i。
- 理解为什么要给一个start：为了避免重复尝试，并且这个start是指的current的start，每一次remove过之后原地传下去都不用+1。

方法二: BFS

code

```

1  class Solution {
2      public List<String> removeInvalidParentheses(String s) {
3          List<String> result = new ArrayList<>();
4          if (s.equals("(") || s.equals(")")) {
5              result.add(s);
6              return result;
7          }
8
9          Deque<String> queue = new ArrayDeque<>();
10         queue.offer(s);
11         HashSet<String> set = new HashSet<>(); //用于存储裁剪后的元素，防止重复
元素加入队列
12         boolean isFound = false; //判断是否找到了有效字符串
13
14         while (!queue.isEmpty()) { //队列不为空
15             String curr = queue.poll();
16             if (isValid(curr)) {
17                 result.add(curr);
18                 isFound = true;
19             }
20             if (isFound) { //找到后不再进行裁剪
21                 continue;
22             }
23             //裁剪过程
24             for (int i = 0; i < curr.length(); i++) {
25                 if (curr.charAt(i) == '(' || curr.charAt(i) == ')') { //只
对'('或')'进行裁剪
26                     String str;
27                     if (i == curr.length()-1) {

```

```

28         str = curr.substring(0, curr.length()-1);
29     } else {
30         str = curr.substring(0, i) + curr.substring(i+1);
31     }
32     if (set.add(str)) { //如果集合中还未有该字符串
33         queue.offer(str);
34     }
35 }
36 }
37 }
38
39 if (result.isEmpty()) {
40     result.add("");
41 }
42 return result;
43 }
44
45 private static boolean isValid(String s) {
46     int left = 0;
47     for (int i = 0; i < s.length(); i++) {
48         int curr = s.charAt(i);
49         if (curr == '(') {
50             left++;
51         } else if (curr == ')') {
52             if (left != 0) {
53                 left--;
54             } else {
55                 return false;
56             }
57         }
58     }
59     return left == 0;
60 }
61 }

```

318. Maximum Product of Word Lengths

题目描述

给定一个字符串数组 words，找到 $\text{length}(\text{word}[i]) * \text{length}(\text{word}[j])$ 的最大值，并且这两个单词不含有公共字母。你可以认为每个单词只包含小写字母。如果不存在这样的两个单词，返回 0。

```

1  示例 1:
2  输入: ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]
3  输出: 16
4  解释: 这两个单词为 "abcw", "xtfn"。
5
6  示例 2:
7  输入: ["a", "ab", "abc", "d", "cd", "bcd", "abcd"]
8  输出: 4
9  解释: 这两个单词为 "ab", "cd"。
10
11 示例 3:
12 输入: ["a", "aa", "aaa", "aaaa"]
13 输出: 0
14 解释: 不存在这样的两个单词。

```

思路

全是小写字母, 可以用一个32为整数表示一个word中出现的字母,
hash[i]存放第i个单词出现过的字母, a对应32位整数的最后一位,
b对应整数的倒数第二位, 依次类推. 时间复杂度 $O(N^2)$
判断两两单词按位与的结果, 如果结果为0且长度积大于最大积则更新

题解

```
1 package cn.edu.csust.leetcode;
2
3 class Solution {
4     public int maxProduct(String[] words) {
5         int [] hash = new int[words.length];
6         int max = 0;
7         for (int i = 0; i < words.length; i++) {
8             for (int j = 0; j < words[i].length(); j++) {
9                 hash[i] |= (1 << words[i].charAt(j) - 'a');
10            }
11        }
12        for (int i = 0; i < words.length; i++) {
13            for (int j = i+1; j < words.length; j++) {
14                if((hash[i] & hash[j]) == 0){
15                    max = Math.max(words[i].length() * words[j].length(),
16max);
17                }
18            }
19        }
20        return max;
21    }
22 }
23 public class MaxmumProductOfWordLengths {
24     public static void main(String[] args) {
25         Solution solution = new Solution();
26         String [] words = new String[]
27 {"abcw", "baz", "foo", "bar", "xtfn", "abcdef"};
28         System.out.println(solution.maxProduct(words));
29     }
30 }
```

319.bulb-switcher

题目描述

初始时有 n 个灯泡处于关闭状态。第一轮，你将会打开所有灯泡。接下来的第二轮，你将会每两个灯泡关闭一个。

第三轮，你每三个灯泡就切换一个灯泡的开关（即，打开变关闭，关闭变打开）。第 i 轮，你每 i 个灯泡就切换一个灯泡的开关。直到第 n 轮，你只需要切换最后一个灯泡的开关。

找出并返回 n 轮后有多少个亮着的灯泡。

示例 1:

```
1  输入: n = 3
2  输出: 1
3  解释:
4  初始时, 灯泡状态 [关闭, 关闭, 关闭].
5  第一轮后, 灯泡状态 [开启, 开启, 开启].
6  第二轮后, 灯泡状态 [开启, 关闭, 开启].
7  第三轮后, 灯泡状态 [开启, 关闭, 关闭].
8
9  你应该返回 1, 因为只有一个灯泡还亮着。
10 示例 2:
11
12 输入: n = 0
13 输出: 0
14 示例 3:
15
16 输入: n = 1
17 输出: 1
```

提示:

$0 \leq n \leq 10^9$

思路

中等题一般需要一个转换思想, 所以这个题是在求n以内平方数的多少

求n轮后亮着的灯泡?

- (1) 第i轮时, 被切换的灯泡位置是i的倍数。
- (2) 由 (1) 得出, 对于第p个灯泡来说, 只有其第“因子”轮才会切换, 若其有q个因子, 则最终被切换q次。因为初始状态是关闭状态, 那么因子数是奇数的灯泡最终是亮着的。
- (3) 只有平方数的因子个数不是成对出现, 举例: $4=1 \times 4, 2 \times 2$, 其因子是1,2,4。
- (4) 那么题目最终转化为1~n里平方数的个数, 进而转化为对n开平方根, 向下取整即可。

代码

```
1  package cn.edu.csust.leetcode;
2  //// 暴力和面向测试编程
3  //class Solution {
4  //    public int bulbSwitch(int n) {
5  //        if(n == 0)
6  //            return 0;
7  //        if(n <= 3)
8  //            return 1;
9  //        if(n == 999999999)
10 //            return 9999;
11 //        if(n == 1000000000)
12 //            return 10000;
13 //        boolean[] switches = new boolean[n + 1];
14 //        int count = 0;
15 //        for (int i = 1; i <= n; i++) {
16 //            for(int j = i; j <= n; j += i){
17 //                switches[j] = !switches[j];
18 //            }
19 //            if(switches[i])
20 //                count++;
21 //        }
```

```

22 //      return count;
23 //    }
24 //
25 //}
26 /**
27  * 初始有n个灯泡关闭
28  * 第i轮的操作是每i个灯泡切换一次开关（开->闭，闭->开），即切换i的倍数位置的开关。
29  * 求n轮后亮着的灯泡？
30  * （1）第i轮时，被切换的灯泡位置是i的倍数。
31  * （2）由（1）得出，对于第p个灯泡来说，只有其第“因子”轮才会切换，若其有q个因子，则最终被
    切换q次。因为初始状态是关闭状态，那么因子数是奇数的灯泡最终是亮着的。
32  * （3）只有平方数的因子个数不是成对出现，举例：4=1*4,2*2，其因子是1,2,4。
33  * （4）那么题目最终转化为1~n里平方数的个数，进而转化为对n开平方根，向下取整即可。
34  */
35 class Solution {
36     public int bulbSwitch(int n) {
37         return (int) Math.floor(Math.sqrt(n));
38     }
39 }
40 public class BulbSwitcher {
41     public static void main(String[] args) {
42         Solution solution = new Solution();
43         System.out.println(solution.bulbSwitch(3));
44     }
45 }
46

```

提交记录

灯泡开关

提交记录

35 / 35 个通过测试用例
 执行用时: 0 ms
 内存消耗: 34.9 MB

状态: **通过**
 提交时间: 7 分钟前

执行用时分布图表






335.路径交叉

题目描述

- 1 给定一个含有 n 个正数的数组 x 。从点 $(0,0)$ 开始，先向北移动 $x[0]$ 米，然后向西移动 $x[1]$ 米，向南移动 $x[2]$ 米，向东移动 $x[3]$ 米，持续移动。也就是说，每次移动后你的方位会发生逆时针变化。

2


```

3  编写一个  $O(1)$  空间复杂度的一趟扫描算法，判断你所经过的路径是否相交。
4
5
6
7  示例 1:
8
9  
10
11
12
13
14  输入: [2,1,1,2]
15  输出: true
16  示例 2:
17
18  
19
20
21
22
23
24  输入: [1,2,3,4]
25  输出: false
26  示例 3:
27
28  
29
30
31
32  输入: [1,1,1,1]
33  输出: true

```

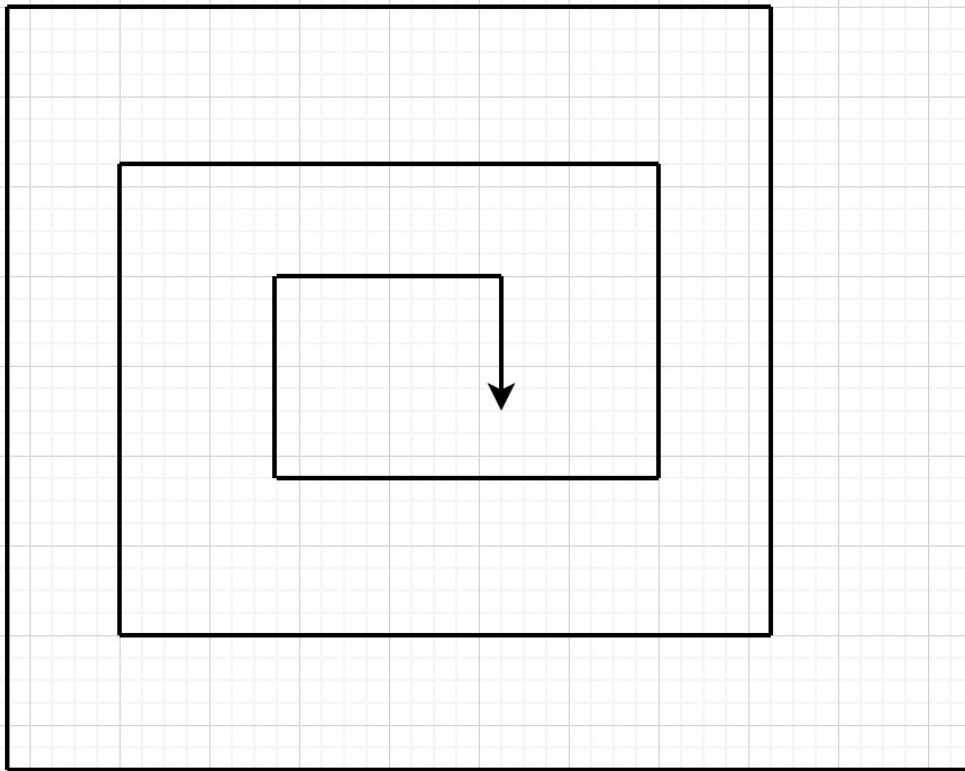
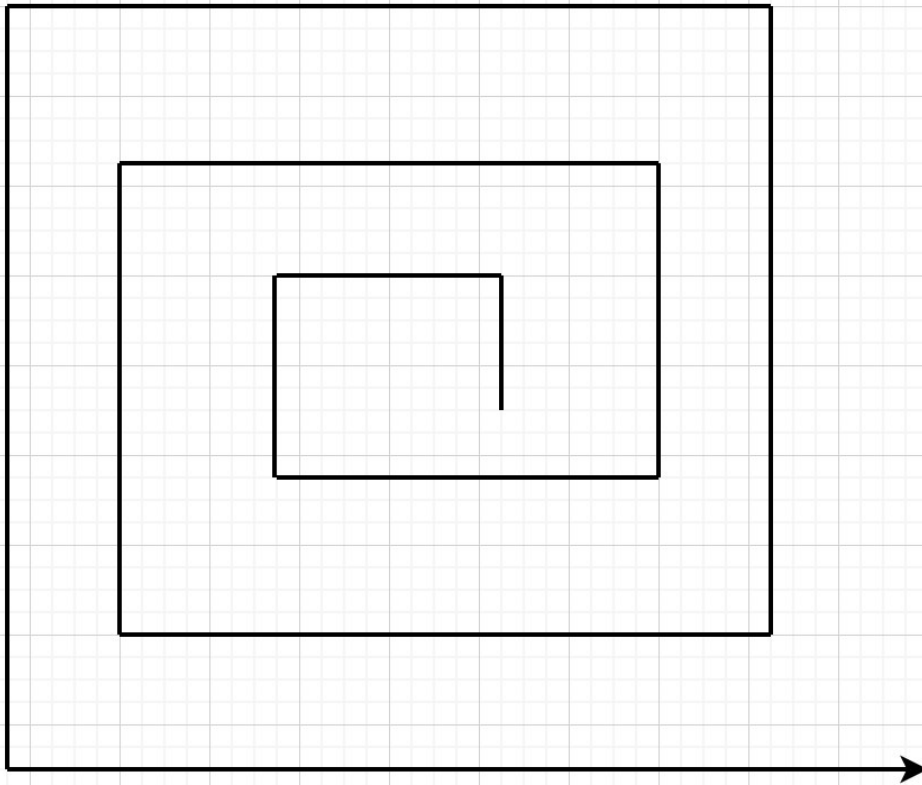
思路

