前言

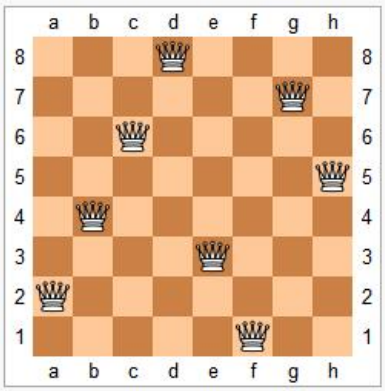
大家好，我是爱学习爱分享的沉默王二。微信搜索「**沉默王二**」可以关注我的原创公众号，回复关键字「**1024**」就可以拉取离线版的《JavaGuide 面试突击》下载地址。

不经意间，在 GitHub 上发现了一个 1G 棒的 LeetCode 刷题笔记，重点来了，是纯正的 Java 版。我见过很多牛逼的刷题笔记，有 Go 版的，有 C++ 版的，唯独没有 Java 版的，所以这次，我感觉找到了宝藏!

题解预览地址:https://leetcode.wang，推荐电脑端打开，手机打开的话将页面滑到最上边，左上角是 菜单
leetcode 题目地址 https://leetcode.com/problemset/all/
github 项目地址:https://github.com/wind-liang/leetcode

51、题目描述（困难难度）

The *n*-queens puzzle is the problem of placing *n* queens on an *n*×*n* chessboard such that no two queens attack each other.



One solution to the eight queens puzzle

Given an integer *n*, return all distinct solutions to the *n*-queens puzzle.

Each solution contains a distinct board configuration of the *n*-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

Example:

```
Input: 4
Output: [
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above.
```

经典的 N 皇后问题。意思就是摆皇后的位置，每行每列以及对角线只能出现 1 个皇后。输出所有的情况。

解法一 回溯法

比较经典的回溯问题了，我们需要做的就是先在第一行放一个皇后，然后进入回溯，放下一行皇后的位置，一直走下去，如果已经放的皇后的数目等于 n 了，就加到最后的结果中。然后再回到上一行，变化皇后的位置，然后去找其他的解。

期间如果遇到当前行所有的位置都不能放皇后了，就再回到上一行，然后变化皇后的位置。再返回到下一行。

说起来可能还费力些，直接看代码吧。

```
public List<List<String>> solveNQueens(int n) {
    List<List<String>> ans = new ArrayList<>();
    backtrack(new ArrayList<Integer>(), ans, n);
```

```
    return ans;
}

private void backtrack(List<Integer> currentQueen, List<List<String>> ans, int n)
{
    // 当前皇后的个数是否等于 n 了，等于的话就加到结果中
    if (currentQueen.size() == n) {
        List<String> temp = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            char[] t = new char[n];
            Arrays.fill(t, '.');
            t[currentQueen.get(i)] = 'Q';
            temp.add(new String(t));
        }
        ans.add(temp);
        return;
    }
    //尝试每一列
    for (int col = 0; col < n; col++) {
        //当前列是否冲突
        if (!currentQueen.contains(col)) {
            //判断对角线是否冲突
            if (isDiagonalAttack(currentQueen, col)) {
                continue;
            }
            //将当前列的皇后加入
            currentQueen.add(col);
            //去考虑下一行的情况
            backtrack(currentQueen, ans, n);
            //将当前列的皇后移除，去判断下一列
            //进入这一步就是两种情况，下边的行走不通了回到这里或者就是已经拿到了一个解回到这里
            currentQueen.remove(currentQueen.size() - 1);
        }

    }

}

private boolean isDiagonalAttack(List<Integer> currentQueen, int i) {
    // TODO Auto-generated method stub
    int current_row = currentQueen.size();
    int current_col = i;
    //判断每一行的皇后的情况
    for (int row = 0; row < currentQueen.size(); row++) {
        //左上角的对角线和右上角的对角线，差要么相等，要么互为相反数，直接写成了绝对值
```

```
        if (Math.abs(current_row - row) == Math.abs(current_col -
currentQueen.get(row))) {
            return true;
        }
    }
    return false;
}
```

