981. 基于时间的键值存储（中等）

1. 题目描述

   创建一个基于时间的键值存储类 TimeMap，它支持下面两个操作：

   1. set(string key, string value, int timestamp)
      - 存储键 key、值 value，以及给定的时间戳 timestamp。
   2. get(string key, int timestamp)
      - 返回先前调用 set(key, value, timestamp_prev) 所存储的值，其中 timestamp_prev <= timestamp。
      - 如果有多个这样的值，则返回对应最大的 timestamp_prev 的那个值。
      - 如果没有值，则返回空字符串（""）。

```
示例 1:
输入: inputs = ["TimeMap","set","get","get","set","get","get"], inputs = [[],
["foo","bar",1],["foo",1],["foo",3],["foo","bar2",4],["foo",4],["foo",5]]
输出: [null,null,"bar","bar",null,"bar2","bar2"]
解释:
TimeMap kv;
kv.set("foo", "bar", 1); // 存储键 "foo" 和值 "bar" 以及时间戳 timestamp = 1
kv.get("foo", 1);  // 输出 "bar"
kv.get("foo", 3); // 输出 "bar" 因为在时间戳 3 和时间戳 2 处没有对应 "foo" 的值，所以唯一的
值位于时间戳 1 处（即 "bar"）
kv.set("foo", "bar2", 4);
kv.get("foo", 4); // 输出 "bar2"
kv.get("foo", 5); // 输出 "bar2"

示例 2:
输入: inputs = ["TimeMap","set","set","get","get","get","get","get"], inputs = [[],
["love","high",10],["love","low",20],["love",5],["love",10],["love",15],
["love",20],["love",25]]
输出: [null,null,null,"","high","high","low","low"]
```

提示：

- 所有的键/值字符串都是小写的。
- 所有的键/值字符串长度都在 [1, 100] 范围内。
- 所有 TimeMap.set 操作中的时间戳 timestamps 都是严格递增的。
- 1 <= timestamp <= 10^7
- TimeMap.set 和 TimeMap.get 函数在每个测试用例中将（组合）调用总计 120000 次。

2. 简单实现

```
class TimeMap {
public:
    unordered_map<string, vector<pair<int, string>>> data;
    struct cmp{
```

```
        bool operator  () (const pair<int, string>& a, const pair<int, string>& b){
            return a.first < b.first;
        }
    };
    /** Initialize your data structure here. */
    TimeMap() {
    }
    void set(string key, string value, int timestamp) {
        data[key].push_back(make_pair(timestamp, value));
    }
    string get(string key, int timestamp) {
        auto it = upper_bound(data[key].begin(), data[key].end(),
make_pair(timestamp,""), cmp());
        if(it == data[key].begin())
            return "";
        it--;
        return it->second;
    }
};
```

135. 分发糖果（困难）

1. 题目描述

老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。

你需要按照以下要求，帮助老师给这些孩子分发糖果：

- 每个孩子至少分配到 1 个糖果。
- 相邻的孩子中，评分高的孩子必须获得更多的糖果。

那么这样下来，老师至少需要准备多少颗糖果呢?

```
示例 1:
输入: [1,0,2]
输出: 5
解释: 你可以分别给这三个孩子分发 2、1、2 颗糖果。

示例 2:
输入: [1,2,2]
输出: 4
解释: 你可以分别给这三个孩子分发 1、2、1 颗糖果。第三个孩子只得到 1 颗糖果，这已满足上述两个条件。
```

2. 简单实现

位置i的孩子的糖果数只受其身边评分比他低的孩子的糖果数的约束，因此可以从评分最低的孩子发起

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        int n = ratings.size();
        if(n == 1) return 1;
        multimap<int, int> idxs;//<评分，索引>
```

```
        int ans = 0;
        vector<int> num(n);//各个位置小孩的糖果数
        for(int i = 0; i < ratings.size(); i++){
            if(ratings[i] == 0){
                ans++;
                num[i] = 1;
            }
            else
                idxs.insert(make_pair(ratings[i], i));
        }
        for(auto it = idxs.begin(); it != idxs.end(); it++){//从分低的开始
            int cur = 1;
            int idx = it->second;
            if(idx > 0 && ratings[idx-1] < ratings[idx])//左边的分少
                cur = max(cur, num[idx-1]+1);
            if(idx < n-1 && ratings[idx+1] < ratings[idx])//右边的分少
                cur = max(cur, num[idx+1]+1);
            num[idx] = cur;
            ans += cur;
        }
        return ans;
    }
};
```

3. 使用一个数组遍历

方法 2：用两个数组

算法

在这种方法中，我们使用两个一维数组 $left2right$ 和 $right2left$ 。数组 $left2right$ 用来存储每名学生只与左边邻居有关的所需糖果数。也就是假设规则为：如果一名学生评分比他左边学生高，那么他应该比他左边学生得到更多糖果。类似的，$right2left$ 数组用来保存只与右边邻居有关的所需糖果数。也就是假设规则为：如果一名学生评分比他右边学生高，那么他应该比他右边学生得到更多糖果。

首先，我们在 $left2rigth$ 和 $right2left$ 中，给每个学生 1 个糖果。然后，我们从左向右遍历整个数组，只要当前学生评分比他左邻居高，我们在 $left2right$ 数组中更新当前学生的糖果数 $left2right[i] = left2right[i-1] + 1$，这是因为在每次更新前，当前学生的糖果数一定小于等于他左邻居的糖果数。

在从左到右扫描后，我们用同样的方法从右到左只要当前学生的评分比他右边（第 $(i+1)$ 个）学生高，就更新 $right2left[i]$ 为 $right2left[i] = right2left[i+1] + 1$ 。

现在，对于数组中第 i 个学生，为了满足题中条件，我们需要给他 $\max(left2right[i], right2left[i])$ 个糖果。因此，最后我们得到最少糖果数：

$$minimum\_candies = \sum_{i=0}^{n-1} \max(left2right[i], right2left[i])$$

方法 3：使用一个数组

算法

在前面的方法中，我们使用了两个数组分别记录每一个学生与他左邻居和右邻居的关系，后来再将两个数组合并。在这里我们可以只用一个数组 $candies$ ，记录当前学生被分配的糖果数。

首先我们给每个学生 1 个糖果，然后我们从左到右遍历并分配糖果，我们仅更新评分比左邻居高且糖果数小于等于左邻居的学生，将其更新为 $candies[i] = candies[i-1] + 1$。在更新的过程中，我们不需要比较 $candies[i]$ 和 $candies[i-1]$ ，因为在更新前一定有 $candies[i] \leq candies[i-1]$ 。

从左到右遍历完后，我们同样地从右到左遍历。现在我们需要更新每个学生 i 同时满足左邻居和右邻居的关系。在这次遍历汇总，如果 $ratings[i] > ratings[i+1]$ ，仅考虑右邻居规则的情况下，我们本应该更新为 $candies[i] = candies[i+1] + 1$ ，但是这次我们仅当 $candies[i] \leq candies[i+1]$ 才更新。这是因为我们在从左到右遍历的时候已经修改过 $candies$ 数组，所以 $candies[i]$ 不一定小于等于 $candies[i+1]$。所以，如果 $ratings[i] > ratings[i+1]$ ，我们更新为 $candies[i] = \max(candies[i], candies[i+1] + 1])$ ，这样 $candies[i]$ 同时满足左邻居和右邻居的约束。

再一次，我们把 $candies$ 数组中的所有元素求和，获得所需结果。

$$minimum_candies = \sum_{i=0}^{n-1} candies[i]$$

其中， n 是评分数组的长度。