符合直觉的做法是 $O(N)$ 时间和空间复杂度的算法。这种算法非常简单，但是题目要求我们使用空间复杂度为 $O(1)$ 的做法。

关于空间复杂度为 $O(N)$ 的算法可以参考我之前的[874.walking-robot-simulation](#)。思路基本是类似，只不过 obstacles（障碍物）不是固定的，而是我们不断遍历的时候动态生成的，我们每遇到一个点，就将其标记为 obstacle。随着算法的进行，我们的 obstacles 逐渐增大，最终和 N 一个量级。

我们考虑进行优化。我们仔细观察发现，如果想让其不相交，从大的范围来看只有两种情况：

1. 我们画的圈不断增大。
2. 我们画的圈不断减少。

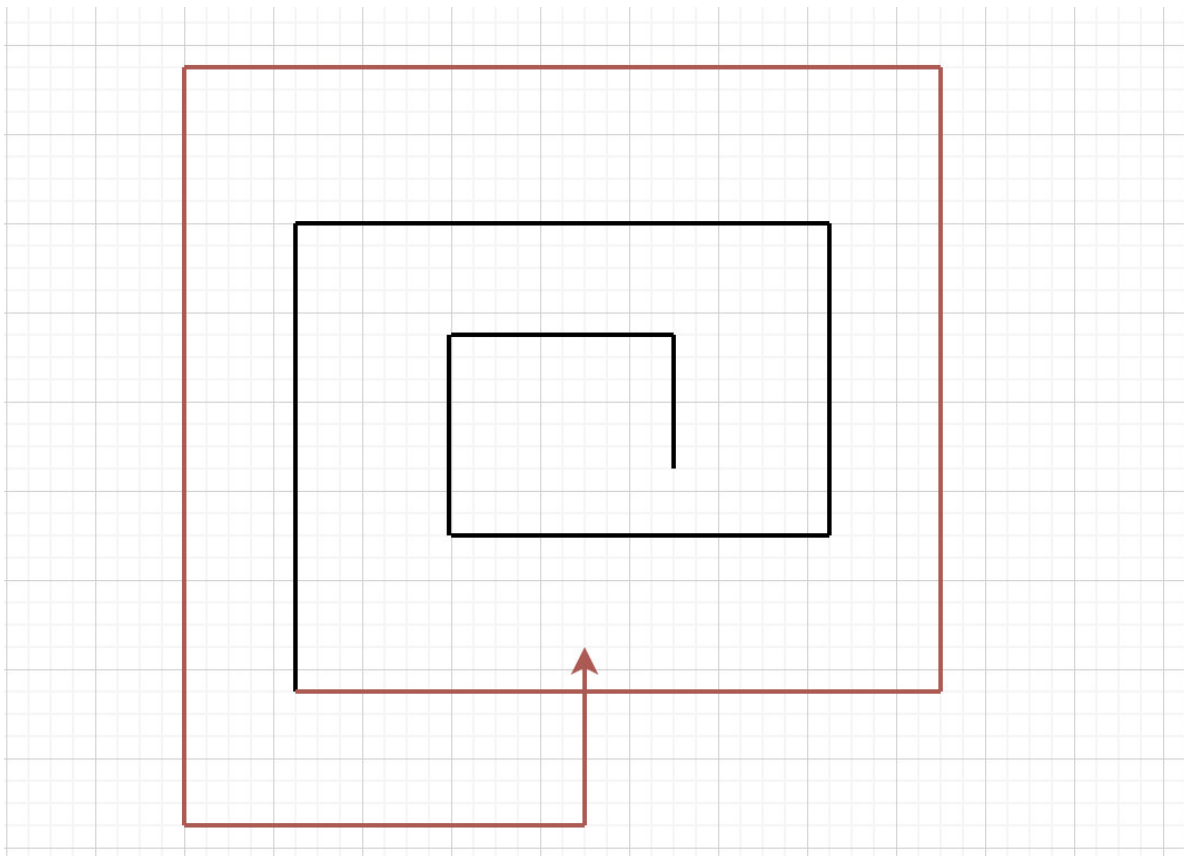


335.self-crossing

@lucifer

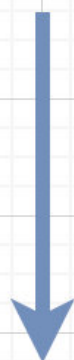
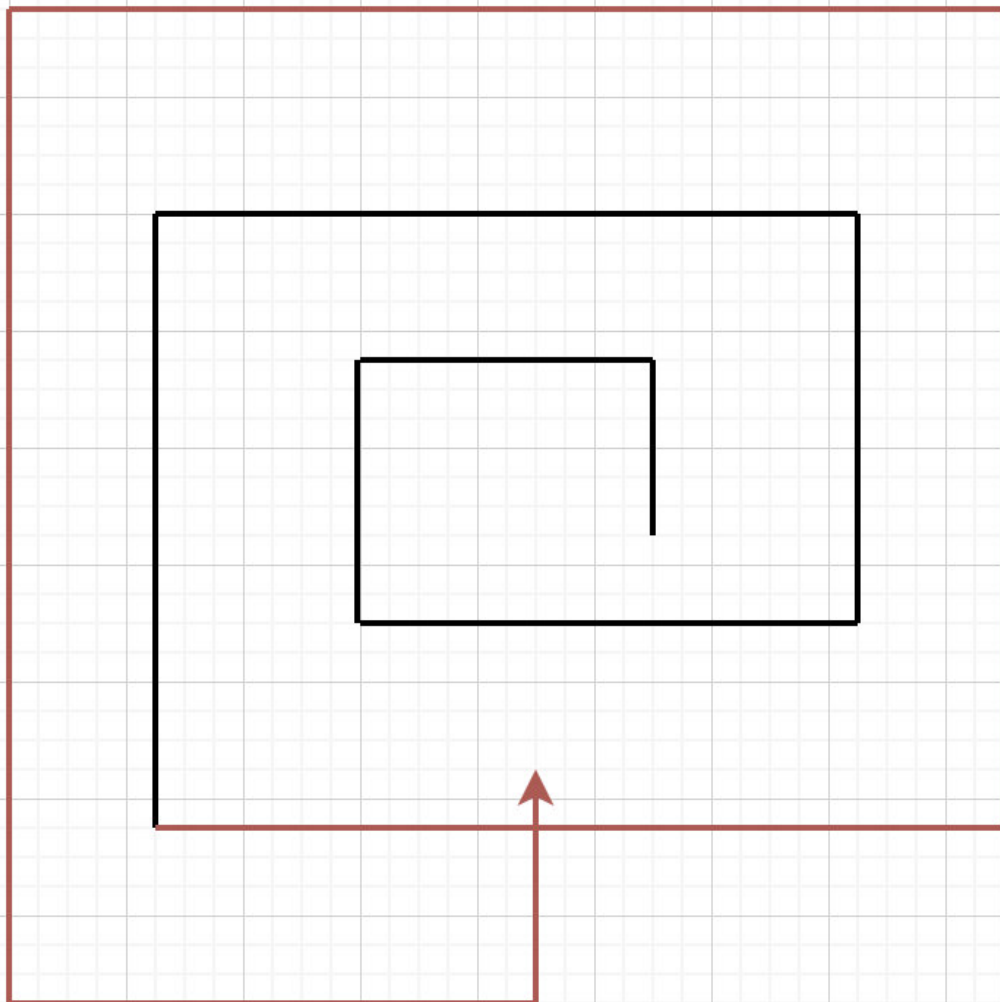
(有没有感觉像迷宫?)