时间复杂度：

空间复杂度：

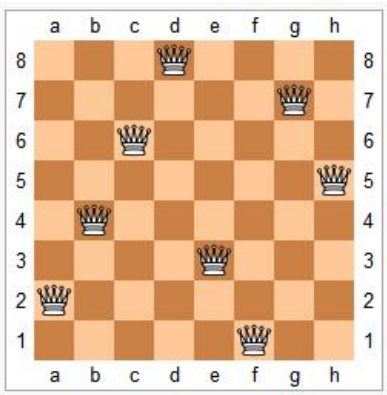
上边我们只判断了列冲突和对角线冲突，至于行冲突，由于我们采取一行一行加皇后，所以一行只会有一个皇后，不会产生冲突。

总

最早接触的一类问题了，学回溯法的话，一般就会以这个为例，所以思路上不会遇到什么困难。

52、题目描述（困难难度）

The *n*-queens puzzle is the problem of placing *n* queens on an *n*×*n* chessboard such that no two queens attack each other.



One solution to the eight queens puzzle

Given an integer *n*, return the number of distinct solutions to the *n*-queens puzzle.

Example:

```
Input: 4
Output: 2
Explanation: There are two distinct solutions to the 4-queens puzzle as shown below.
[
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

和上一题一样，只不过这次不需要返回所有结果，只需要返回有多少个解就可以。

解法一

我们直接把上道题的 ans 的 size 返回就可以了，此外 currentQueen.size ( ) == n 的时候，也不用去生成一个解了，直接加一个数字占位。

```
public int totalNQueens(int n) {
    List<Integer> ans = new ArrayList<>();
    backtrack(new ArrayList<Integer>(), ans, n);
    return ans.size();
}

private void backtrack(List<Integer> currentQueen, List<Integer> ans, int n) {
    if (currentQueen.size() == n) {
        ans.add(1);
        return;
```

```
    }
    for (int col = 0; col < n; col++) {
        if (!currentQueen.contains(col)) {
            if (isDiagonalAttack(currentQueen, col)) {
                continue;
            }
            currentQueen.add(col);
            backtrack(currentQueen, ans, n);
            currentQueen.remove(currentQueen.size() - 1);
        }

    }

}

private boolean isDiagonalAttack(List<Integer> currentQueen, int i) {
    int current_row = currentQueen.size();
    int current_col = i;
    for (int row = 0; row < currentQueen.size(); row++) {
        if (Math.abs(current_row - row) == Math.abs(current_col -
currentQueen.get(row))) {
            return true;
        }
    }
    return false;
}
```

时间复杂度：

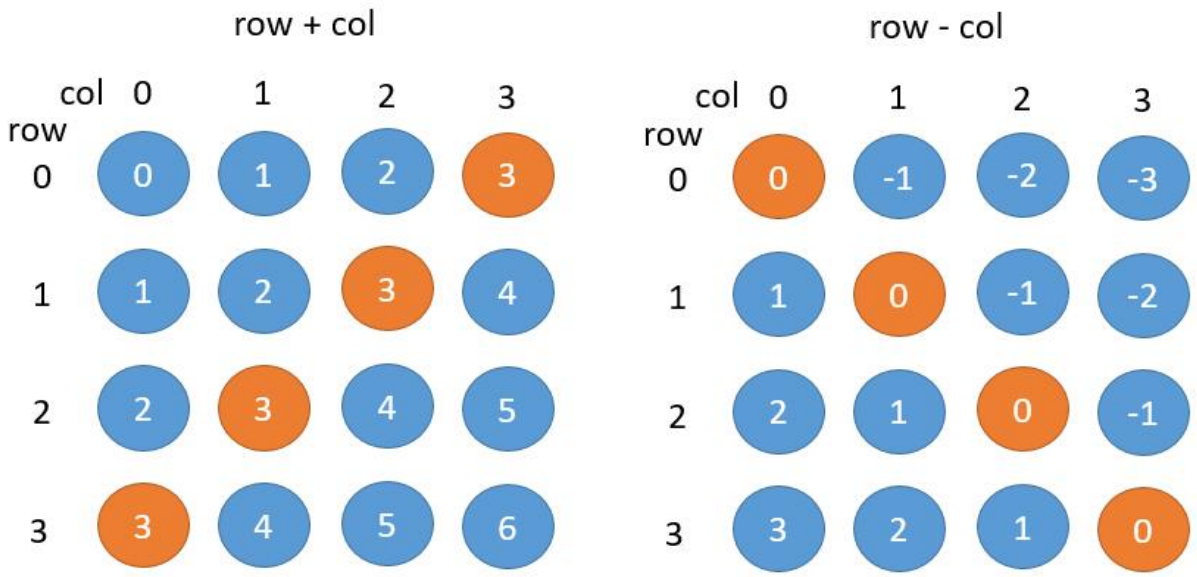
空间复杂度：

解法二

参考这里。

既然不用返回所有解，那么我们就不需要 currentQueen 来保存当前已加入皇后的位置。只需要一个 bool 型数组，来标记列是否被占有就可以了。

由于没有了 currentQueen，所有不能再用之前 isDiagonalAttack 判断对角线冲突的方法了。我们可以观察下，对角线元素的情况。



可以发现对于同一条副对角线，row + col 的值是相等的。

对于同一条主对角线，row - col 的值是相等的。

我们同样可以用一个 bool 型数组，来保存当前对角线是否有元素，把它们相加相减的值作为下标。

对于 row - col ，由于出现了负数，所以可以加 1 个 n，由 [- 3, 3] 转换为 [1 , 7] 。