```java
public class Solution {
    public int candy(int[] ratings) {
        int[] candies = new int[ratings.length];
        Arrays.fill(candies, 1);
        for (int i = 1; i < ratings.length; i++) {
            if (ratings[i] > ratings[i - 1]) {
                candies[i] = candies[i - 1] + 1;
            }
        }
        int sum = candies[ratings.length - 1];
        for (int i = ratings.length - 2; i >= 0; i--) {
            if (ratings[i] > ratings[i + 1]) {
                candies[i] = Math.max(candies[i], candies[i + 1] + 1);
            }
            sum += candies[i];
        }
        return sum;
    }
}
```

4. 常数空间O(n)

方法 4：常数空间一次遍历

算法

这个方法通过观察（如下面的图所展示）发现，为了获得最少总数的糖果，糖果的分配每次都是增加 1 的。进一步的，在分配糖果时，给一个学生的最少数目是 1 。所以，局部的分配形式一定是 $1, 2, 3, ..., \mathrm{n}$ 或者 $\mathrm{n}, ..., 2, 1$，总和是 $\frac{n(n+1)}{2}$ 。

现在我们可以把评分数组 $ratings$ 当做一些上升和下降的坡。每当坡是上升的，糖果的分配一定是 $1, 2, 3, ..., \mathrm{m}$ 这样的。同样的，如果是一个下降的坡，一定是 $\mathrm{k}, ..., 2, 1$ 的形式。一个随之而来的情况是，每个峰都只会在这些坡中的一个出现。那么我们应该把这个峰放在上升的坡中还是下降的坡中呢？

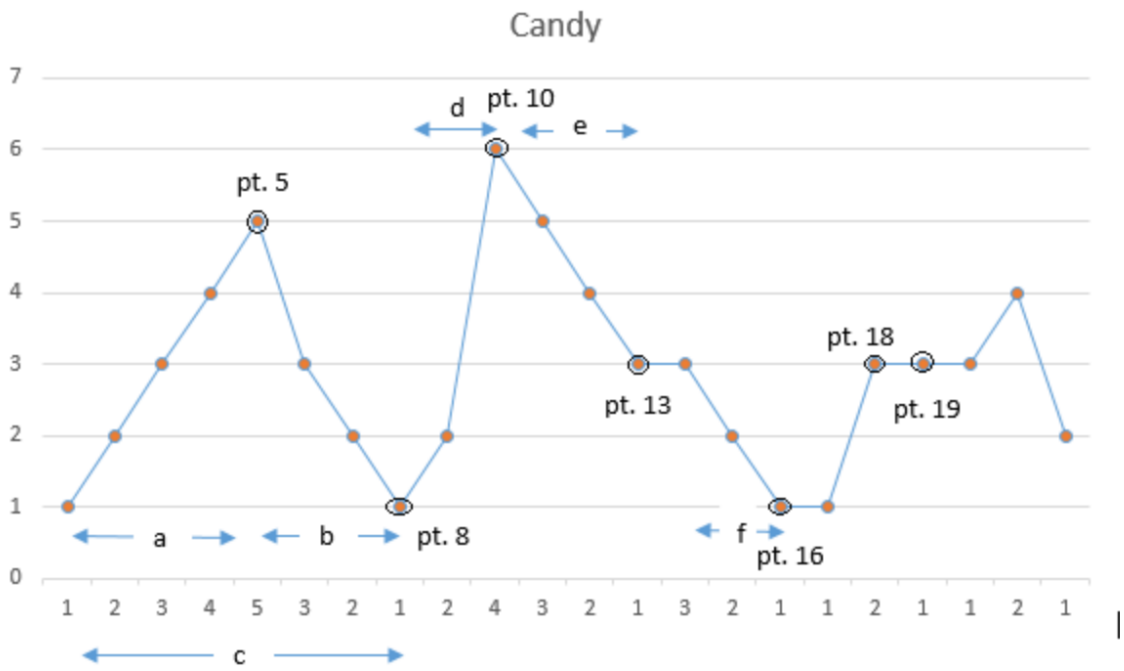
为了解决这个问题，我们观察到为了同时满足左右邻居的约束，峰值一定是上升的坡和下降的坡中所有点的最大值，所以为了决定需要的糖果数，峰点需要算在上升坡和下降坡较多点的那一边。局部谷点也只能被包括在一个坡中，但是这种情况很容易解决，因为局部谷点总是只会被分配 1 个糖果（可以在下一个坡开始计数时减去）。

接下来考虑实现，我们维护两个变量 old_slope 和 new_slope 来决定现在是在峰还是在谷，同时我们用 up 和 $down$ 两个变量分别记录上升或者下降坡中的学生个数（不包括峰点）。我们总是在一个下降的坡接着上升坡（或者上升坡接一个下降坡）的时候更新 $candies$ 的总数。

在一个山的结束处，我们决定将峰点算在上升坡还是下降坡中，决定的依据是比较 up 和 $down$ 两个变量。因此，峰值的数目应该为 $\max(up, down) + 1$。此时，我们将 up 和 $down$ 变量重新初始化，表示一个新的山的开始。

下面的图展示了如下样例的结果。

```
rankings: [1 2 3 4 5 3 2 1 2 6 5 4 3 3 2 1 1 3 3 3 4 2]
```



从这个图中，我们可以看到糖果在局部的分配中一定是 $1, 2, ..., n$ 或者 $n, ..., 2, 1$ 这样的形式。对于由 a 和 b 组成的第一座山，在分配峰点（$pt.5$）糖果的时候，它应该被分配到 a 中满足左邻居约束，b 中的局部谷点（$pt.8$）标志着第一座山（c）的结束。在计算的时候，我们可以把这个点归属为当前的山，也可以归属到接下来的山中。点 $pt.13$ 标记的是第二座山的结束，因为 $pt.13$ 和 $pt.14$ 两个学生的评分是相同的。因此，区域 e 比区域 d 有更多的点，局部峰点 $(pt.10)$ 应该被划分到 e 区域满足右邻居的约束。现在第三座山 f 应该被考虑为一座只有下降坡没有上升坡的山 $(up = 0)$ 。类似的，因为与旁边的学生评分相同， $pt.16, 18, 19$ 也是山的结束。

```
public class Solution {
    public int count(int n) {
        return (n * (n + 1)) / 2;
    }
    public int candy(int[] ratings) {
        if (ratings.length <= 1)
            return ratings.length;
        int candies = 0;
        int up = 0;
        int down = 0;
        int old_slope = 0;
        for (int i = 1; i < ratings.length; i++) {
            int new_slope = (ratings[i] > ratings[i - 1]) ? 1 : (ratings[i] <
ratings[i - 1] ? -1 : 0);
            if ((old_slope > 0 && new_slope == 0) || (old_slope < 0 && new_slope >=
0)) {//峰或谷
                candies += count(up) + count(down) + Math.max(up, down);
                up = 0;
```

```
                down = 0;
            }
            if (new_slope > 0)
                up++;
            if (new_slope < 0)
                down++;
            if (new_slope == 0)
                candies++;

            old_slope = new_slope;
        }
        candies += count(up) + count(down) + Math.max(up, down) + 1;
        return candies;
    }
}
```

13. 罗马数字转整数（简单）

1. 题目描述

罗马数字包含以下七种字符: I， V， X， L，C，D 和 M。

```
字符           数值
I              1
V              5
X              10
L              50
C              100
D              500
M              1000
```

例如， 罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X + II 。 27 写做 XXVII, 即为 XX + V + II 。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

- I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
- X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
- C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内。

```
示例 1:
输入: "III"
输出: 3

示例 2:
输入: "IV"
输出: 4

示例 3:
```

```
输入: "IX"
输出: 9

示例 4:
输入: "LVIII"
输出: 58
解释: L = 50, V= 5, III = 3.