这样我们会发现，其实我们画最新一笔的时候，并不是之前画的所有的都需要考虑，我们只需要最近的几个就可以了，实际上是最近的五个，不过不知道也没关系，我们稍后会讲解。

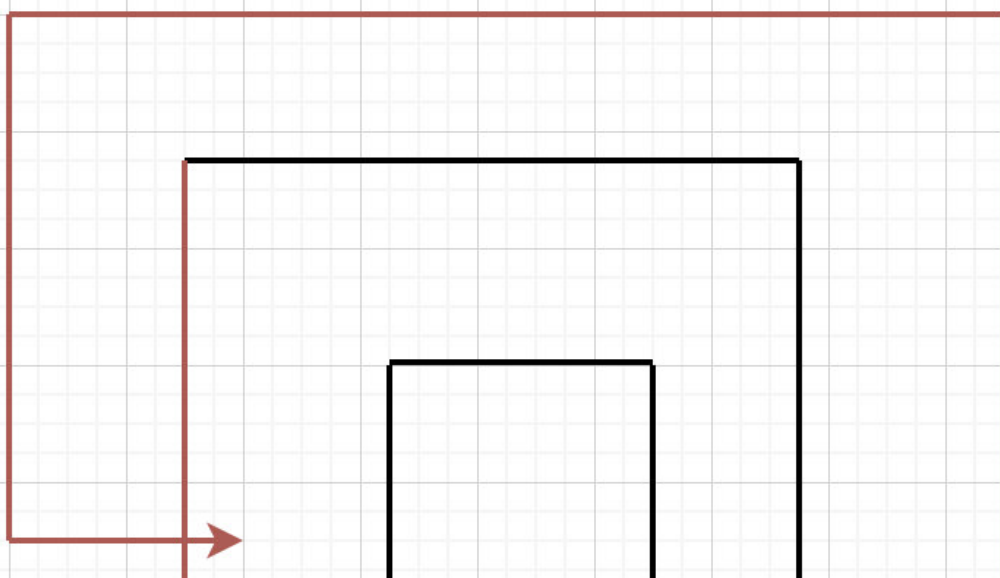


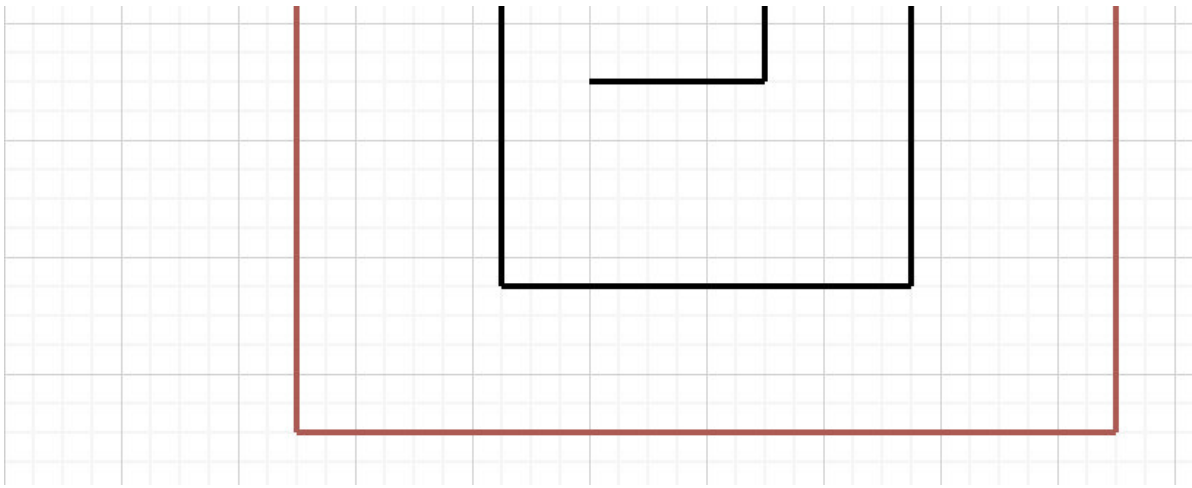
红色部分指的是我们需要考虑的，而剩余没有被红色标注的部分则无需考虑。不是因为我们无法与之相交，而是我们一旦与之相交，则必然我们也一定会与红色标记部分相交。

然而我们画的方向也是不用考虑的。比如我当前画的方向是从左到右，那和我画的方向是从上到下有区别么？在这里是没区别的，不信我帮你将上图顺时针旋转 90 度看一下：



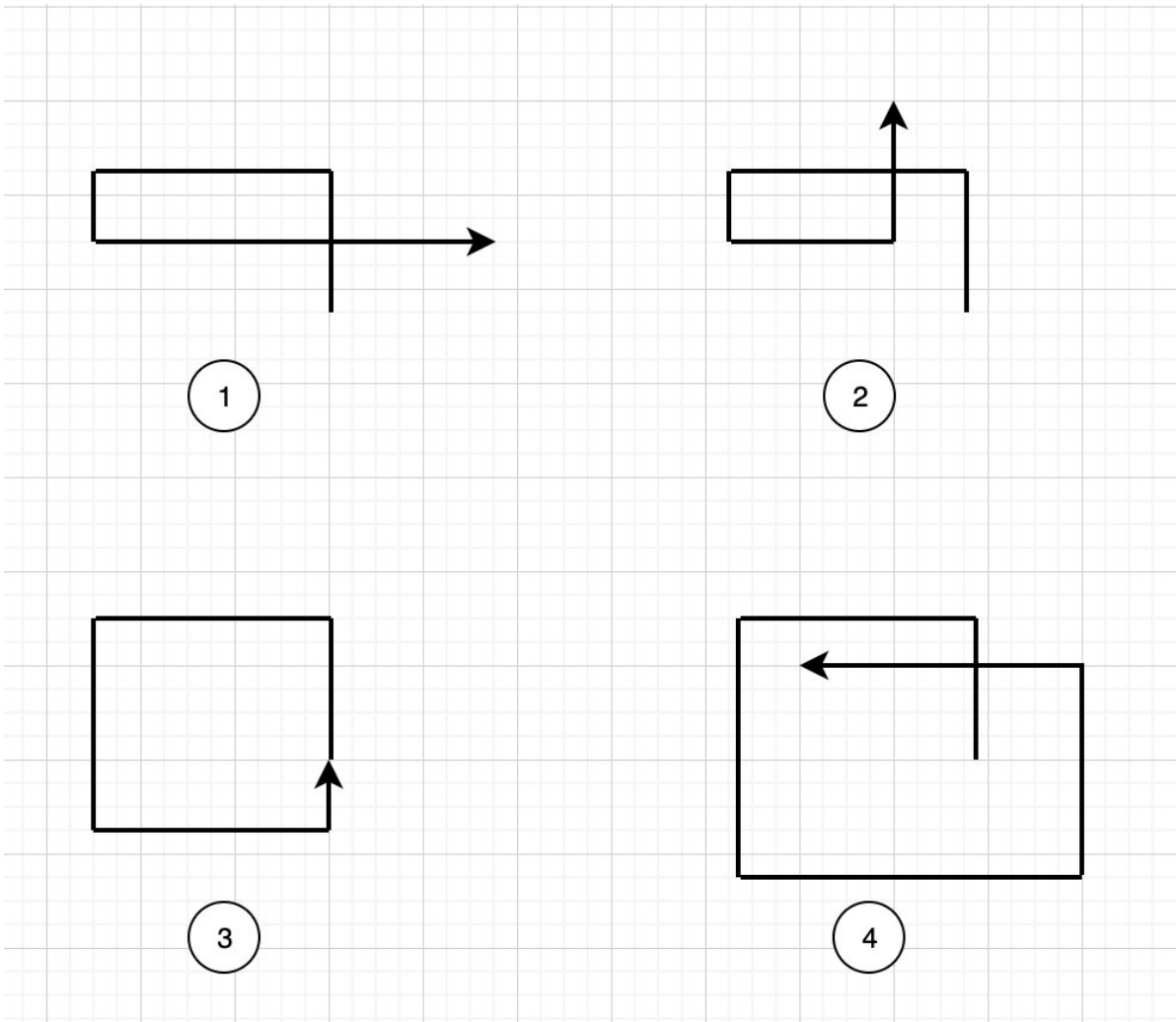
顺时针旋转90度





方向对于我们考虑是否相交没有差别。

当我们仔细思考的时候，会发现其实相交的情况只有以下几种：



这个时候代码就呼之欲出了。

- 我们只需要遍历数组 x ，假设当前是第 i 个元素。
- 如果 $x[i] \geq x[i-2]$ and $x[i-1] \leq x[i-3]$ ，则相交（第一种情况）
- 如果 $x[i-1] \leq x[i-3]$ and $x[i-2] \leq x[i]$ ，则相交（第二种情况）
- 如果 $i > 3$ and $x[i-1] == x[i-3]$ and $x[i] + x[i-4] == x[i-2]$ ，则相交（第三种情况）
- 如果 $i > 4$ and $x[i] + x[i-4] \geq x[i-2]$ and $x[i-1] \geq x[i-3] - x[i-5]$ \ and $x[i-1] \leq x[i-3]$ and $x[i-2] \geq x[i-4]$ and $x[i-3] \geq x[i-5]$ ，则相交（第四种情况）
- 否则不相交

代码

```
1 package cn.edu.csust.leetcode;
2 class Solution{
3     public boolean isSelfCrossing(int[] distance) {
4         int len = distance.length;
5         if(len < 4) return false;
6         if(distance[2] <= distance[0] && distance[3] >= distance[1]){
7             return true;
8         }
9         if(len > 4 && ((distance[3] <= distance[1] && distance[4] >=
distance[2]) ||
10             (distance[3] == distance[1] && distance[4] + distance[0] >=
distance[2]))){
11             return true;
12         }
13         for (int i = 5; i < len; i++) {
14             if(distance[i - 1] <= distance[i - 3] && distance[i] >=
distance[i - 2]){
15                 return true;
16             }
17             if (distance[i - 1] <= distance[i - 3] && distance[i - 4] <=
distance[i - 2]
18                 && distance[i] + distance[i - 4] >= distance[i - 2]
19                 && distance[i - 1] + distance[i - 5] >= distance[i - 3]){
20                 return true;
21             }
22         }
23         return false;
24     }
25 }
26 public class SelfCross {
27     public static void main(String[] args) {
28         Solution solution = new Solution();
29         System.out.println(solution.isSelfCrossing(new int []{1,2,3,4}));
30     }
31 }
```

367.有效完全平方数

题目描述

给定一个正整数 num，编写一个函数，如果 num 是一个完全平方数，则返回 true，否则返回 false。

进阶：不要使用任何内置的库函数，如 sqrt。

```
1 示例 1:
2 输入: num = 16
3 输出: true
4 示例 2:
5
6 输入: num = 14
7 输出: false
```

提示:

$$1 \leq num \leq 2^{31} - 1$$

思路

思路一：挨个找

思路二：二分找

代码

```
1 package cn.edu.csust.leetcode;
2 class Solution{
3     public boolean isPerfectSquare(int num){
4         int end = 1 << 16;
5         int temp= 0;
6         for (int i = 1; i < end; i++) {
7             if(temp == num){
8                 return true;
9             }
10            else if(temp > num){
11                return false;
12            }
13        }
14        return false;
15    }
16    // 凡是在线性有序数组中查找可以考虑二分法
17    public boolean isPerfectSquare2(int num) {
18        int low = 1;
19        int high = num;
20        while (low <= high) {
21            int mid = low + (high - low) / 2;
22            // int product = mid * mid; 越界
23            int t = num / mid;
24            if (t == mid) {
25                return num%mid == 0;
26            }
27            // low = mid + 1;
28            } else if (t < mid) {
29                high = mid - 1;
30            } else {
31                low = mid + 1;
32            }
33        }
34        return false;
35    }
36 }
37 }
```

执行结果: 通过 [显示详情 >](#)

[▶ 添加备注](#)

执行用时: **0 ms** , 在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗: **35.1 MB** , 在所有 Java 提交中击败了 **62.24%** 的用户

通过测试用例: **70 / 70**

炫耀一下:



[✍ 写题解, 分享我的解题思路](#)

提交结果	执行用时	内存消耗	语言	提交时间	备注
通过	0 ms	35.1 MB	Java	2021/11/04 08:58	▶ 添加备注

375. Guess number higher or lower2

题目描述

我们正在玩一个猜数游戏, 游戏规则如下:

我从 1 到 n 之间选择一个数字, 你来猜我选了哪个数字。

每次你猜错了, 我都会告诉你, 我选的数字比你的大了或者小了。

然而, 当你猜了数字 x 并且猜错了的时候, 你需要支付金额为 x 的现金。直到你猜到我选的数字, 你才算赢得了这个游戏。

```
1  示例:
2
3  n = 10, 我选择了8.
4
5  第一轮: 你猜我选择的数字是5, 我会告诉你, 我的数字更大一些, 然后你需要支付5块。
6  第二轮: 你猜是7, 我告诉你, 我的数字更大一些, 你支付7块。
7  第三轮: 你猜是9, 我告诉你, 我的数字更小一些, 你支付9块。
8
9  游戏结束。8 就是我选的数字。
10
11 你最终要支付 5 + 7 + 9 = 21 块钱。
12 给定 n ≥ 1, 计算你至少需要拥有多少现金才能确保你能赢得这
```

思路

动态规划, 选择从某一个值到了另一个值范围的最大花费, 并取与之前花费最小的那个值最小最大值问题。

代码

```
1  class Solution {
2      public int getMoneyAmount(int n) {
3          /**
4           dp[i][j]表示从[i,j]中猜出正确数字所需要的最少花费金额.(dp[i][i] = 0)
5           假设在范围[i,j]中选择x, 则选择x的最少花费金额为: max(dp[i][x-1], dp[x+1]
           [j]) + x
6       }
7   }
```



```

6      用max的原因是我们要计算最坏反馈情况下的最少花费金额(选了x之后，正确数字落在花费
      更高的那侧)
7
8      初始化为(n+2)*(n+2)数组的原因：处理边界情况更加容易，例如对于求解dp[1][n]时x
      如果等于1，需要考虑dp[0][1] (0不可能出现，dp[0][n]为0)
9      而当x等于n时，需要考虑dp[n+1][n+1] (n+1也不可能出现，dp[n+1][n+1]为0)
10
11     如何写出相应的代码更新dp矩阵，递推式dp[i][j] = max(max(dp[i][x-1],
      dp[x+1][j]) + x), x~[i:j]，可以画出矩阵图协助理解，可以发现
12     dp[i][x-1]始终在dp[i][j]的左部，dp[x+1][j]始终在dp[i][j]的下部，所以更新
      dp矩阵时i的次序应当遵循bottom到top的规则，j则相反，由于
13     i肯定小于等于j，所以我们只需要遍历更新矩阵的一半即可(下半矩阵)
14     */
15     int[][] dp = new int[n+2][n+2];
16     for(int i = n; i >= 1; --i) {
17         for(int j = i; j <= n; ++j) {
18             if(i == j)
19                 dp[i][j] = 0;
20             else {
21                 dp[i][j] = Integer.MAX_VALUE;
22                 for(int x = i; x <= j; ++x)
23                     dp[i][j] = Math.min(dp[i][j], Math.max(dp[i][x-1],
      dp[x+1][j]) + x);
24             }
25         }
26     }
27     return dp[1][n];
28 }
29 }
30
31 //思维和代码都比较简单的深度优先
32 class Solution {
33     public int getMoneyAmount(int n) {
34         int [][] dp = new int [n + 1][n + 1];
35         int ret = dfs(dp, 0, n);
36         return ret;
37     }
38
39     private int dfs(int[][] dp, int left, int right) {
40         if(left >= right){
41             return 0;
42         }
43         if(dp[left][right] != 0){
44             return dp[left][right];
45         }
46         int ret = 1000;
47         for(int i = left; i <= right; i++){
48             int cost = i + Math.max(dfs(dp, left, i - 1), dfs(dp, i + 1,
      right));
49             ret = Math.min(ret, cost);
50         }
51         dp[left][right] = ret;
52         return ret;
53     }
54 }

```

提交记录

执行结果：**通过** [显示详情 >](#)

[添加备注](#)

执行用时：**20 ms**，在所有 Java 提交中击败了 **53.37%** 的用户

内存消耗：**37.6 MB**，在所有 Java 提交中击败了 **36.24%** 的用户

通过测试用例：**27 / 27**

炫耀一下：



[写题解，分享我的解题思路](#)

提交结果	执行用时	内存消耗	语言	提交时间	备注
通过	20 ms	37.6 MB	Java	2021/11/12 09:02	添加备注
解答错误	N/A	N/A	Java	2021/11/12 08:32	添加备注

391. Perfect Rectangle

题目描述

给你一个数组 `rectangles`，其中 `rectangles[i] = [xi, yi, ai, bi]` 表示一个坐标轴平行的矩形。这个矩形的左下顶点是 (xi, yi) ，右上顶点是 (ai, bi) 。

如果所有矩形一起精确覆盖了某个矩形区域，则返回 `true`；否则，返回 `false`。

```
1  示例 1:
2
3
4  输入: rectangles = [[1,1,3,3],[3,1,4,2],[3,2,4,4],[1,3,2,4],[2,3,3,4]]
5  输出: true
6  解释: 5 个矩形一起可以精确地覆盖一个矩形区域。
7  示例 2:
8
9
10 输入: rectangles = [[1,1,2,3],[1,3,2,4],[3,1,4,2],[3,2,4,4]]
11 输出: false
12 解释: 两个矩形之间有间隔，无法覆盖成一个矩形。
13 示例 3:
14
15
16 输入: rectangles = [[1,1,3,3],[3,1,4,2],[1,3,2,4],[3,2,4,4]]
17 输出: false
18 解释: 图形顶端留有空缺，无法覆盖成一个矩形。
19 示例 4:
20
21
22 输入: rectangles = [[1,1,3,3],[3,1,4,2],[1,3,2,4],[2,2,4,4]]
23 输出: false
```

提示：
 $1 \leq \text{rectangles.length} \leq 2 * 10^4$
 $\text{rectangles}[i].\text{length} == 4$
 $-10^5 \leq xi, yi, ai, bi \leq 10^5$
思路

如果是完美矩形 那么一定满足两点：

- (1) 最左下 最左上 最右下 最右上 的四个点只出现一次 其他点成对出现
- (2) 四个点围城的矩形面积 = 小矩形的面积之和

代码

```

1  package cn.edu.csust.leetcode;
2
3  import java.util.HashSet;
4  import java.util.Set;
5
6  class Solution{
7      /**
8       * 如果是完美矩形 那么一定满足两点：
9       * (1) 最左下 最左上 最右下 最右上 的四个点只出现一次 其他点成对出现
10      * (2) 四个点围城的矩形面积 = 小矩形的面积之和
11      * @param rectangles 给定的矩形
12      * @return 返回能否形成完美矩形的结果
13      */
14      public boolean isRectangleCover(int [][] rectangles){
15
16
17          int left = 1000000;
18          int bottom = 1000000;
19          int right = 0;
20          int top = 0;
21
22          String lb = null;
23          String lt = null;
24          String rb = null;
25          String rt = null;
26
27          int areaSum = 0;
28          Set<String> set = new HashSet<>();
29
30          for (int i = 0; i < rectangles.length; i++) {
31              left = Math.min(left, rectangles[i][0]);
32              bottom = Math.min(bottom, rectangles[i][1]);
33              right = Math.max(right, rectangles[i][2]);
34              top = Math.max(top, rectangles[i][3]);
35
36              lb = (rectangles[i][0] + " " + rectangles[i][1]);
37              lt = (rectangles[i][0] + " " + rectangles[i][3]);
38              rb = (rectangles[i][2] + " " + rectangles[i][1]);
39              rt = (rectangles[i][2] + " " + rectangles[i][3]);
40
41              if(set.contains(lb)) set.remove(lb);

```

```

42         else set.add(lb);
43         if(set.contains(lt)) set.remove(lt);
44         else set.add(lt);
45         if(set.contains(rb)) set.remove(rb);
46         else set.add(rb);
47         if(set.contains(rt)) set.remove(rt);
48         else set.add(rt);
49
50         areaSum += (rectangles[i][2] - rectangles[i][0]) *
(rectangles[i][3] - rectangles[i][1]);
51     }
52
53     lb = left + " " + bottom;
54     lt = left + " " + top;
55     rb = right + " " + bottom;
56     rt = right + " " + top;
57     // 这一步判断是否完美镶嵌
58     boolean isCover = set.size() == 4 && set.contains(lb) &&
set.contains(lt) && set.contains(rb) && set.contains(rt);
59     // 这一步判断面积是否重叠，如果重叠新生成面积必定大于原来面积，如果有空缺那么新生
成面积小于原来面积
60     isCover = isCover && (right - left) * (top - bottom) == areaSum;
61
62     return isCover;
63 }
64 }
65 public class PerfectRectangle {
66     public static void main(String[] args) {
67         Solution solution = new Solution();
68         // int [][] rectangles = {{1,1,3,3},{3,1,4,2},{1,3,2,4},{3,2,4,4}};
69         int [][] rectangles = {{1,1,3,3},{3,1,4,2},{1,3,2,4},{2,2,4,4}};
70         System.out.println(solution.isRectangleCover(rectangles));
71     }
72 }
73

```

扫描线算法

```

1  class Solution {
2      public boolean isRectangleCover(int[][] rectangles) {
3          int n = rectangles.length;
4          int[][] rs = new int[n * 2][4];
5          for (int i = 0, idx = 0; i < n; i++) {
6              int[] re = rectangles[i];
7              rs[idx++] = new int[]{re[0], re[1], re[3], 1};
8              rs[idx++] = new int[]{re[2], re[1], re[3], -1};
9          }
10         Arrays.sort(rs, (a,b)->{
11             if (a[0] != b[0]) return a[0] - b[0];
12             return a[1] - b[1];
13         });
14         n *= 2;
15         // 分别存储相同的横坐标下「左边的线段」和「右边的线段」 (y1, y2)
16         List<int[]> l1 = new ArrayList<>(), l2 = new ArrayList<>();
17         for (int l = 0; l < n; ) {
18             int r = l;
19             l1.clear();

```

```

20         l2.clear();
21         // 找到横坐标相同部分
22         while (r < n && rs[r][0] == rs[l][0]) r++;
23         for (int i = l; i < r; i++) {
24             int[] cur = new int[]{rs[i][1], rs[i][2]};
25             List<int[]> list = rs[i][3] == 1 ? l1 : l2;
26             if (list.isEmpty()) {
27                 list.add(cur);
28             } else {
29                 int[] prev = list.get(list.size() - 1);
30                 if (cur[0] < prev[1]) return false; // 存在重叠
31                 else if (cur[0] == prev[1]) prev[1] = cur[1]; // 首尾相连
32                 else list.add(cur);
33             }
34         }
35         if (l > 0 && r < n) {
36             // 若不是完美矩形的边缘竖边，检查是否成对出现
37             if (l1.size() != l2.size()) return false;
38             for (int i = 0; i < l1.size(); i++) {
39                 if (l1.get(i)[0] == l2.get(i)[0] && l1.get(i)[1] ==
12.get(i)[1]) continue;
40                 return false;
41             }
42         } else {
43             // 若是完美矩形的边缘竖边，检查是否形成完整一段
44             if (l1.size() + l2.size() != 1) return false;
45         }
46         l = r;
47     }
48     return true;
49 }
50 }

```

提交记录

执行结果: **通过** [显示详情 >](#)

[添加备注](#)