```
public int totalNQueens(int n) {
    List<Integer> ans = new ArrayList<>();
    boolean[] cols = new boolean[n]; // 列
    boolean[] d1 = new boolean[2 * n]; // 主对角线
    boolean[] d2 = new boolean[2 * n]; // 副对角线
    return backtrack(0, cols, d1, d2, n, 0);
}

private int backtrack(int row, boolean[] cols, boolean[] d1, boolean[] d2, int n,
int count) {
    if (row == n) {
        count++;
    } else {
        for (int col = 0; col < n; col++) {
            int id1 = row - col + n; //主对角线加 n
            int id2 = row + col;
            if (cols[col] || d1[id1] || d2[id2])
                continue;
            cols[col] = true;
            d1[id1] = true;
            d2[id2] = true;
            count = backtrack(row + 1, cols, d1, d2, n, count);
```

```
                cols[col] = false;
                d1[id1] = false;
                d2[id2] = false;
            }

        }
        return count;
    }
```

时间复杂度：

空间复杂度：

总

和上一题相比，通过三个 bool 型数组来标记是否占有，不存储具体的位置，从而解决了这道题。

53、题目描述（简单难度）

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example:

```
Input: [-2,1,-3,4,-1,2,1,-5,4],
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

Follow up:

If you have figured out the O(*n*) solution, try coding another solution using the divide and conquer approach, which is more subtle.

给一个数组，找出一个连续的子数组，长度任意，和最大。

解法一 动态规划思路一

用一个二维数组 dp[i] [len] 表示从下标 i 开始，长度为 len 的子数组的元素和。

这样长度是 len + 1 的子数组就可以通过长度是 len 的子数组去求，也就是下边的递推式，

dp [i] [len + 1] = dp[i] [len] + nums [i + len - 1]。

当然，和第 5 题一样，考虑到求 i + 1 的情况的时候，我们只需要 i 时候的情况，所有我们其实没必要用一个二维数组，直接用一维数组就可以了。

```
public int maxSubArray(int[] nums) {
```

```
    int n = nums.length;
    int[] dp = new int[n];
    int max = Integer.MIN_VALUE;
    for (int len = 1; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            //直接覆盖掉前边对应的情况就行
            dp[i] = dp[i] + nums[i + len - 1];
            //更新 max
            if (dp[i] > max) {
                max = dp[i];
            }
        }
    }
    return max;
}
```

时间复杂度：O（n^2）。

空间复杂度：O（n）。

解法二 动态规划思路二

参考这里。

用一个一维数组 dp [i] 表示以下标 i 结尾的子数组的元素的最大的和，也就是这个子数组最后一个元素是下边为 i 的元素，并且这个子数组是所有以 i 结尾的子数组中，和最大的。