示例 5:
输入: "MCMXCIV"
输出: 1994
解释: M = 1000, CM = 900, XC = 90, IV = 4.
```

2. 简单实现

```
class Solution {
public:
    int romanToInt(string s) {
        int len = s.size();
        int num = 0;
        for(int i = 0; i < len; i++){
            switch(s[i]){
                case 'I':
                    if(s[i+1] == 'V'){
                        num += 4;
                        i++;
                    }
                    else if(s[i+1] == 'X'){
                        num += 9;
                        i++;
                    }
                    else
                        num += 1;
                    break;
                case 'V':
                    num += 5;
                    break;
                case 'X':
                     if(s[i+1] == 'L'){
                        num += 40;
                        i++;
                    }
                    else if(s[i+1] == 'C'){
                        num += 90;
                        i++;
                    }
                    else
                        num += 10;
                    break;
                case 'L':
                    num += 50;
                    break;
                case 'C':
```

```
                    if(s[i+1] == 'D'){
                        num += 400;
                        i++;
                    }
                    else if(s[i+1] == 'M'){
                        num += 900;
                        i++;
                    }
                    else
                        num += 100;
                    break;
                case 'D':
                    num += 500;
                    break;
                case 'M':
                    num += 1000;
                    break;
            }
        }
        return num;
    }
};
```

212. 单词搜索II（困难）

1. 题目描述

   给定一个二维网格 board 和一个字典中的单词列表 words，找出所有同时在二维网格和字典中出现的单词。

   单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。

   ```
   示例:
   输入:
   words = ["oath","pea","eat","rain"] and board =
   [
     ['o','a','a','n'],
     ['e','t','a','e'],
     ['i','h','k','r'],
     ['i','f','l','v']
   ]
   输出: ["eat","oath"]
   ```

   说明: 你可以假设所有输入都由小写字母 a-z 组成。

   提示:

   - 你需要优化回溯算法以通过更大数据量的测试。你能否早点停止回溯?
   - 如果当前单词不存在于所有单词的前缀中，则可以立即停止回溯。什么样的数据结构可以有效地执行这样的操作? 散列表是否可行? 为什么? 前缀树如何? 如果你想学习如何实现一个基本的前缀树，请先查看这个问题： 实现Trie（前缀树）。

2. 简单实现

前缀树+DFS

```
class Trie {
public:
    unordered_map<char, Trie*> children;
    bool isWord;
    Trie() {
        isWord = false;
    }
    void insert(string word) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->children.count(word[i]) <= 0)
                cur->children[word[i]] = new Trie();
            cur = cur->children[word[i]];
        }
        cur->isWord = true;
    }
};
class Solution {
public:
    vector<string> ans;
    int m,n;
    vector<vector<int>> dirs = {{0,1}, {0,-1}, {1,0}, {-1,0}};
    void dfs(vector<vector<char>>& board, int x, int y, Trie* cur,
vector<vector<bool>>& visited, string temp){
        if(cur->isWord){
            ans.push_back(temp);
            cur->isWord = false;//表示该单词已经找到过了，不用再找
        }
        for(int i = 0; i < 4; i++){
            int xx = x + dirs[i][0];
            int yy = y + dirs[i][1];
            if(xx >= 0 && xx < m && yy >= 0 && yy < n
               && cur->children.find(board[xx][yy]) != cur->children.end()
               && !visited[xx][yy]){
                visited[xx][yy] = true;
                dfs(board, xx, yy, cur->children[board[xx][yy]], visited,
temp+board[xx][yy]);
                visited[xx][yy] = false;
            }
        }
    }
    vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
        m = board.size();
        if(m == 0) return {};
        n = board[0].size();
        Trie* dic = new Trie();
        for(int i = 0; i < words.size(); i++)//构造字典树
            dic->insert(words[i]);
        vector<vector<bool>> visited = vector<vector<bool>>(m, vector<bool>(n,
false));
        for(int i = 0; i < m; i++){
```

```
            for(int j = 0; j < n; j++){
                if(dic->children.find(board[i][j]) != dic->children.end() &&
!visited[i][j]){
                    visited[i][j] = true;
                    string temp = "";
                    temp += board[i][j];
                    dfs(board, i, j, dic->children[board[i][j]], visited, temp);
                    visited[i][j] = false;
                }
            }
        }
        return ans;
    }
};
```

还可以改进，把isword改成在words里的索引，就不用一直维护temp了，省时间空间

362. 敲击计数器（中等）

要会员

1055. 形成字符串的最短路径（中等）

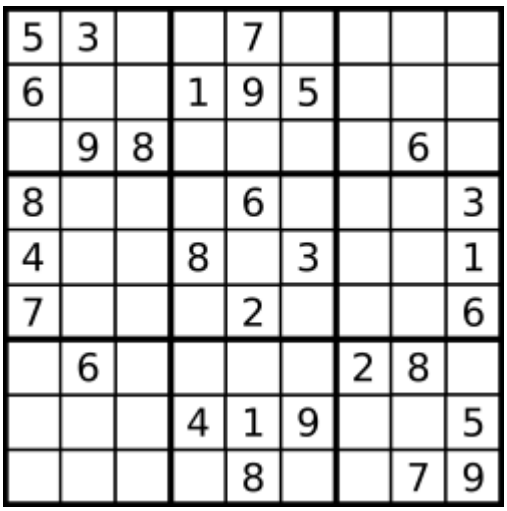
要会员

36. 有效的数独（中等）

1. 题目描述

   判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。

   - 数字 1-9 在每一行只能出现一次。
   - 数字 1-9 在每一列只能出现一次。
   - 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。



上图是一个部分填充的有效的数独。

数独部分空格内已填入了数字，空白格用 '.' 表示。

```
示例 1:
输入:
[
  ["5","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".",".","2","8","."],
  [".",".",".",".","4","1","9",".",".","5"],
  [".",".",".",".",".","8",".",".","7","9"]
]
输出: true

示例 2:
输入:
[
  ["8","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".",".","2","8","."],
  [".",".",".",".","4","1","9",".",".","5"],
  [".",".",".",".",".","8",".",".","7","9"]
]
输出: false
解释: 除了第一行的第一个数字从 5 改为 8 以外，空格内其他数字均与 示例1 相同。
     但由于位于左上角的 3x3 宫内有两个 8 存在，因此这个数独是无效的。
```

说明:

- 一个有效的数独（部分已被填充）不一定是可解的。
- 只需要根据以上规则，验证已经填入的数字是否有效即可。
- 给定数独序列只包含数字 1-9 和字符 '.' 。
- 给定数独永远是 9x9 形式的。

2. 简单实现

```
class Solution {
public:
    bool isValidSudoku(vector<vector<char>>& board) {
        int R,C;
        if(!(R=board.size()) || !(C=board[0].size())) return false;
        int rows[10][10] = {0};
        int cols[10][10] = {0};
        int cells[3][3][10] = {0};

        for(int i=0;i<R;++i){
```

```
            for(int j=0;j<C;++j){
                if(board[i][j] == '.') continue;
                int ch= board[i][j]-'0';
                if(++rows[i][ch] > 1 || ++cols[j][ch] > 1 || ++cells[i/3][j/3][ch]
> 1) return false;
            }
        }
        return true;
    }