执行用时: **57 ms**, 在所有 Java 提交中击败了 **17.65%** 的用户

内存消耗: **47.5 MB**, 在所有 Java 提交中击败了 **44.70%** 的用户

通过测试用例: **47 / 47**

炫耀一下:



[写题解, 分享我的解题思路](#)

提交结果	执行用时	内存消耗	语言	提交时间	备注
通过	57 ms	47.5 MB	Java	2021/11/16 09:24	添加备注

407.接雨水2(difficult)

题目描述

给你一个 $m \times n$ 的矩阵, 其中的值均为非负整数, 代表二维高度图每个单元的高度, 请计算图中形状最多能接多少体积的雨水。

```
1  示例 1:
2
3
4
5  输入: heightMap = [[1,4,3,1,3,2],[3,2,1,3,2,4],[2,3,3,2,3,1]]
6  输出: 4
7  解释: 下雨后, 雨水将会被上图蓝色的方块中。总的接雨水量为1+2+1=4。
8  示例 2:
9
10
11
12 输入: heightMap = [[3,3,3,3,3],[3,2,2,2,3],[3,2,1,2,3],[3,2,2,2,3],
13 输出: 10
```

提示:

$m == heightMap.length$

$n == heightMap[i].length$

$1 \leq m, n \leq 200$

$0 \leq heightMap[i][j] \leq 2 * 10^4$

思路

从最外层的围墙的最短层往里层找，如果往里找到的块高度比边界框的高度小，那么就计算高度差累加盛水量，如果比边界高，就将这个高度作为新的围墙高度

代码

```
1 // 构建堆
2 // 调整小根堆
3 // 出堆入堆
4 // 深度优先
5 class Node {
6     int height;
7     int row;
8     int col;
9     // 全参构造
10    public Node(int height, int row, int col){
11        this.height = height;
12        this.row = row;
13        this.col = col;
14    }
15    // 无参构造
16    public Node(){
17
18    }
19 }
20 class solution{
21     /**
22      * 根据当前的围墙构造，返回当前的围墙可盛水数量，一个立方的水为一个单位
23      * 先把最外围的一圈作为围栏， 选择一个最低的围栏，
24      * 如果这个围栏的邻节点都比它大， 此围栏可删除，
25      * 邻节点作为新的围栏； 如果邻节点比它小，
26      * 那么邻节点可储蓄的水为 二者高度之差，
27      * 此时在邻节点设置围栏，高度为当前围栏高度即可。
28      * @param heightMap 围墙高度图
29      * @return 返回最大可盛水量
30      */
31    public int trapRainwater(int[][] heightMap) {
32        int m = heightMap.length;
33        int n = heightMap[0].length;
34        boolean [][] visited = new boolean[m][n];
35        int sum = 0, maxH = 0;
36        if(m <= 2 || n <= 2)
37            return 0;
38        // 初始化堆
39        PriorityQueue<Node> queue = new PriorityQueue<>((o1, o2) ->
40            o1.height - o2.height);
41        // Node [] heap = new Node[m * n];
42        int countEnd = 0;
43        // 遍历矩阵周围， 入堆，调整堆得到最小的围墙高度
44        for(int i = 0; i < m; i++){
45            queue.add(new Node(heightMap[i][0], i, 0));
46            queue.add(new Node(heightMap[i][n - 1], i, n - 1));
47            visited[i][0] = true;
48            visited[i][n - 1] = true;
49        }
50        for(int i = 1; i < n - 1; i++){
```

```

50         queue.add(new Node(heightMap[0][i], 0, i));
51         queue.add(new Node(heightMap[m - 1][i], m - 1, i));
52         visited[0][i] = true;
53         visited[m - 1][i] = true;
54     }
55     while(!queue.isEmpty()){
56         // 出堆
57         Node node = queue.poll();
58         int [][] direction = new int[][]{
59             {node.row + 1, node.col},
60             {node.row - 1, node.col},
61             {node.row, node.col + 1},
62             {node.row, node.col - 1}};
63         maxH = Math.max(maxH, node.height);
64         sum += maxH - node.height;
65         for (int i = 0; i < 4; i++) {
66             int nrow = direction[i][0];
67             int ncol = direction[i][1];
68             // 边界判断
69             if(nrow < 0 || nrow >= m || ncol < 0 || ncol >= n ||
visited[nrow][ncol]){
70                 continue;
71             }
72             // 标记访问
73             visited[nrow][ncol] = true;
74             // 新的围墙入堆
75             queue.add(new Node(heightMap[nrow][ncol], nrow, ncol));
76         }
77     }
78     // 调整堆
79     // minHeapDown(heap, countStart - 1, countEnd - 1);
80 }
81 return sum;
82 }
83
84 /**
85  * 根据当前的围墙构造，返回当前的围墙可盛水数量，一个立方的水为一个单位
86  * 先把最外围的一圈作为围栏， 选择一个最低的围栏，
87  * 如果这个围栏的邻节点都比它大， 此围栏可删除，
88  * 邻节点作为新的围栏； 如果邻节点比它小，
89  * 那么邻节点可储蓄的水为 二者高度之差，
90  * 此时在邻节点设置围栏， 高度为当前围栏高度即可。
91  * @param heightMap 围墙高度图
92  * @return 返回最大可盛水量
93  */
94 public int trapRainwater2(int[][] heightMap) {
95     int m = heightMap.length;
96     int n = heightMap[0].length;
97     boolean [][] visited = new boolean[m][n];
98     int sum = 0, maxH = 0;
99     if(m <= 2 || n <= 2)
100         return 0;
101     // 初始化堆
102     PriorityQueue<Node> queue = new PriorityQueue<>((o1, o2) ->
o1.height - o2.height);
103     // Node [] heap = new Node[m * n];
104     int countEnd = 0;
105     // 遍历矩阵周围， 入堆， 调整堆得到最小的围墙高度

```



```

106         for(int i = 0; i < m;i++){
107             queue.add(new Node(heightMap[i][0], i, 0));
108             queue.add(new Node(heightMap[i][n - 1], i, n - 1));
109             visited[i][0] = true;
110             visited[i][n - 1] = true;
111         }
112         for(int i = 1; i < n - 1;i++){
113             queue.add(new Node(heightMap[0][i], 0, i));
114             queue.add(new Node(heightMap[m - 1][i], m - 1, i));
115             visited[0][i] = true;
116             visited[m - 1][i] = true;
117         }
118         while(!queue.isEmpty()){
119             // 出堆
120             Node node = queue.poll();
121             int [][] direction = new int[][]{
122                 {node.row + 1, node.col},
123                 {node.row - 1, node.col},
124                 {node.row, node.col + 1},
125                 {node.row, node.col - 1}};
126             for (int i = 0; i < 4; i++) {
127                 int nrow = direction[i][0];
128                 int ncol = direction[i][1];
129                 // 边界判断
130                 if(nrow >= 0 && nrow < m && ncol >= 0 && ncol < n &&
!visited[nrow][ncol]){
131                     // 标记访问
132                     visited[nrow][ncol] = true;
133                     // 新的围墙入堆
134                     queue.add(new Node(heightMap[nrow][ncol], nrow, ncol));
135                     if(node.height > heightMap[nrow][ncol]){
136                         sum += node.height - heightMap[nrow][ncol];
137                         heightMap[nrow][ncol] = node.height;
138                     }
139                 }
140
141
142             }
143             // 调整堆
144             // minHeapDown(heap, countStart - 1, countEnd - 1);
145         }
146         return sum;
147     }
148 }

```

42 / 42 个通过测试用例

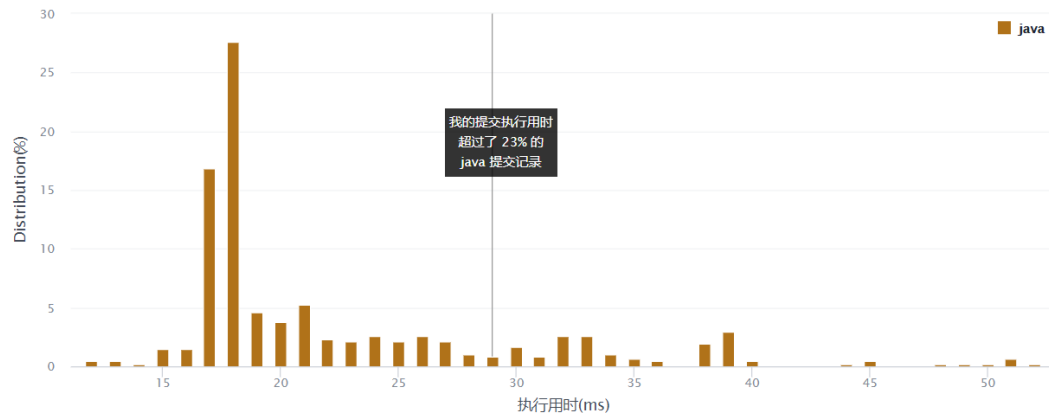
执行用时: 29 ms

内存消耗: 41.8 MB

状态: 通过

提交时间: 24 分钟前

执行用时分布图表



488 ZumaGame

题目描述

你正在参与祖玛游戏的一个变种。

在这个祖玛游戏变体中，桌面上有一排彩球，每个球的颜色可能是：红色 'R'、黄色 'Y'、蓝色 'B'、绿色 'G' 或白色 'W'。你的手中也有一些彩球。

你的目标是 清空 桌面上所有的球。每一回合：

从你手上的彩球中选出 任意一颗，然后将其插入桌面上那一排球中：两球之间或这一排球的任一端。

接着，如果有出现 三个或者三个以上 且 颜色相同 的球相连的话，就把它移除掉。

如果这种移除操作同样导致出现三个或者三个以上且颜色相同的球相连，则可以继续移除这些球，直到不再满足移除条件。

如果桌面上所有球都被移除，则认为你赢得本场游戏。

重复这个过程，直到你赢了游戏或者手中没有更多的球。

给你一个字符串 `board`，表示桌面上最开始的那排球。另给你一个字符串 `hand`，表示手里的彩球。请你按上述操作步骤移除掉桌上所有球，计算并返回所需的 最少 球数。如果不能移除桌上所有的球，返回 `-1`。

```
1  示例 1:
2
3  输入: board = "WRRBBW", hand = "RB"
4  输出: -1
5  解释: 无法移除桌面上的所有球。可以得到的最好局面是:
6
7  - 插入一个 'R'，使桌面变为 WRRRBBW。WRRRBBW -> WBBW
8  - 插入一个 'B'，使桌面变为 WBBBW。WBBBW -> WW
9    桌面上还剩下球，没有其他球可以插入。
10  示例 2:
11
12  输入: board = "WRRBBBW", hand = "WRBRW"
13  输出: 2
14  解释: 要想清空桌面上的球，可以按下述步骤:
15
```

```

16 - 插入一个 'R' , 使桌面变为 WRRRBBWW 。WRRRBBWW -> WWBBWW
17 - 插入一个 'B' , 使桌面变为 WWBBWW 。WWBBWW -> WWW -> empty
18     只需从手中出 2 个球就可以清空桌面。
19     示例 3:
20
21 输入: board = "G", hand = "GGGGG"
22 输出: 2
23 解释: 要想清空桌面上的球, 可以按下述步骤:
24
25 - 插入一个 'G' , 使桌面变为 GG 。
26 - 插入一个 'G' , 使桌面变为 GGG 。GGG -> empty
27     只需从手中出 2 个球就可以清空桌面。
28     示例 4:
29
30 输入: board = "RBYBBRRB", hand = "YRBGB"
31 输出: 3
32 解释: 要想清空桌面上的球, 可以按下述步骤:
33
34 - 插入一个 'Y' , 使桌面变为 RBYBBRRB 。RBYBBRRB -> RBBRRB -> RRRB -> B
35 - 插入一个 'B' , 使桌面变为 BB 。
36 - 插入一个 'B' , 使桌面变为 BBB 。BBB -> empty
37     只需从手中出 3 个球就可以清空桌面。