这样的话就有两种情况，

- 如果 dp [i - 1] < 0，那么 dp [i] = nums [i]。
- 如果 dp [i - 1] >= 0，那么 dp [i] = dp [i - 1] + nums [i]。

```
public int maxSubArray(int[] nums) {
    int n = nums.length;
    int[] dp = new int[n];
    int max = nums[0];
    dp[0] = nums[0];
    for (int i = 1; i < n; i++) {
        //两种情况更新 dp[i]
        if (dp[i - 1] < 0) {
            dp[i] = nums[i];
        } else {
            dp[i] = dp[i - 1] + nums[i];
        }
        //更新 max
        max = Math.max(max, dp[i]);
    }
```

```
    return max;
}
```

时间复杂度： O（n）。

空间复杂度：O（n）。

当然，和以前一样，我们注意到更新 i 的情况的时候只用到 i - 1 的时候，所以我们不需要数组，只需要两个变量。

```
public int maxSubArray(int[] nums) {
    int n = nums.length;
    //两个变量即可
    int[] dp = new int[2];
    int max = nums[0];
    dp[0] = nums[0];
    for (int i = 1; i < n; i++) {
        //利用求余，轮换两个变量
        if (dp[(i - 1) % 2] < 0) {
            dp[i % 2] = nums[i];
        } else {
            dp[i % 2] = dp[(i - 1) % 2] + nums[i];
        }
        max = Math.max(max, dp[i % 2]);
    }
    return max;
}
```

时间复杂度： O（n）。

空间复杂度：O（1）。

再粗暴点，直接用一个变量就可以了。

```
public int maxSubArray(int[] nums) {
    int n = nums.length;
    int dp = nums[0];
    int max = nums[0];
    for (int i = 1; i < n; i++) {
        if (dp < 0) {
            dp = nums[i];
        } else {
            dp= dp + nums[i];
        }
        max = Math.max(max, dp);
    }
    return max;
```

```
}
```

而对于

```
if (dp < 0) {
    dp = nums[i];
} else {
    dp= dp + nums[i];
}
```

其实也可以这样理解，

```
dp= Math.max(dp + nums[i],nums[i]);
```

然后就变成了这里提到的算法。

解法三 折半

题目最后说

> If you have figured out the O(*n*) solution, try coding another solution using the divide and conquer approach, which is more subtle.

这里找到了种解法，分享下。

假设我们有了一个函数 int getSubMax(int start, int end, int[] nums) ， 可以得到 num [start, end) （左包右不包）中子数组最大值。

如果， start == end，那么 getSubMax 直接返回 nums [start] 就可以了。

```
if (start == end) {
    return nums[start];
}
```

然后对问题进行分解。

先找一个 mid ， mid = (start + end) / 2。

然后，对于我们要找的和最大的子数组有两种情况。

- mid 不在我们要找的子数组中

  这样的话，子数组的最大值要么是 mid 左半部分数组的子数组产生，要么是右边的产生，最大值的可以利用 getSubMax 求出来。

  ```
  int leftMax = getSubMax(start, mid, nums);
  int rightMax = getSubMax(mid + 1, end, nums);
  ```

- mid 在我们要找的子数组中

  这样的话，我们可以分别从 mid 左边扩展，和右边扩展，找出两边和最大的时候，然后加起来就可以了。当然如果，左边或者右边最大的都小于 0 ，我们就不加了。

```
int containsMidMax = getContainMidMax(start, end, mid, nums);
private int getContainMidMax(int start, int end, int mid, int[] nums) {
    int containsMidLeftMax = 0; //初始化为 0 ，防止最大的值也小于 0
    //找左边最大
    if (mid > 0) {
        int sum = 0;
        for (int i = mid - 1; i >= 0; i--) {
            sum += nums[i];
            if (sum > containsMidLeftMax) {
                containsMidLeftMax = sum;
            }
        }

    }
    int containsMidRightMax = 0;
    //找右边最大
    if (mid < end) {
        int sum = 0;
        for (int i = mid + 1; i <= end; i++) {
            sum += nums[i];
            if (sum > containsMidRightMax) {
                containsMidRightMax = sum;
            }
        }
    }
    return containsMidLeftMax + nums[mid] + containsMidRightMax;
}
```