};
```

157. 用Read4读取N个字符（简单）

要会员

296. 最佳的碰头地点（困难）

要会员

734. 句子相似性（简单）

要会员

777. 在LR字符串中交换相邻字符（中等）

1. 题目描述

   在一个由 'L' , 'R' 和 'X' 三个字符组成的字符串（例如"RXXLRXRXL"）中进行移动操作。一次移动操作指用一个"LX"替换一个"XL"，或者用一个"XR"替换一个"RX"。现给定起始字符串start和结束字符串end，请编写代码，当且仅当存在一系列移动操作使得start可以转换成end时， 返回True。

   ```
   示例 :
   输入: start = "RXXLRXRXL", end = "XRLXXRRLX"
   输出: True
   解释:
   我们可以通过以下几步将start转换成end:
   RXXLRXRXL ->
   XRXLRXRXL ->
   XRLXRXRXL ->
   XRLXXRRXL ->
   XRLXXRRLX
   ```

   提示:

   - `1 <= len(start) = len(end) <= 10000`。
   - `start` 和 `end` 中的字符串仅限于 `'L'`, `'R'` 和 `'X'`。
2. 正确解法

写了半天DFS，实际上是一道找规律的题

思路

将 `'L'`，`'R'` 分别理解为一个队伍中面向左和面向右的人，`'X'` 理解为队伍中的空挡。可以问自己一个问题，一次移动操作之后有什么是保持不变的？ 这是状态转换问题中一个很常见的思路。

算法

这道题的 *转换不变性* 在于字符串中的 `'L'` 和 `'R'` 是不会互相穿插的，也就是队伍中的人在移动过程中是不能穿过人的。这意味着开始和结束的字符串如果只看 `'L'` 和 `'R'` 的话是一模一样的。

除此之外，第 `n` 个 `'L'` 不可能移动到初始位置的右边，第 `n` 个 `'R'` 不可能移动到初始位置的左边，我们把这个特性称为 "*可到达性*"。设 $(i_1, i_2, \cdots)$，$(i_1^{'}, i_2^{'}, \cdots)$ 为每个字符 `'L'` 在原始字符串和目标字符串的位置，$(j_1, j_2, \cdots)$，$(j_1^{'}, j_2^{'}, \cdots)$ 为每个字符 `'R'` 在原始字符串和目标字符串的位置，如果 $i_k \geq i_k^{'}$ 和 $j_k \leq j_k^{'}$ 都能满足，这个字符串就是 "*可到达的*"。

根据 *转换不变性* 和 *可到达性*，在算法中可以分别检查这两个性质是否满足。如果都满足，则返回 `True`，否则返回 `False`。

可以用双指针来解决这个问题，对于 `i`，`j` 两个指针，分别让他们指向 `start` 和 `end`，且保证 `start[i] != 'X'`，`end[j] != 'X'`。接下来开始移动指针，如果 `start[i] != end[j]`，则不满足 *转换不变性*，如果 `start[i] == 'L'` 且 `i < j`，则不满足 *可到达性*。

```
class Solution {
public:
    bool canTransform(string start, string end) {
        if(start == end) return true;
        int i = 0, j = 0;
        int len = start.size();
        while(i < len || j < len){
            while(i < len && start[i] == 'X') i++;
            while(j < len && end[j] == 'X') j++;
            if(i >= len || j >= len)
                break;
            if(start[i] != end[j])//转换不变性
                return false;
            if(start[i] == 'L' && i < j)//可到达性
                return false;
            if(start[i] == 'R' && i > j)//可到达性
                return false;
            i++;
            j++;
        }
        return i==j;//必须同时结束，否则说明有一个字符串有多余的L或R
    }
};
```

528. 按权重随计选择（中等）

1. 题目描述

   给定一个正整数数组 w ， 其中 w[i] 代表位置 i 的权重，请写一个函数 pickIndex ， 它可以随机地获取位置 i，选取位置 i 的概率与 w[i] 成正比。

   说明:

   - 1 <= w.length <= 10000
   - 1 <= w[i] <= 10^5
   - pickIndex 将被调用不超过 10000 次

   ```
   示例1:
   输入:
   "Solution","pickIndex"],[]]
   输出: [null,0]

   示例2:
   输入:
   "Solution","pickIndex","pickIndex","pickIndex","pickIndex","pickIndex"],[],[],[],
   [],[]]
   输出: [null,0,1,1,1,0]
   ```

   输入语法说明：输入是两个列表：调用成员函数名和调用的参数。Solution 的构造函数有一个参数，即数组 w。pickIndex 没有参数。输入参数是一个列表，即使参数为空，也会输入一个 [] 空列表。

2. 简单实现

   想象一个数轴，以权重为各段的间隔

   ```
   class Solution {
   public:
       vector<long> data;//前缀和
       int size;
       Solution(vector<int>& w) {
           size = w.size();
           data = vector<long>(size);
           data[0] = w[0];
           for(int i = 1; i < w.size(); i++)
               data[i] = data[i-1] + w[i];
           srand((unsigned)time(NULL));
       }

       int pickIndex() {
           int n = rand() % data[size-1];
           return upper_bound(data.begin(), data.end(), n) - data.begin();
       }
   };
   ```

1153. 字符串转化（困难）

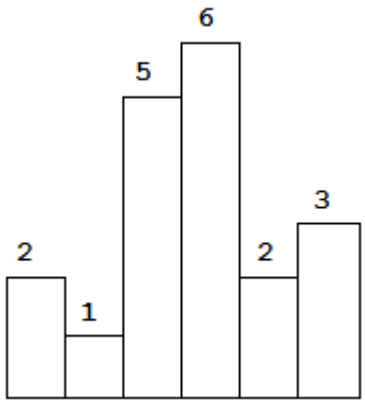
非会员

84. 柱状图的最大矩形（困难）

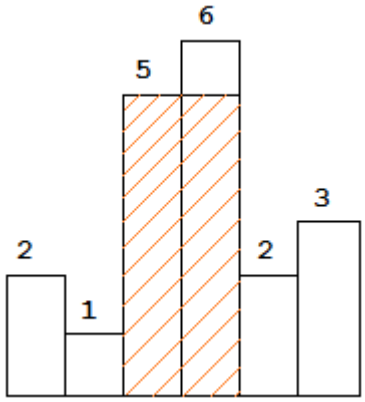
1. 题目描述

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 [2,1,5,6,2,3]。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

示例:

输入: [2,1,5,6,2,3] 输出: 10

2. 简单实现——单调栈

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        int ans = 0;
        vector<int> st;//单调栈，按高度递增存储柱子的索引
        //添加左右0边界，可以将原边界也统一运算
        heights.insert(heights.begin(), 0);
        heights.push_back(0);
        for (int i = 0; i < heights.size(); i++) {
            while (!st.empty() && heights[st.back()] > heights[i]) {//i号柱子为当前栈顶高度的右边界
                int cur = st.back();//为cur号柱子找到左右边界
                st.pop_back();
                int left = st.back() + 1;//cur号柱子左侧第一个比它矮的柱子的右边那根柱子
                int right = i - 1;//cur号柱子右侧第一个比它矮的柱子的左边那根柱子
                ans = max(ans, (right - left + 1) * heights[cur]);
```

```
            }
            st.push_back(i);
        }
        return ans;
    }
};
```