```

提示:

1 <= board.length <= 16

1 <= hand.length <= 5

board 和 hand 由字符 'R'、'Y'、'B'、'G' 和 'W' 组成

桌面上一开始的球中, 不会有三个及三个以上颜色相同且连着的球

思路

总体思路是进行深度优先试探

实施的过程中有很多的trick,

比如说在深度的时候不一定非要等到将所有的球或者步骤都试探完才结束, 而是进行适当的剪枝, 降低递归深度增加运行效率, 节约运行空间

还有就是消除的时候可以使用两个指针, 双向扫描达到效率翻倍的效果。

代码:

```

1  package cn.edu.csust.leetcode;
2
3  import java.util.Arrays;
4  import java.util.HashMap;
5  import java.util.Map;
6
7  class Solution{
8      private static final int MAX = 0x3F3F3F3F;
9      public int findMinStep(String board, String hand){
10         int answer = memorization(board, hand, new HashMap<>(), 1 <<
hand.length());
11         return answer == MAX? -1:answer;
12     }
13

```

```

14     private int memorization(String board, String hand, HashMap<String,
Integer> cache, int cur) {
15     //     如果待消除的串长度为0, 则返回0,
16     if(board.length() == 0){
17         return 0;
18     }
19     if(cache.containsKey(board)) return cache.get(board);
20
21     int ans = MAX;
22     // 遍历手中所有的球
23     for (int i = 0; i < hand.length(); i++) {
24         if(((cur >> i) & 1) == 1){
25             continue;
26         }
27         int next = (1 << i) | cur;
28
29     }
30 }
31 }
32 public class ZumaGame {
33     public static void main(String[] args) {
34         Solution solution = new Solution();
35         String board = "WWRBWBW";
36         String hand = "WRBRW";
37         System.out.println(solution.findMinStep(board, hand));
38     }
39 }

```

495. TeemoAttacking

题目描述

在《英雄联盟》的世界中，有一个叫“提莫”的英雄。他的攻击可以让敌方英雄艾希（编者注：寒冰射手）进入中毒状态。

当提莫攻击艾希，艾希的中毒状态正好持续 `duration` 秒。

正式地讲，提莫在 `t` 发起发起攻击意味着艾希在时间区间 `[t, t + duration - 1]`（含 `t` 和 `t + duration - 1`）处于中毒状态。如果提莫在中毒影响结束 前 再次攻击，中毒状态计时器将会 重置，在新的攻击之后，中毒影响将会在 `duration` 秒后结束。

给你一个 非递减 的整数数组 `timeSeries`，其中 `timeSeries[i]` 表示提莫在 `timeSeries[i]` 秒时对艾希发起攻击，以及一个表示中毒持续时间的整数 `duration`。

返回艾希处于中毒状态的 总 秒数。

```

1  示例 1:
2
3  输入: timeSeries = [1,4], duration = 2
4  输出: 4
5  解释: 提莫攻击对艾希的影响如下:
6
7  - 第 1 秒，提莫攻击艾希并使其立即中毒。中毒状态会维持 2 秒，即第 1 秒和第 2 秒。
8  - 第 4 秒，提莫再次攻击艾希，艾希中毒状态又持续 2 秒，即第 4 秒和第 5 秒。
9      艾希在第 1、2、4、5 秒处于中毒状态，所以总中毒秒数是 4 。
10  示例 2:
11
12 输入: timeSeries = [1,2], duration = 2

```

```
13 输出：3
14 解释：提莫攻击对艾希的影响如下：
15
16 - 第 1 秒，提莫攻击艾希并使其立即中毒。中毒状态会维持 2 秒，即第 1 秒和第 2 秒。
17 - 第 2 秒，提莫再次攻击艾希，并重置中毒计时器，艾希中毒状态需要持续 2 秒，即第 2 秒和第 3 秒。
18 艾希在第 1、2、3 秒处于中毒状态，所以总中毒秒数是 3 。
```

提示：

$1 \leq \text{timeSeries.length} \leq 10^4$

$0 \leq \text{timeSeries}[i], \text{duration} \leq 10^7$

timeSeries 按 非递减 顺序排列

思路

当前与前一个的时刻进行比较，如果这个时间段大于duration，则加上duration，否则加上这一个时间段，最后返回累计求和

代码

```
1  class Solution {
2      public int findPoisonedDuration(int[] timeSeries, int duration) {
3          if(duration == 0){
4              return 0;
5          }
6          int res=0;
7          for(int i=1;i<timeSeries.length;i++){
8              if(timeSeries[i]<=timeSeries[i-1]+duration){
9                  res+=timeSeries[i]-timeSeries[i-1];
10             }
11             else res+= duration;
12         }
13         res +=duration;
14         return res;
15     }
16 }
```



提交结果



执行通过

38 / 38 个通过测试用例



执行用时 2 ms

击败用户 90.5%



内存消耗 40.2 MB

击败用户 53.5%

恭喜完成今日打卡任务

奖励已发放，连续打卡 16 天

查看打卡成绩单

分享一下



微信



朋友圈



QQ



微博

提交的代码

添加备注 ▶

Java | 刚刚

写题解，分享我的解题思路

500. 键盘行

题目描述

给你一个字符串数组 `words`，只返回可以使用在 美式键盘 同一行的字母打印出来的单词。键盘如下图所示。

美式键盘 中：

第一行由字符 `"qwertyuiop"` 组成。

第二行由字符 `"asdfghjkl"` 组成。

第三行由字符 `"zxcvbnm"` 组成。

提示：

$1 \leq words.length \leq 20$

$1 \leq words[i].length \leq 100$

`words[i]` 由英文字母（小写和大写字母）组成

```
1  示例 1：
2  输入：words = ["Hello","Alaska","Dad","Peace"]
3  输出：["Alaska","Dad"]
4
5  示例 2：
6  输入：words = ["omk"]
7  输出：[]
8
9  示例 3：
10 输入：words = ["adsdf","sfd"]
11 输出：["adsdf","sfd"]
```

思路

首先将26个字母的行数记录在一个整型数组中,挨个遍历字符串以及他们的字符，如果全都相同则添加到结果集中，否则舍弃，最后返回结果

代码

```
1 class Solution{
2     public String[] findwords(String[] words) {
3         int [] alphaRow = new int[]{2, 3, 3, 2, 1, 2, 2, 2, 1, 2, 2, 2, 3,
4         3, 1, 1, 1, 1, 2, 1, 1, 3, 1, 3, 1, 3};
5         List<String> list = new ArrayList<>();
6         for (String word: words) {
7             boolean flag = true;
8             String temp = word.toLowerCase();
9             int first = alphaRow[temp.charAt(0) - 'a'];
10            for (int i = 1, j = temp.length() - 1; i <= j ; i++,j--) {
11                if(alphaRow[temp.charAt(i) - 'a'] != first ||
12                alphaRow[temp.charAt(j) - 'a'] != first){
13                    flag = false;
14                    break;
15                }
16            }
17            if(flag){
18                list.add(word);
19            }
20        }
21        String [] strings = new String[list.size()];
22        return list.toArray(strings);
23    }
24 }
```

520.DetectCapital

题目描述

我们定义，在以下情况时，单词的大写用法是正确的：

- 全部字母都是大写，比如 "USA" 。
- 单词中所有字母都不是大写，比如 "leetcode" 。
- 如果单词不只含有一个字母，只有首字母大写， 比如 "Google" 。
- 给你一个字符串 word 。如果大写用法正确，返回 true ；否则，返回 false 。

```
1 示例 1:
2
3 输入: word = "USA"
4 输出: true
5 示例 2:
6
7 输入: word = "FlaG"
8 输出: false
```

提示：

$1 \leq word.length \leq 100$
word 由小写和大写英文字母组成。

思路:

挨个判断

```
1 package cn.edu.csust.leetcode.Array;
2
3 class Solution {
4     public boolean detectCapitalUse(String word) {
5         boolean all_upper = Character.isUpperCase(word.charAt(0));
6         boolean all_lower = Character.isLowerCase(word.charAt(0));
7         boolean firstUpper = all_upper;
8         for(int i = 1; i < word.length(); i++){
9             char ch = word.charAt(i);
10            all_upper = all_upper && Character.isUpperCase(ch);
11            all_lower = all_lower && Character.isLowerCase(ch);
12            firstUpper = firstUpper && Character.isLowerCase(ch);
13        }
14        return all_upper || all_lower || firstUpper;
15    }
16 }
17 public class DetectCapital {
18     public static void main(String[] args) {
19         Solution solution = new Solution();
20         System.out.println(solution.detectCapitalUse("USA"));
21         System.out.println(solution.detectCapitalUse("leetcode"));
22         System.out.println(solution.detectCapitalUse("Flag"));
23         System.out.println(solution.detectCapitalUse("FlaG"));
24     }
25 }
```

检测大写字母

提交记录

550 / 550 个通过测试用例

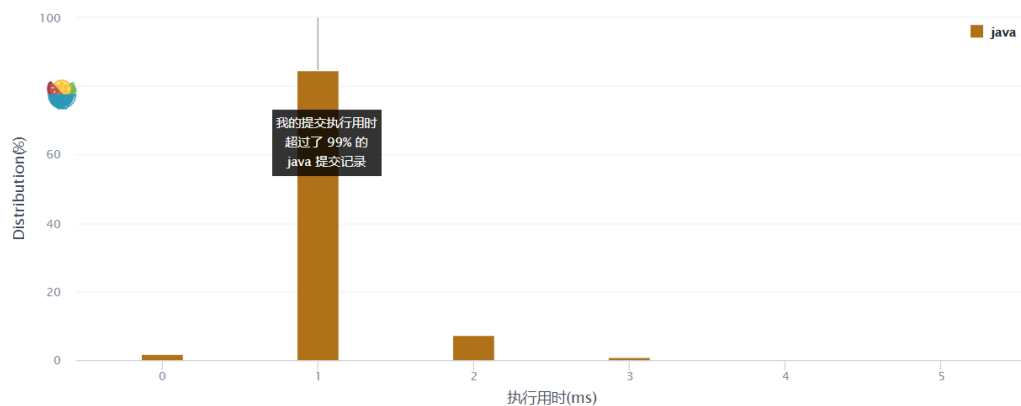
执行用时: 1 ms

内存消耗: 36.6 MB

状态: 通过

提交时间: 8 小时前

执行用时分布图表



575.糖果分发(DistributeCandies):easy

直接用集合去重，取Math.min(set.size(), candyType.length >> 1);