  最后，我们只需要返回这三个中最大的值就可以了。

综上，递归出口，问题分解就都有了。

```
public int maxSubArray(int[] nums) {
    return getSubMax(0, nums.length - 1, nums);
}

private int getSubMax(int start, int end, int[] nums) {
    //递归出口
    if (start == end) {
        return nums[start];
    }
```

```
    int mid = (start + end) / 2;
    //要找的数组不包含 mid，然后得到左边和右边最大的值
    int leftMax = getSubMax(start, mid, nums);
    int rightMax = getSubMax(mid + 1, end, nums);
    //要找的数组包含 mid
    int containsMidMax = getContainMidMax(start, end, mid, nums);
    //返回它们 3 个中最大的
    return Math.max(containsMidMax, Math.max(leftMax, rightMax));
}

private int getContainMidMax(int start, int end, int mid, int[] nums) {
    int containsMidLeftMax = 0; //初始化为 0 ，防止最大的值也小于 0
    //找左边最大
    if (mid > 0) {
        int sum = 0;
        for (int i = mid - 1; i >= 0; i--) {
            sum += nums[i];
            if (sum > containsMidLeftMax) {
                containsMidLeftMax = sum;
            }
        }

    }
    int containsMidRightMax = 0;
    //找右边最大
    if (mid < end) {
        int sum = 0;
        for (int i = mid + 1; i <= end; i++) {
            sum += nums[i];
            if (sum > containsMidRightMax) {
                containsMidRightMax = sum;
            }
        }
    }
    return containsMidLeftMax + nums[mid] + containsMidRightMax;
}
```

时间复杂度：O（n log (n) ）。由于 getContainMidMax 这个函数耗费了 O（n）。所以时间复杂度反而相比之前的算法变大了。

空间复杂度：

总

解法一和解法二的动态规划，只是在定义的时候一个表示以 i 开头的子数组，一个表示以 i 结尾的子数组，却造成了时间复杂度的差异。问题就是解法一中求出了太多的没必要的和，不如解法二直接，只保存最大的和。解法三，一半一半的求，从而使问题分解，也是经常遇到的思想。

54、题目描述（中等难度）

Given a matrix of *m* x *n* elements (*m* rows, *n* columns), return all elements of the matrix in spiral order.

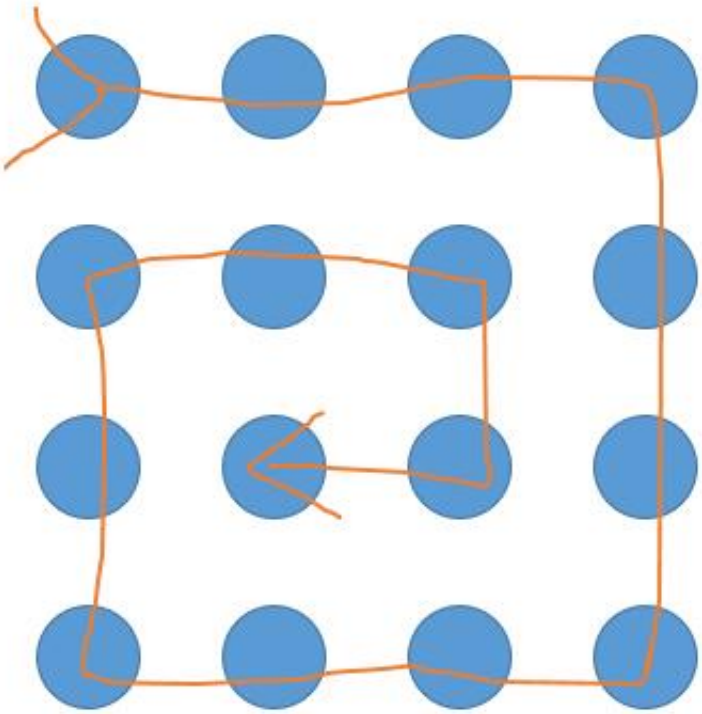
Example 1:

```
Input:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
Output: [1,2,3,6,9,8,7,4,5]
```