344. 反转字符串（简单）

1. 题目描述

   编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 char[] 的形式给出。

   不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用 O(1) 的额外空间解决这一问题。

   你可以假设数组中的所有字符都是 ASCII 码表中的可打印字符。

```
示例 1:
输入: ["h","e","l","l","o"]
输出: ["o","l","l","e","h"]

示例 2:
输入: ["H","a","n","n","a","h"]
输出: ["h","a","n","n","a","H"]
```

2. 简单实现

```
class Solution {
public:
    void reverseString(vector<char>& s) {
        int len = s.size();
        if(len <= 1) return;
        int l = 0, r = len -1;
        while(l < r)
            swap(s[l++], s[r--]);
    }
};
```

230. 二叉搜索树中第K小的元素（中等）

1. 题目描述

   给定一个二叉搜索树，编写一个函数 kthSmallest 来查找其中第 k 个最小的元素。

   说明：你可以假设 k 总是有效的，1 ≤ k ≤ 二叉搜索树元素个数。

```
示例 1:
输入: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
```

```
  \
   2
输出: 1

示例 2:
输入: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
输出: 3
```

进阶：如果二叉搜索树经常被修改（插入/删除操作）并且你需要频繁地查找第 k 小的值，你将如何优化 kthSmallest 函数?

2. 简单实现

```
class Solution {
public:
    unordered_map<TreeNode*, int> s;//左子树节点数
    int helper(TreeNode* root){
        if(!root) return 0;
        int l = helper(root->left);
        int r = helper(root->right);
        s[root] = l;
        return l + r + 1;
    }
    int kthSmallest(TreeNode* root, int k) {
        helper(root);
        while(1){
            if(s[root] == k-1)
                return root->val;
            else if(s[root] < k-1){
                k -= s[root] + 1;
                root = root->right;
            }
            else{
                root = root->left;
            }
        }
        return -1;
    }
};
```

3. 最优解法

```
class Solution {
public:
    int track(TreeNode* root, int &k){
        if(root->left){
```

```
            int val = track(root->left, k);
            if(k == 0)
                return val;
        }
        k--;
        if(k == 0)
            return root->val;
        if(root->right)
            return track(root->right, k);
        return -1;
    }
    int kthSmallest(TreeNode* root, int k) {
        return track(root, k);
    }
};
```

240. 搜索二维矩阵II（中等）

1. 题目描述

编写一个高效的算法来搜索 m x n 矩阵 matrix 中的一个目标值 target。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

```
示例:
现有矩阵 matrix 如下:
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
给定 target = 5，返回 true。
给定 target = 20，返回 false。
```

2. 简单实现

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size();
        if(m == 0) return false;
        int n = matrix[0].size();
        if(n == 0) return false;
        int x = 0, y = n - 1;
        while(x < m && y >= 0){
            if(matrix[x][y] == target)
                return true;
            else if(matrix[x][y] > target)
                y--;
```

```
            else
                x++;
        }
        return false;
    }
};
```

259. 较小的三数之和（中等）

要会员

1146. 快照数组（中等）

1. 题目描述

   实现支持下列接口的「快照数组」- SnapshotArray：

   - SnapshotArray(int length) - 初始化一个与指定长度相等的 类数组 的数据结构。初始时，每个元素都等于 0。
   - void set(index, val) - 会将指定索引 index 处的元素设置为 val。
   - int snap() - 获取该数组的快照，并返回快照的编号 snap_id（快照号是调用 snap() 的总次数减去 1）。
   - int get(index, snap_id) - 根据指定的 snap_id 选择快照，并返回该快照指定索引 index 的值。

```
示例：
输入：["SnapshotArray","set","snap","set","get",[0,5],[],[0,6],[0,0]]
输出：[null,null,0,null,5]
解释：
SnapshotArray snapshotArr = new SnapshotArray(3); // 初始化一个长度为 3 的快照数组
snapshotArr.set(0,5);  // 令 array[0] = 5
snapshotArr.snap();  // 获取快照，返回 snap_id = 0
snapshotArr.set(0,6);
snapshotArr.get(0,0);  // 获取 snap_id = 0 的快照中 array[0] 的值，返回 5
```

提示：

- 1 <= length <= 50000
- 题目最多进行50000 次set，snap，和 get的调用 。
- 0 <= index < length
- 0 <= snap_id < 我们调用 snap() 的总次数
- 0 <= val <= 10^9

2. 简单实现

   利用map记录每个index对应的数值在不同时间的变化

```
class SnapshotArray {
public:
    int id;//当前时间id
    unordered_map<int, map<int, int>> data;//<index, <id, value>>
    SnapshotArray(int length) {
        id = 0;
```

```
    }
    void set(int index, int val) {
        data[index][id] = val;
    }
    int snap() {
        return id++;
    }
    int get(int index, int snap_id) {
        auto it = data[index].upper_bound(snap_id);
        if(it == data[index].begin())//在snap_id前未改变过该位置的数值
            return 0;
        it--;
            return it->second;
    }
};
```

285. 二叉搜索树中的顺序后继（中等）

要会员

167. 两数之和 II - 输入有序数组（简单）

1. 题目描述

   给定一个已按照升序排列 的有序数组，找到两个数使得它们相加之和等于目标数。

   函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

   说明:

   - 返回的下标值（index1 和 index2）不是从零开始的。
   - 你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

```
示例:
输入: numbers = [2, 7, 11, 15], target = 9
输出: [1,2]
解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。
```

2. 简单实现

```
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        int l = 0;
        int r = numbers.size() - 1;
        while(l < r){
            int sum = numbers[l] + numbers[r];
            if(sum == target)
                return {l+1, r+1};
            else if(sum > target)
                r--;
            else
```

```
                l++;
            }
            return {-1, -1};
        }
};
```

465. 最优账单平衡（困难）

要会员

130. 被围绕的区域（中等）

1. 题目描述

   给定一个二维的矩阵，包含 'X' 和 'O'（字母 O）。

   找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

```
示例:
X X X X
X O O X
X X O X
X O X X

运行你的函数后，矩阵变为:
X X X X
X X X X
X X X X
X O X X
解释:
被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。 任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。
```

2. 简单实现

   BFS找到与边界O相连的O，剩下的O都变成X

```
class Solution {
public:
    void solve(vector<vector<char>>& board) {
        int m = board.size();
        if(m <= 2) return;
        int n = board[0].size();
        if(n <= 2) return;
        queue<pair<int, int>> q;
        vector<vector<bool>> visited = vector<vector<bool>>(m, vector<bool>(n,
false));
        for(int i = 0; i < m; i++){
            if(board[i][0] == 'O'){
                visited[i][0] = true;
```

```
                q.push(make_pair(i,0));
            }
            if(board[i][n-1] == 'O'){
                visited[i][n-1] = true;
                q.push(make_pair(i,n-1));

            }
        }
        for(int j = 0; j < n; j++){
            if(board[0][j] == 'O'){
                visited[0][j] = true;
                q.push(make_pair(0,j));
            }
            if(board[m-1][j] == 'O'){
                visited[m-1][j] = true;
                q.push(make_pair(m-1,j));
            }
        }
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                int x = q.front().first;
                int y = q.front().second;
                q.pop();
                if(x-1>=0 && !visited[x-1][y] && board[x-1][y]=='O'){
                    visited[x-1][y] = true;
                    q.push(make_pair(x-1,y));
                }
                if(x+1<m && !visited[x+1][y] && board[x+1][y]=='O'){
                    visited[x+1][y] = true;
                    q.push(make_pair(x+1,y));
                }
                 if(y-1>=0 && !visited[x][y-1] && board[x][y-1]=='O'){
                    visited[x][y-1] = true;
                    q.push(make_pair(x,y-1));
                }
                if(y+1<n && !visited[x][y+1] && board[x][y+1]=='O'){
                    visited[x][y+1] = true;
                    q.push(make_pair(x,y+1));
                }
            }
        }
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(board[i][j] == 'O' && visited[i][j] == false)
                    board[i][j] = 'X';
    }
};
```

171. Excel表列序号（简单）

1. 题目描述

给定一个Excel表格中的列名称，返回其相应的列序号。

例如，

```
A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
...