题目描述

给定一个偶数长度的数组，其中不同的数字代表着不同种类的糖果，每一个数字代表一个糖果。你需要把这些糖果平均分给一个弟弟和一个妹妹。返回妹妹可以获得的最大糖果的种类数。

```
1  示例 1：
2
3  输入：candies = [1,1,2,2,3,3]
4  输出：3
5  解析：一共有三种种类的糖果，每一种都有两个。
6      最优分配方案：妹妹获得[1,2,3]，弟弟也获得[1,2,3]。这样使妹妹获得糖果的种类数最多。
7  示例 2：
8
9  输入：candies = [1,1,2,3]
10 输出：2
11 解析：妹妹获得糖果[2,3]，弟弟获得糖果[1,1]，妹妹有两种不同的糖果，弟弟只有一种。这样使得妹妹可以获得的糖果种类数最多。
12 注意：
13
14 数组的长度为[2, 10,000]，并且确定为偶数。
15 数组中数字的大小在范围[-100,000, 100,000]内。
```

代码

```
1  class Solution{
2      public int distributeCandies(int[] candyType) {
3          Set<Integer> set = new HashSet<>();
4          int len = candyType.length;
5          for (int i = 0; i < len && set.size() < len / 2;) {
6              set.add(candyType[i++]);
7          }
8          return set.size();
9      }
10 }
11 public class DistributeCandidates {
12     public static void main(String[] args) {
13         Solution solution = new Solution();
14         System.out.println(solution.distributeCandies(new int[]{1,1,2,3}));
15     }
16 }
```

598. 范围求和 II

给定一个初始元素全部为 0，大小为 m*n 的矩阵M 以及在 M 上的一系列更新操作。

操作用二维数组表示，其中的每个操作用一个含有两个正整数 a 和 b 的数组表示，含义是将所有符合 $0 \leq i < a$ 以及 $0 \leq j < b$ 的元素 $M[i][j]$ 的值都增加 1。

在执行给定的一系列操作后，你需要返回矩阵中含有最大整数的元素个数。

```
1  示例 1：
```

```

2  输入：
3  m = 3, n = 3
4  operations = [[2,2],[3,3]]
5  输出：4
6  解释：
7  初始状态，M =
8  [[0, 0, 0],
9   [0, 0, 0],
10  [0, 0, 0]]
11
12 执行完操作 [2,2] 后，M =
13  [[1, 1, 0],
14   [1, 1, 0],
15   [0, 0, 0]]
16
17 执行完操作 [3,3] 后，M =
18  [[2, 2, 1],
19   [2, 2, 1],
20   [1, 1, 1]]
21
22 M 中最大的整数是 2，而且 M 中有4个值为2的元素。因此返回 4。

```

注意:

1. m 和 n 的范围是 [1,40000]。
2. a 的范围是 [1,m], b 的范围是 [1,n]。
3. 操作数目不超过 10000。

思路

求每个操作在行和列上的最小值，

如果操作数为0，那么最后的结果为m * n

代码

```

1  class Solution {
2      public int maxCount(int m, int n, int[][] ops) {
3          if(ops.length == 0)
4              return m * n;
5          int minRow=ops[0][0];
6          int minCol=ops[0][1];
7          for(int i=1;i<ops.length;i++){
8              if(ops[i][0]<minRow){
9                  minRow = ops[i][0];
10             }
11             if(ops[i][1]<minCol){
12                 minCol = ops[i][1];
13             }
14         }
15         return minRow * minCol;
16     }
17 }

```

629.KInversePairs

题目描述

629. K个逆序对数组

难度困难116收藏分享切换为英文接收动态反馈

给出两个整数 n 和 k ，找出所有包含从 1 到 n 的数字，且恰好拥有 k 个逆序对的不同的数组的个数。

逆序对的定义如下：对于数组的第 i 个和第 j 个元素，如果满 $i < j$ 且 $a[i] > a[j]$ ，则其为一个逆序对；否则不是。

由于答案可能很大，只需要返回 答案 $\text{mod } 10^9 + 7$ 的值。

示例 1:

```
1 输入：n = 3, k = 0
2 输出：1
3 解释：
4 只有数组 [1,2,3] 包含了从1到3的整数并且正好拥有 0 个逆序对。
```

示例 2:

```
1 输入：n = 3, k = 1
2 输出：2
3 解释：
4 数组 [1,3,2] 和 [2,1,3] 都有 1 个逆序对。
```

说明:

- n 的范围是 $[1, 1000]$ 并且 k 的范围是 $[0, 1000]$ 。

思路

动态规划递推公式

在推出递推公式的时候，这一步是关键

假如当前的4个数字的排列方式为：xxxx

再往其中添加一个数字5有如下几种添加方式：

- xxxx5
多出0个逆序对，因此有：
 $f1(5, k) = f(4, k)$
- xxx5x
多出1个逆序对，因此有：
 $f2(5, k+1) = f(4, k) \Rightarrow f2(5, k) = f(4, k-1)$
- xx5xx
多出1个逆序对，因此有：
 $f3(5, k+2) = f(4, k) \Rightarrow f3(5, k) = f(4, k-2)$
- x5xxx
多出1个逆序对，因此有：
 $f4(5, k+3) = f(4, k) \Rightarrow f4(5, k) = f(4, k-3)$
- 5xxxx
多出1个逆序对，因此有：

$$f(5, k+4) = f(4, k) \Rightarrow f(5, k) = f(4, k-4)$$

=>

$$f(5, k) = f_1 + f_2 + f_3 + \dots + f_5$$

=>

$$f(5, k) = f(4, k) + f(4, k-1) + f(4, k-2) + f(4, k-3) + f(4, k-4)$$

=>

$$f(n, k) = f(n-1, k) + f(n-1, k-1) + f(n-1, k-2) + f(n-1, k-3) + \dots + f(n-1, k-n+1)$$

=>

$$f(n, k+1) = f(n-1, k+1) + f(n-1, k) + f(n-1, k-1) + \dots + f(n-1, k-n+2)$$

=>

$$f(n, k+1) - f(n, k) = f(n-1, k+1) - f(n-1, k-n+1)$$

=>

$$f(n, k+1) = f(n, k) + f(n-1, k+1) - f(n-1, k-n+1)$$

=>

$$f(n, k) = f(n, k-1) + f(n-1, k) - f(n-1, k-n)$$

两个递推公式:

1. $f(n, k) = f(n-1, k) + f(n-1, k-1) + f(n-1, k-2) + f(n-1, k-3) + \dots + f(n-1, k-n+1)$
2. $f(n, k) = f(n, k-1) + f(n-1, k) - f(n-1, k-n)$

代码

```

1 package cn.edu.csust.leetcode.DynamicProgramming;
2 // dp(n, k+1) = dp(n - 1, k + 1) + dp(n, k) - dp(n - 1, k - n + 1)
3 class Solution {
4     public int kInversePairs(int n, int k) {
5         if(k > n * (n - 1) / 2){
6             return 0;
7         }
8         if(k == 0 || k == n * (n - 1) / 2){
9             return 1;
10        }
11        int mod = 1000000007;
12        long [][] dp = new long[n + 1][k + 1];
13        dp[2][0] = 1;
14        dp[2][1] = 1;
15        for (int i = 3; i <= n; i++) {
16            dp[i][0] = 1;
17            for (int j = 1; j < Math.min(k, n * (n - 1)/2); j++) {
18                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
19                if(j >= i){
20                    dp[i][j] -= dp[i][j - i];
21                }
22                dp[i][j] = (dp[i][j] + mod) % mod; //处理dp[i][j]为负数的情况
23            }
24        }
25        return (int)dp[n][k];
26    }
27 }
28 public class KInversePairs {
29     public static void main(String[] args) {
30         Solution solution = new Solution();
31         System.out.println(solution.kInversePairs(4, 0));
32     }

```

677. Map Sum Pairs

题目描述

实现一个 `MapSum` 类，支持两个方法，`insert` 和 `sum`：

`MapSum()` 初始化 `MapSum` 对象

`void insert(String key, int val)` 插入 `key-val` 键值对，字符串表示键 `key`，整数表示值 `val`。如果键 `key` 已经存在，那么原来的键值对将被替代成新的键值对。

`int sum(string prefix)` 返回所有以该前缀 `prefix` 开头的键 `key` 的值的总和。

示例：

```

1  输入：
2  ["MapSum", "insert", "sum", "insert", "sum"]
3  [[], ["apple", 3], ["ap"], ["app", 2], ["ap"]]
4  输出：
5  [null, null, 3, null, 5]
6
7  解释：
8  MapSum mapSum = new MapSum();
9  mapSum.insert("apple", 3);
10 mapSum.sum("ap");           // return 3 (apple = 3)
11 mapSum.insert("app", 2);
12 mapSum.sum("ap");           // return 5 (apple + app = 3 + 2 = 5)
13
14
```

提示：

- $1 \leq \text{key.length}, \text{prefix.length} \leq 50$
- `key` 和 `prefix` 仅由小写英文字母组成
- $1 \leq \text{val} \leq 1000$
- 最多调用 50 次 `insert` 和 `sum`

思路

1. 看到 Key value 就会情不自禁想到 Hashmap，就跟看到查找可能第一反应是 for 循环而不是二分查找一样

2. 本题的正确打开方式是：新定义一个类，作为树节点，每个节点有 26 个子分支，当然不是所有的分支都会用的上，这里可能会有空间的浪费但是其实浪费不了多少，因为每个子分支只是一个指针类型。最后根据最佳前缀树进行深度或者广度遍历求和，深度遍历时间更佳，广度遍历 just-so-so

代码

```

1  package cn.edu.csust.leetcode;
2
3  import sun.text.normalizer.Trie;
4
5  import java.util.*;
6
7  /**
8   * Your MapSum object will be instantiated and called as such:

```

```

9  * MapSum obj = new MapSum();
10 * obj.insert(key,val);
11 * int param_2 = obj.sum(prefix);
12 * 时间效率很低
13 */
14 class MapSum1 {
15     Map<String, Integer> map = new HashMap<>();
16     public MapSum() {
17
18     }
19
20     public void insert(String key, int val) {
21         map.put(key, val);
22     }
23
24     public int sum(String prefix) {
25         Set<String> keySet = map.keySet();
26         int total = 0;
27         for (String key: keySet) {
28             if(key.startsWith(prefix)){
29                 total += map.get(key);
30             }
31         }
32         return total;
33     }
34 }
35
36 // 广度优先前缀树
37 class MapSum2{
38     int ret;
39     TrieNode root;
40     class TrieNode{
41         int val = 0;
42         TrieNode[] next = new TrieNode[26];
43     }
44     public MapSum(){
45         ret = 0;
46         root = new TrieNode();
47     }
48
49     public void insert(String key, Integer val){
50         TrieNode cur = root;
51         for (int i = 0; i < key.length(); i++) {
52             int k = key.charAt(i) - 'a';
53             if(cur.next[k] == null){
54                 cur.next[k] = new TrieNode();
55             }
56             cur = cur.next[k];
57         }
58         cur.val = val;
59     }
60
61     public int sum(String prefix) {
62         TrieNode cur = root;
63         ret = 0;
64         for(int i =0 ;i < prefix.length();i++){
65             int u = prefix.charAt(i) - 'a';
66             if(cur.next[u] == null){