Example 2:

```
Input:
[
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9,10,11,12]
]
Output: [1,2,3,4,8,12,11,10,9,5,6,7]
```

从第一个位置开始，螺旋状遍历二维矩阵。



解法一

可以理解成贪吃蛇，从第一个位置开始沿着边界走，遇到边界就转换方向接着走，直到走完所有位置。

```
/*
 * direction 0 代表向右, 1 代表向下, 2 代表向左, 3 代表向上
*/
public List<Integer> spiralOrder(int[][] matrix) {
    List<Integer> ans = new ArrayList<>();
    if(matrix.length == 0){
        return ans;
    }
    int start_x = 0,
    start_y = 0,
    direction = 0,
    top_border = -1,  //上边界
    right_border = matrix[0].length,  //右边界
    bottom_border = matrix.length, //下边界
    left_border = -1; //左边界
    while(true){
        //全部遍历完结束
        if (ans.size() == matrix.length * matrix[0].length) {
```

```
            return ans;
        }
    //注意 y 方向写在前边，x 方向写在后边
        ans.add(matrix[start_y][start_x]);
        switch (direction) {
            //当前向右
            case 0:
                //继续向右是否到达边界
                //到达边界就改变方向，并且更新上边界
                if (start_x + 1 == right_border) {
                    direction = 1;
                    start_y += 1;
                    top_border += 1;
                } else {
                    start_x += 1;
                }
                break;
            //当前向下
            case 1:
                //继续向下是否到达边界
                //到达边界就改变方向，并且更新右边界
                if (start_y + 1 == bottom_border) {
                    direction = 2;
                    start_x -= 1;
                    right_border -= 1;
                } else {
                    start_y += 1;
                }
                break;
            case 2:
                if (start_x - 1 == left_border) {
                    direction = 3;
                    start_y -= 1;
                    bottom_border -= 1;
                } else {
                    start_x -= 1;
                }
                break;
            case 3:
                if (start_y - 1 == top_border) {
                    direction = 0;
                    start_x += 1;
                    left_border += 1;
                } else {
                    start_y -= 1;
                }
```

```
                    break;
                }
            }

        }
```

时间复杂度：O（m * n），m 和 n 是数组的长宽。

空间复杂度：O（1）。

总

在 leetcode 的 solution 和 discuss 看了下，基本就是这个思路了，只是实现上有些不同，怎么用来标记是否走过，当前方向，怎么遍历，实现有些不同，但本质上是一样的。就是充分理解题意，然后模仿遍历的过程。

55、题目描述（中等难度）

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

Example 1:

```
Input: [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

Example 2:

```
Input: [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter what. Its maximum
             jump length is 0, which makes it impossible to reach the last index.
```

45题的时候已经见过这道题了，只不过之前是返回从第 0 个位置能跳到最后一个位置的最小步数，这道题是返回是否能跳过去。

leetCode Solution 中给出的是动态规划的解法，进行了一步一步的优化，但都也比较慢。不过，思路还是值得参考的，上边说的比较详细，这里就不啰嗦了。这里，由于受到 45 题的影响，自己对 45 题的解法改写了一下，从而解决了这个问题。

下边的解法都是基于45题 的想法，大家可以先过去看一下，懂了之后再回到下边来看。

解法一 顺藤摸瓜

45 题的代码。

```
public int jump(int[] nums) {
    int end = 0;
    int maxPosition = 0;
    int steps = 0;
    for(int i = 0; i < nums.length - 1; i++){
        //找能跳的最远的
        maxPosition = Math.max(maxPosition, nums[i] + i);
        if( i == end){ //遇到边界，就更新边界，并且步数加一
            end = maxPosition;
            steps++;
        }
    }
    return steps;
}
```

这里的话，我们完全可以把 step 去掉，并且考虑下当前更新的 i 是不是已经超过了边界。

```
public boolean canJump(int[] nums) {
    int end = 0;
    int maxPosition = 0;
    for(int i = 0; i < nums.length - 1; i++){
        //当前更新超过了边界，那么意味着出现了 0 ，直接返回 false
        if(end < i){
            return false;
        }
        //找能跳的最远的
        maxPosition = Math.max(maxPosition, nums[i] + i);

        if( i == end){ //遇到边界，就更新边界，并且步数加一
            end = maxPosition;
        }
    }
    //最远的距离是否到答末尾
    return maxPosition>=nums.length-1;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