示例 1:
输入: "A"
输出: 1

示例 2:
输入: "AB"
输出: 28

示例 3:
输入: "ZY"
输出: 701
```

2. 简单实现

26进制转10进制

```
class Solution {
public:
    int titleToNumber(string s) {
        int ans = 0;
        for(int i = 0; i < s.size(); i++)
            ans = ans*26 + (s[i] - 'A' + 1);
        return ans;
    }
};
```

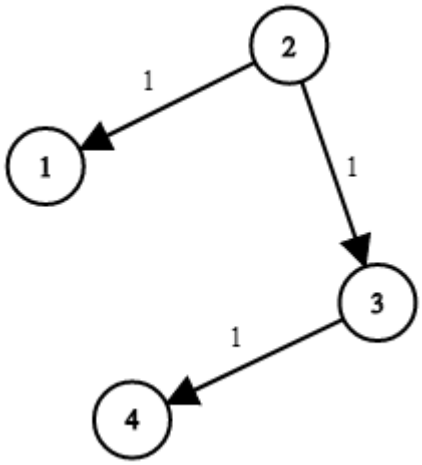
743. 网络延迟时间（中等）

1. 题目描述

有 N 个网络节点，标记为 1 到 N。给定一个列表 times，表示信号经过有向边的传递时间。 times[i] = (u, v, w)，其中 u 是源节点，v 是目标节点， w 是一个信号从源节点传递到目标节点的时间。

现在，我们从某个节点 K 发出一个信号。需要多久才能使所有节点都收到信号？如果不能使所有节点收到信号，返回 -1。

示例：



输入：times = [[2,1,1],[2,3,1],[3,4,1]], N = 4, K = 2 输出：2

注意:

- N 的范围在 [1, 100] 之间。
- K 的范围在 [1, N] 之间。
- times 的长度在 [1, 6000] 之间。
- 所有的边 times[i] = (u, v, w) 都有 1 <= u, v <= N 且 0 <= w <= 100。

2. 简单实现

迪杰斯特拉算法

```
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int N, int K) {
        unordered_map<int, unordered_map<int, int>> g;//图
        for(int i = 0; i < times.size(); i++)
            g[times[i][0]][times[i][1]] = times[i][2];
        unordered_map<int,int> t;//从K到各点的传输延迟
        priority_queue<pair<int, int>, vector<pair<int, int>>,
                       greater<pair<int, int>>> q;//升序
        q.push(make_pair(0, K));
        while(!q.empty()){
            //取出目前未访问的离K最近的点
            int d = q.top().first;
            int cur = q.top().second;
            q.pop();
            if(t.find(cur) != t.end()) continue;
            t[cur] = d;//记录最近距离
            for(auto it = g[cur].begin(); it != g[cur].end(); it++){//相邻点
                if(t.find(cur) != t.end())
                    q.push(make_pair(d+it->second, it->first));
            }
        }
        if(t.size() < N) return -1;
        int ans = 0;
        for(auto it = t.begin(); it != t.end(); it++)
            ans = max(ans, it->second);
        return ans;
```

```
    }
};
```

26. 删除排序数组中的重复项（简单）

1. 题目描述

   给定一个排序数组，你需要在 原地 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

   不要使用额外的数组空间，你必须在 原地 修改输入数组 并在使用 O(1) 额外空间的条件下完成。

```
示例 1:
给定数组 nums = [1,1,2],
函数应该返回新的长度 2, 并且原数组 nums 的前两个元素被修改为 1, 2。
你不需要考虑数组中超出新长度后面的元素。

示例 2:
给定 nums = [0,0,1,1,1,2,2,3,3,4],
函数应该返回新的长度 5, 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。
你不需要考虑数组中超出新长度后面的元素。
```

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);
// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度, 它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

2. 简单实现

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int len = nums.size();
        if(len <= 0)
            return 0;
        int index = 0, cur = 1;
        while(cur < len){
            if(nums[index] != nums[cur]){
                nums[++index] = nums[cur];
            }
            cur++;
```

```
        }
        return index+1;
    }
};
```

101. 对称二叉树（简单）

1. 题目描述

   给定一个二叉树，检查它是否是镜像对称的。

   例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

   但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:

```
   1
  / \
  2   2
   \   \
   3    3
```

   进阶：你可以运用递归和迭代两种方法解决这个问题吗?

2. 简单实现

```
class Solution {
public:
    bool isequal(TreeNode* root1, TreeNode* root2){
        if(root1 == NULL && root2 == NULL)
            return true;
        else if((root1 == NULL && root2 != NULL) || (root1 != NULL && root2 ==
NULL))
            return false;

        if(root1->val != root2->val)
            return false;
        else
            return isequal(root1->left, root2->right) && isequal(root1->right,
root2->left);
    }
    bool isSymmetric(TreeNode* root) {
        if(!root)
            return true;
        return isequal(root->left, root->right);
    }
```

```
};
```

方法二：迭代

除了递归的方法外，我们也可以利用队列进行迭代。队列中每两个连续的结点应该是相等的，而且它们的子树互为镜像。最初，队列中包含的是 `root` 以及 `root` 。该算法的工作原理类似于 BFS，但存在一些关键差异。每次提取两个结点并比较它们的值。然后，将两个结点的左右子结点按相反的顺序插入队列中。当队列为空时，或者我们检测到树不对称（即从队列中取出两个不相等的连续结点）时，该算法结束。

Java

```java
public boolean isSymmetric(TreeNode root) {
    Queue<TreeNode> q = new LinkedList<>();
    q.add(root);
    q.add(root);
    while (!q.isEmpty()) {
        TreeNode t1 = q.poll();
        TreeNode t2 = q.poll();
        if (t1 == null && t2 == null) continue;
        if (t1 == null || t2 == null) return false;
        if (t1.val != t2.val) return false;
        q.add(t1.left);
        q.add(t2.right);
        q.add(t1.right);
        q.add(t2.left);
    }
    return true;
}
```

249. 移位字符串分组（中等）

要会员

152. 乘积最大子数组（中等）

1. 题目描述

   给你一个整数数组 nums ，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字）。

```
示例 1:
输入: [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。

示例 2:
输入: [-2,0,-1]
输出: 0
解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。
```

2. 简单实现

摩尔投票法遍历并维护乘积的最大最小值

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int ans = INT_MIN, imax = 1, imin = 1;
        for(int i=0; i<nums.size(); i++){
            if(nums[i] < 0){ //负的，两个最值互换
              int tmp = imax;
              imax = imin;
              imin = tmp;
            }
            imax = max(imax*nums[i], nums[i]);//因为可能会有0
            imin = min(imin*nums[i], nums[i]);//因为可能会有0

            ans = max(ans, imax);
        }
        return ans;
    }
};
```

108. 将有序数组转换为二叉搜索树（简单）

1. 题目描述

将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