```

```

67         return 0;
68     }
69     cur = cur.next[u];
70 }
71 Queue<TrieNode> q = new LinkedList<>();
72 q.add(cur);
73 while(!q.isEmpty()) {
74     TrieNode node = q.poll();
75     ret += node.val;
76     for(int i = 0; i < 26; i++) {
77         if(node.next[i] != null) q.add(node.next[i]);
78     }
79 }
80 return ret;
81 }
82 }
83 //深度优先
84 class MapSum{
85     int ret;
86     TrieNode root;
87     class TrieNode{
88         int val = 0;
89         TrieNode[] next = new TrieNode[26];
90     }
91     public MapSum(){
92         ret = 0;
93         root = new TrieNode();
94     }
95
96     public void insert(String key, Integer val){
97         TrieNode cur = root;
98         for (int i = 0; i < key.length(); i++) {
99             int k = key.charAt(i) - 'a';
100             if(cur.next[k] == null){
101                 cur.next[k] = new TrieNode();
102             }
103             cur = cur.next[k];
104         }
105         cur.val = val;
106     }
107
108     public int sum(String prefix) {
109         TrieNode cur = root;
110         for(int i = 0; i < prefix.length(); i++){
111             int u = prefix.charAt(i) - 'a';
112             if(cur.next[u] == null){
113                 return 0;
114             }
115             cur = cur.next[u];
116         }
117
118         return dfs(cur);
119     }
120     private int dfs(TrieNode node){
121         if(node == null)
122             return 0;
123         int sum = 0;
124         for(int i = 0; i < 26; ++i){

```



```

125         sum += dfs(node.next[i]);
126     }
127     return node.val + sum;
128 }
129 }
130 public class MapSumPairs {
131     // public static void main(String[] args) {
132     //     MapSum mapSum = new MapSum();
133     //     mapSum.insert("apple", 3);
134     //     System.out.println(mapSum.sum("ap"));
135     //     mapSum.insert("app", 2);
136     //     System.out.println(mapSum.sum("ap"));
137     // }
138     public static void main(String[] args) {
139         MapSum mapSum = new MapSum();
140         mapSum.insert("a", 3);
141         System.out.println(mapSum.sum("ap"));
142         mapSum.insert("b", 2);
143         System.out.println(mapSum.sum("a"));
144         System.out.println(mapSum.root.next[0].val);
145     }
146 }
147

```

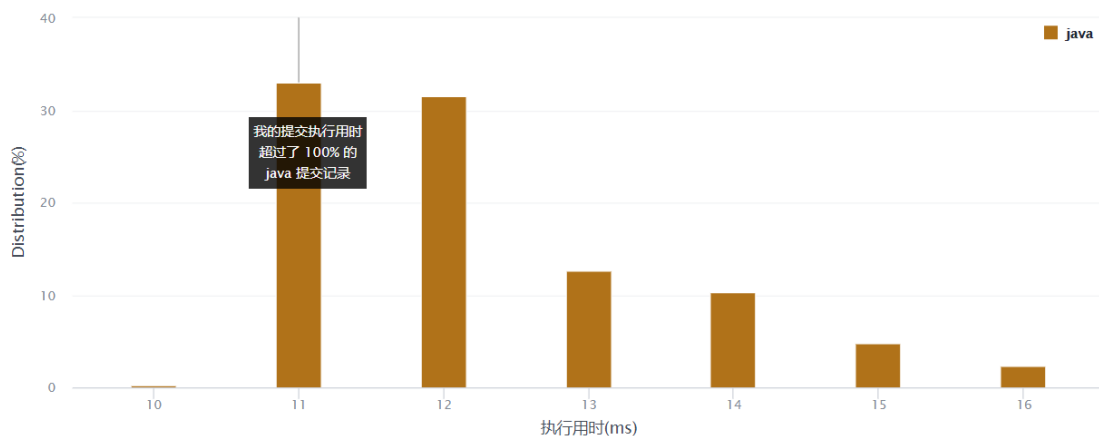
键值映射

提交记录

35 / 35 个通过测试用例
 执行用时: 11 ms
 内存消耗: 38.3 MB

状态: **通过**
 提交时间: 2 分钟前

执行用时分布图表



869.重新排序得到2的幂:

题目描述

给定正整数 N ，我们按任何顺序（包括原始顺序）将数字重新排序，注意其前导数字不能为零。

如果我们可以通过上述方式得到 2 的幂，返回 true；否则，返回 false。

1 示例 1:
 2

```
3  输入: 1
4  输出: true
5  示例 2:
6
7  输入: 10
8  输出: false
9  示例 3:
10
11 输入: 16
12 输出: true
13 示例 4:
14
15 输入: 24
16 输出: false
17 示例 5:
18
19 输入: 46
20 输出: true
```

提示: $1 \leq N \leq 10^9$

这种题无非就是比较数字串中各个单数字出现的频率是否相同

只要明白这一点就可以在做题了

思路

- 首先根据输入的数字大概判断在哪个区间，比如10以内，就对应2的[0, 1, 2, 3]次幂，以此类推
- 再根据得到的区间挨个比较数字字符串数字频率是否相同，采用HashMap结构返回比较结果

代码

```
1  /**
2   思路一：预先存结果
3   */
4  class Solution{
5      public boolean reorderedPowerOf2(int N) {
6          String[] rec =
7          {"1","2","4","8","16","23","46","128","256","125","0124","0248","0469","1289",
8           "13468",
9           "23678","35566","011237","122446","224588","0145678","0122579","0134449","0
10          368888",
11           "11266777","23334455","01466788","112234778","234455668","012356789","01123
12          44778"};
13          char[] at = String.valueOf(N).toCharArray();
14          Arrays.sort(at);
15          String str = new String(at);
16          for(String p:rec){
17              if(str.equals(p)) return true;
18          }
19          return false;
20      }
21  }
22  /**
23   思路二：正常思路
```

```

20 */
21 class Solution2 {
22     public boolean reorderedPowerOf2(int n) {
23         List<Map<Integer, Integer>> power2MapList = new ArrayList<>();
24         Map<Integer, Integer> map = getNumberMap(n);
25         int [] interval = getInterval(n);
26         for (int num = interval[0]; num <= interval[interval.length -
1]; num++) {
27             Map<Integer, Integer> map1 = getNumberMap((1 << num));
28             if(map1.equals(map)) return true;
29
30         }
31         return false;
32     }
33     private Map<Integer, Integer> getNumberMap(int n){
34         Map<Integer, Integer> map = new HashMap<>();
35         while(n != 0){
36             int num = n % 10;
37             n /= 10;
38             map.put(num, map.getOrDefault(num, 0) + 1);
39         }
40         return map;
41     }
42     private int [] getInterval(int n){
43         if(n < 10){
44             return new int[]{0,3};
45         }
46         else if(n < 100){
47             return new int[]{4, 6};
48         }
49         else if(n < Math.pow(10, 3)){
50             return new int[]{7, 9};
51         }
52         else if(n < Math.pow(10, 4)){
53             return new int[]{10, 13};
54         }
55         else if(n < Math.pow(10, 5)){
56             return new int[]{14, 16};
57         }
58         else if(n < Math.pow(10, 6)){
59             return new int[]{17, 19};
60         }
61         else if(n < Math.pow(10, 7)){
62             return new int[]{20,23};
63         }
64         else if(n < Math.pow(10, 8)){
65             return new int[]{24, 26};
66         }
67         return new int[]{27, 29};
68     }
69 }
70 }
71
72 //时空Beats双90%，思路简单
73 class Solution3{
74     public boolean reorderedPowerOf2(int N){
75         int [] mapN = new int[10];
76

```

```

77         int count = getNumberMap(N, mapN);
78         for(int i = 29;i>=0;i--){
79             int t = (1<<i);
80             int [] mapPow = new int[10];
81             int countT = getNumberMap(t, mapPow);
82             if(countT == count && Arrays.equals(mapPow, mapN)){
83                 return true;
84             }
85         }
86         return false;
87     }
88     public int getNumberMap(int num, int [] count){
89
90         int n = 0;
91         while(num != 0){
92             count[num % 10]++;
93             num /= 10;
94             n++;
95         }
96         return n;
97     }
98 }

```

1218.最长定差子序列:mid

题目描述

给你一个整数数组 `arr` 和一个整数 `difference`，请你找出并返回 `arr` 中最长等差子序列的长度，该子序列中相邻元素之间的差等于 `difference`。

子序列 是指在改变其余元素顺序的情况下，通过删除一些元素或不删除任何元素而从 `arr` 派生出来的序列

输入示例：

```

1  示例 1:
2
3  输入: arr = [1,2,3,4], difference = 1
4  输出: 4
5  解释: 最长的等差子序列是 [1,2,3,4]。
6  示例 2:
7
8  输入: arr = [1,3,5,7], difference = 1
9  输出: 1
10 解释: 最长的等差子序列是任意单个元素。
11 示例 3:
12
13 输入: arr = [1,5,7,8,5,3,4,2,1], difference = -2
14 输出: 4
15 解释: 最长的等差子序列f 是 [7,5,3,1]。

```

提示：

- $1 \leq arr.length \leq 10^5$
- $-10^4 \leq arr[i], difference \leq 10^4$

思路

哈希表记录各个不同的序列的长度，最后取最长的一条

代码

```
1 package cn.edu.csust.leetcode;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 class Solution {
7
8     public int longestSubsequence(int[] arr, int difference) {
9         int res = 1;
10        Map<Integer, Integer> map = new HashMap<>();
11        for (int i = arr.length - 1; i >= 0; i--) {
12            int key = arr[i] + difference;
13            Integer val = map.get(key);
14            if(val != null){
15                int temp = val + 1;
16                res = Math.max(res, temp);
17                map.put(arr[i], temp);
18                //System.out.println(key + " exists! Now remove " + key + "
19                put " + arr[i]+ " and the value of " + arr[i] + " is: " + temp);
19            }
20            else{
21                map.put(arr[i], 1);
22                //System.out.println(key + " does not exist! put " +
23                arr[i]);
23            }
24        }
25        return res;
26    }
27    public int longestSubsequence2(int[] arr, int difference) {
28        int res = 1;
29        Map<Integer, Integer> map = new HashMap<>();
30        for (int i = 0; i < arr.length; i++) {
31            int key = arr[i] - difference;
32            Integer val = map.get(key);
33            if(val != null){
34                int temp = val + 1;
35                res = Math.max(res, temp);
36                map.put(arr[i], temp);
37            }
38            else{
39                map.put(arr[i], 1);
40            }
41        }
42        return res;
43    }
44 }
45 public class LongestArithmeticSubsequenceOfGivenDifference {
46     public static void main(String[] args) {
47         Solution solution = new Solution();
48         System.out.println(solution.longestSubsequence(new int []{1,2,3,
49 4,5, 7}, 1));
```

```
49         System.out.println(solution.longestSubsequence(new int []
    {1,5,7,8,5,3,4,2,1}, -2));
50     }
51 }
52
```

最长定差子序列

提交记录

39 / 39 个通过测试用例

执行用时: 34 ms

内存消耗: 55.5 MB

状态: 通过

提交时间: 17 分钟前

执行用时分布图表