```
示例:
给定有序数组: [-10,-3,0,5,9],
一个可能的答案是: [0,-3,9,-10,null,5], 它可以表示下面这个高度平衡二叉搜索树:
          0
     / \
   -3   9
   /   /
 -10  5
```

2. 简单实现

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums, int l, int r){
        if(l > r) return NULL;
        if(l == r) return new TreeNode(nums[l]);
        int mid = l + (r - l) / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = sortedArrayToBST(nums, l, mid - 1);
        root->right = sortedArrayToBST(nums, mid + 1, r);
        return root;
    }
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return sortedArrayToBST(nums, 0, nums.size()-1);
    }
};
```

153.寻找旋转排序数组中的最小值（中等）

1. 题目描述

   假设按照升序排序的数组在预先未知的某个点上进行了旋转。(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。

   请找出其中最小的元素。你可以假设数组中不存在重复元素。

```
示例 1:
输入: [3,4,5,1,2]
输出: 1

示例 2:
输入: [4,5,6,7,0,1,2]
输出: 0
```

2. 简单实现

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        if(nums.size() == 1)
            return nums[0];
        int l = 0;
        int r = nums.size() - 1;
        if(nums[l] < nums[r]) return nums[0];
        while(l < r){
            int mid = l + (r - l) / 2;
            if(nums[mid] >= nums[0])//左半部
                l = mid + 1;
            else if(nums[mid] < nums[mid-1])
                return nums[mid];
            else
                r = mid;
```

```
        }
        return nums[l];
    }
};
```

102. 二叉树的层序遍历（中等）

1. 题目描述

给你一个二叉树，请你返回其按 层序遍历 得到的节点值。 （即逐层地，从左到右访问所有节点）。

```
示例：
二叉树：[3,9,20,null,null,15,7],
    3
   / \
  9  20
    /  \
   15   7
返回其层次遍历结果：
[
  [3],
  [9,20],
  [15,7]
]
```

2. 简单实现

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        if(!root) return vector<vector<int>>();
        vector<vector<int>> ans;
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()){
            int size = q.size();
            vector<int> cur = vector<int>(size);
            for(int i = 0; i < size; i++){
                TreeNode* temp = q.front();
                q.pop();
                cur[i] = temp->val;
                if(temp->left) q.push(temp->left);
                if(temp->right) q.push(temp->right);
            }
            ans.push_back(cur);
        }
        return ans;
    }
};
```

425. 单词方块（困难）

要会员

168. Excel表列名称（简单）

1. 题目描述

   给定一个正整数，返回它在 Excel 表中相对应的列名称。

   例如，

```
1 -> A
2 -> B
3 -> C
...
26 -> Z
27 -> AA
28 -> AB
...
```

```
示例 1:
输入: 1
输出: "A"

示例 2:
输入: 28
输出: "AB"

示例 3:
输入: 701
输出: "ZY"
```

2. 简单实现

   十进制转二十六进制

```
class Solution {
public:
    string convertToTitle(int n) {
        string ans = "";
        while(n){
            n--;//转换成0开始的
            int cur = n % 26;
            ans += 'A' + cur;
            n /= 26;
        }
        reverse(ans.begin(), ans.end());
        return ans;
    }
};
```

9. 回文数（简单）

1. 题目描述

   判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

   ```
   示例 1:
   输入: 121
   输出: true

   示例 2:
   输入: -121
   输出: false
   解释: 从左向右读, 为 -121 。 从右向左读, 为 121- 。因此它不是一个回文数。

   示例 3:
   输入: 10
   输出: false
   解释: 从右向左读, 为 01 。因此它不是一个回文数。
   ```

   进阶: 你能不将整数转为字符串来解决这个问题吗?

2. 简单实现

   不转字符串，倒着求一遍这个数，应该相等

   ```
   class Solution {
   public:
       bool isPalindrome(int x) {
           if(x < 0)
               return false;
           int cache = x;
           long inv = 0;//long防止溢出
           while(x>0){
               inv = inv*10 + x%10;
               x /= 10;
           }
           return cache==inv;
       }
   };
   ```

459. 重复的子字符串（简单）

1. 题目描述

   给定一个非空的字符串，判断它是否可以由它的一个子串重复多次构成。给定的字符串只含有小写英文字母，并且长度不超过10000。

```
示例 1:
输入: "abab"
输出: True
解释: 可由子字符串 "ab" 重复两次构成。

示例 2:
输入: "aba"
输出: False

示例 3:
输入: "abcabcabcabc"
输出: True
解释: 可由子字符串 "abc" 重复四次构成。 (或者子字符串 "abcabc" 重复两次构成。)
```

2. 正确解法

解题思路

这题有参考别人的想法，换做我确实很难想到，但还是用我自己的话总结下思路吧。
思路大致如下：如果一个非空字符串s可以由它的一个子串重复多次构成，可以理解为s中存在m个子串，那么当两个字符串结合起来变成ss时，字符串s在新字符串ss的第二次位置不等于s的长度（相当于前一个字符串s中有n个子串，在后一个字符串中有m-n个子串，所以此时的位置不等于s的长度）；反之，一个非空字符串s不可以由它的一个子串重复多次构成，那么当两个字符串结合起来变成ss时，字符串s在新字符串ss的第二次位置就在后一个字符串首字符的位置，其位置刚好等于s的长度。根据这一特征来判断。

```cpp
class Solution {
public:
    bool repeatedSubstringPattern(string s) {
        return (s+s).find(s,1)!=s.size();
    }
};
```

568. 最大休假天数（困难）

要会员

731. 我的日程安排表II（中等）

1. 题目描述

实现一个 MyCalendar 类来存放你的日程安排。如果要添加的时间内不会导致三重预订时，则可以存储这个新的日程安排。

MyCalendar 有一个 book(int start, int end)方法。它意味着在 start 到 end 时间内增加一个日程安排，注意，这里的时间是半开区间，即 [start, end), 实数 x 的范围为， start <= x < end。

当三个日程安排有一些时间上的交叉时（例如三个日程安排都在同一时间内），就会产生三重预订。

每次调用 MyCalendar.book方法时，如果可以将日程安排成功添加到日历中而不会导致三重预订，返回 true。否则，返回 false 并且不要将该日程安排添加到日历中。

请按照以下步骤调用MyCalendar 类: MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

```
示例:
MyCalendar();
MyCalendar.book(10, 20); // returns true
MyCalendar.book(50, 60); // returns true
MyCalendar.book(10, 40); // returns true
MyCalendar.book(5, 15); // returns false
MyCalendar.book(5, 10); // returns true
MyCalendar.book(25, 55); // returns true
解释:
前两个日程安排可以添加至日历中。 第三个日程安排会导致双重预订，但可以添加至日历中。
第四个日程安排活动（5,15）不能添加至日历中，因为它会导致三重预订。
第五个日程安排（5,10）可以添加至日历中，因为它未使用已经双重预订的时间10。
第六个日程安排（25,55）可以添加至日历中，因为时间 [25,40] 将和第三个日程安排双重预订;
时间 [40,50] 将单独预订，时间 [50,55）将和第二个日程安排双重预订。
```

提示:

- 每个测试用例，调用 MyCalendar.book 函数最多不超过 1000次。
- 调用函数 MyCalendar.book(start, end)时， start 和 end 的取值范围为 [0, 10^9]。

2. 简单实现

维护一重预订列表和双重预订列表。当预订一个新的日程安排 `[start, end)` 时，如果它与双重预订列表冲突，则会产生三重预定。

```
class MyCalendarTwo {
public:
    MyCalendarTwo() {
        records.clear();
        intersects.clear();
    }

    typedef pair<int,int> interval;
    interval find_repeat(interval &i1, interval &i2) {
        interval i;
        i.first = max(i1.first, i2.first);
        i.second = min(i1.second, i2.second);
        return i;
    }
    bool real_interval(interval &i) {
        if (i.first < i.second)
            return true;
        else
            return false;
    }
    vector<interval> records;
    vector<interval> intersects;

    bool book(int start, int end) {
```

```
            interval i;
            i.first = start;
            i.second = end;
            for (auto intersect : intersects) {//双重预定
                interval result = find_repeat(i, intersect);
                if (real_interval(result))
                    return false;
            }

            for (auto record : records) {//一重预定
                interval result = find_repeat(i, record);
                if (real_interval(result))
                    intersects.push_back(result);
            }
            records.push_back(i);

            return true;
        }
    };
```

852. 山脉数组的峰顶索引（简单）

1. 题目描述

   我们把符合下列属性的数组 A 称作山脉：

   - A.length >= 3
   - 存在 0 < i < A.length - 1 使得A[0] < A[1] < ... A[i-1] < A[i] > A[i+1] > ... > A[A.length - 1]

   给定一个确定为山脉的数组，返回任何满足 A[0] < A[1] < ... A[i-1] < A[i] > A[i+1] > ... > A[A.length - 1] 的 i 的值。

```
示例 1：
输入：[0,1,0]
输出：1

示例 2：
输入：[0,2,1,0]
输出：1
```

提示：

- 3 <= A.length <= 10000
- 0 <= A[i] <= 10^6
- A 是如上定义的山脉

2. 简单实现

```
class Solution {
public:
    int peakIndexInMountainArray(vector<int>& A) {
```

```
        int l = 0, r = A.size() - 1;
        while(l <= r){
            int mid = l + (r - l) / 2;
            if(A[mid] > A[mid-1] && A[mid] > A[mid+1])
                return mid;
            else if(A[mid] > A[mid-1])
                l = mid + 1;
            else
                r = mid - 1;
        }
        return -1;
    }
};
```

63. 不同路径II（中等）

1. 题目描述

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径?

网格中的障碍物和空位置分别用 1 和 0 来表示。

说明：m 和 n 的值均不超过 100。

```
示例 1:

输入:
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
输出: 2
解释:
3x3 网格的正中间有一个障碍物。
从左上角到右下角一共有 2 条不同的路径:
1. 向右 -> 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右 -> 向右
```

2. 简单实现

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        vector<long> dp(n);//long防止后面可能出现的加法溢出
        int i = 0;
        while(i < n && obstacleGrid[0][i] != 1)
            dp[i++] = 1;
        while(i < n)
            dp[i++] = 0;
        for(int i = 1; i < m; i++){
            if(obstacleGrid[i][0] == 1)
                dp[0] = 0;
            for(int j = 1; j < n; j++){
                if(obstacleGrid[i][j] == 1)
                    dp[j] = 0;
                else
                    dp[j] += dp[j-1];
            }
        }
        return dp[n-1];
    }
};
```

973. 最接近原点的K个点（中等）

1. 题目描述

我们有一个由平面上的点组成的列表 points。需要从中找出 K 个距离原点 (0, 0) 最近的点。

（这里，平面上两点之间的距离是欧几里德距离。）

你可以按任何顺序返回答案。除了点坐标的顺序之外，答案确保是唯一的。

```
示例 1:
输入: points = [[1,3],[-2,2]], K = 1
输出: [[-2,2]]
解释:
(1, 3) 和原点之间的距离为 sqrt(10),
(-2, 2) 和原点之间的距离为 sqrt(8),
由于 sqrt(8) < sqrt(10), (-2, 2) 离原点更近。
我们只需要距离原点最近的 K = 1 个点，所以答案就是 [[-2,2]]。

示例 2:
输入: points = [[3,3],[5,-1],[-2,4]], K = 2
输出: [[3,3],[-2,4]]
（答案 [[-2,4],[3,3]] 也会被接受。）
```

提示:

- 1 <= K <= points.length <= 10000
- `-10000 < points[i][0] < 10000`
- `-10000 < points[i][1] < 10000`

2. 简单实现

```
class Solution {
public:
    vector<vector<int>> kClosest(vector<vector<int>>& points, int K) {
        priority_queue<pair<int, int>> q;
        for(int i = 0; i < points.size(); i++){
            int dis = pow(points[i][0], 2) + pow(points[i][1], 2);
            if(q.size() < K)
                q.push(make_pair(dis, i));
            else if(dis < q.top().first){
                q.pop();
                q.push(make_pair(dis, i));
            }
        }
        vector<vector<int>> ans;
        while(!q.empty()){
            ans.push_back(points[q.top().second]);
            q.pop();
        }
        return ans;
    }
};
```

3. 最优解法——类似快排的分治法

思路

我们想要一个复杂度比 $N \log N$ 更低的算法。 显然，做到这件事情的唯一办法就是利用题目中可以按照任何顺序返回 K 个点的条件，否则的话，必要的排序将会话费我们 $N \log N$ 的时间。

我们随机地选择一个元素 `x = A[i]` 然后将数组分为两部分： 一部分是到原点距离小于 `x` 的，另一部分是到原点距离大于等于 `x` 的。 这个快速选择的过程与快速排序中选择一个关键元素将数组分为两部分的过程类似。

如果我们快速选择一些关键元素，那么每次就可以将问题规模缩减为原来的一半，平均下来时间复杂度就是线性的。

算法

我们定义一个函数 `work(i, j, K)` ，它的功能是部分排序 `(points[i], points[i+1], ..., points[j])` 使得最小的 `K` 个元素出现在数组的首部，也就是 `(i, i+1, ..., i+K-1)` 。

首先，我们从数组中选择一个随机的元素作为关键元素，然后使用这个元素将数组分为上述的两部分。为了能使用线性时间的完成这件事，我们需要两个指针 `i` 与 `j` ，然后将它们移动到放错了位置元素的地方，然后交换这些元素。

然后，我们就有了两个部分 `[oi, i]` 与 `[i+1, oj]` ，其中 `(oi, oj)` 是原来调用 `work(i, j, K)` 时候 `(i, j)` 的值。假设第一部分有 `10` 个元，第二部分有 `15` 个元素。如果 `K = 5` 的话，我们只需要对第一部分调用 `work(oi, i, 5)` 。否则的话，假如说 `K = 17` ，那么第一部分的 `10` 个元素应该都需要被选择，我们只需要对第二部分调用 `work(i+1, oj, 7)` 就行了。

```
class Solution {
public:
    vector<vector<int>> kClosest(vector<vector<int>>& points, int K) {
        partial_sort(points.begin(), points.begin() + K, points.end(),
                     [] (auto & p1, auto & p2) {
                        int d1 = p1[0] * p1[0] + p1[1] * p1[1];
                        int d2 = p2[0] * p2[0] + p2[1] * p2[1];
                        return d1 < d2;});
        points.resize(K);
        return points;
    }
};
```