解法二 顺瓜摸藤

每次找最左边的能跳到当前位置的下标，之前的代码如下。

```
public int jump(int[] nums) {
```

```
    int position = nums.length - 1; //要找的位置
    int steps = 0;
    while (position != 0) { //是否到了第 0 个位置
        for (int i = 0; i < position; i++) {
            if (nums[i] >= position - i) {
                position = i; //更新要找的位置
                steps++;
                break;
            }
        }
    }
    return steps;
}
```

这里修改的话，只需要判断最后回没回到 0 ，并且如果 while 里的 for 循环没有进入 if ，就意味着一个位置都没找到，就要返回 false。

```
public boolean canJump(int[] nums) {
    int position = nums.length - 1; //要找的位置
    boolean isUpdate = false;
    while (position != 0) { //是否到了第 0 个位置
        isUpdate = false;
        for (int i = 0; i < position; i++) {
            if (nums[i] >= position - i) {
                position = i; //更新要找的位置
                isUpdate = true;
                break;
            }
        }
        //如果没有进入 for 循环中的 if 语句，就返回 false
        if(!isUpdate){
            return false;
        }
    }
    return true;
}
```

时间复杂度：O（n^2）。

空间复杂度：O（1）。

解法三

让我们直击问题的本质，与 45 题不同，我们并不需要知道最小的步数，所以我们对跳的过程并不感兴趣。并且如果数组里边没有 0，那么无论怎么跳，一定可以从第 0 个跳到最后一个位置。

所以我们只需要看 0 的位置，如果有 0 的话，我们只需要看 0 前边的位置，能不能跳过当前的 0 ，如果 0 前边的位置都不能跳过当前 0，那么直接返回 false。如果能的话，就看后边的 0 的情况。

```
public boolean canJump(int[] nums) {
    for (int i = 0; i < nums.length - 1; i++) {
        //找到 0 的位置
        if (nums[i] == 0) {
            int j = i - 1;
            boolean isCanSkipZero = false;
            while (j >= 0) {
                //判断 0 前边的元素能否跳过 0
                if (j + nums[j] > i) {
                    isCanSkipZero = true;
                    break;
                }
                j--;
            }
            if (!isCanSkipZero) {
                return false;
            }
        }
    }
    return true;
}
```

但这样时间复杂度没有提高， 在 @Zhengwen 的提醒下，可以用下边的方法。

我们判断 0 前边的元素能否跳过 0 ，不需要每次都向前查找，我们只需要用一个变量保存当前能跳的最远的距离，然后判断最远距离和当前 0 的位置就可以了。

```
public boolean canJump(int[] nums) {
    int max = 0;
    for (int i = 0; i < nums.length - 1; i++) {
        if (nums[i] == 0 && i >= max) {
            return false;
        }
        max = Math.max(max, nums[i] + i);
    }
    return true;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

参考这里，我们甚至不需要考虑 0 的位置，只需要判断最大距离有没有超过当前的 i 。

```
public boolean canJump(int[] nums) {
    int max = 0;
    for (int i = 0; i < nums.length; i++) {
        if (i > max) {
            return false;
        }
        max = Math.max(max, nums[i] + i);
    }
    return true;
}
```

总

当自己按照 45 题的思路写完的时候，看 Solution 的时候都懵逼了，这道题竟然这么复杂？不过 Solution 把问题抽象成动态规划的思想，以及优化的过程还是非常值得学习的。

56、题目描述（中等难度）

56. Merge Intervals

Medium 1826 139 Favorite Share

Given a collection of intervals, merge all overlapping intervals.

Example 1:

```
Input: [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
```

Example 2:

```
Input: [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

给定一个列表，将有重叠部分的合并。例如[[1 3] [2 6]] 合并成 [1 6] 。

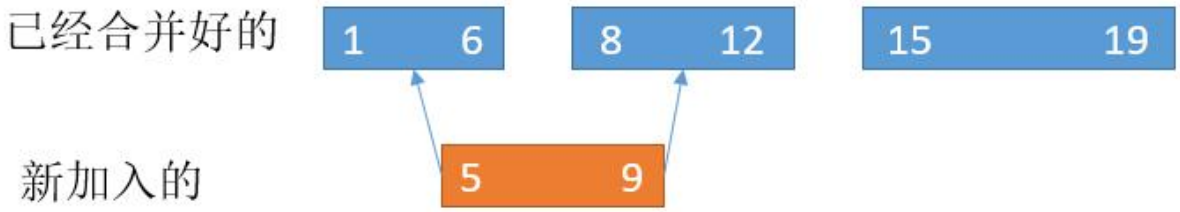
解法一

常规的思想，将大问题化解成小问题去解决。

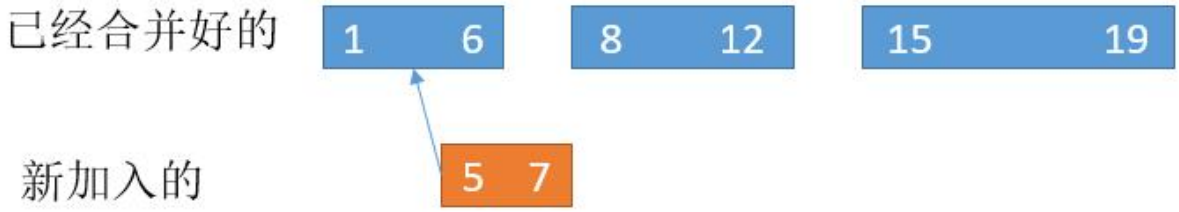
假设给了一个大小为 n 的列表，然后我们假设 n - 1 个元素的列表已经完成了全部合并，我们现在要解决的就是剩下的 1 个，怎么加到已经合并完的 n -1 个元素中。

这样的话分下边几种情况， 我们把每个范围叫做一个节点，节点包括左端点和右端点。

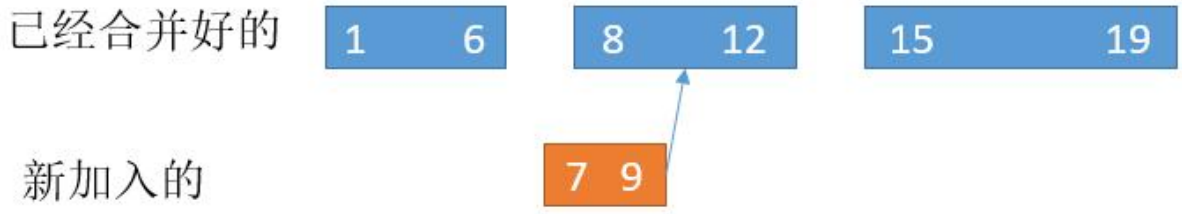
1. 如下图，新加入的节点左端点和右端点，分别在两个节点之间。这样，我们只要删除这两个节点，并且使用左边节点的左端点，右边的节点的右端点作为一个新节点插入即可。也就是删除 [1 6] 和 [8 12] ，加入 [1 12] 到合并好的列表中。



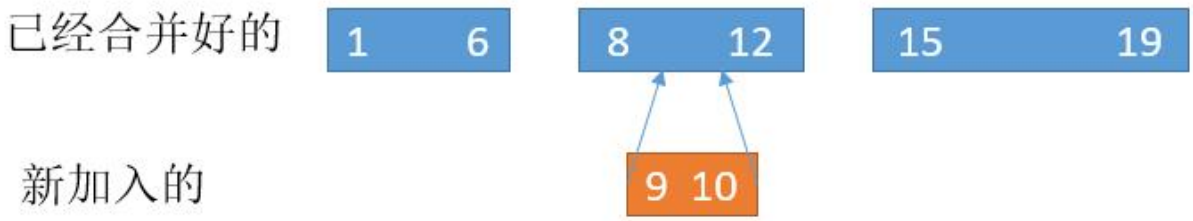
2. 如下图，新加入的节点只有左端点在之前的一个节点之内，这样的话将这个节点删除，使用删除的节点的左端点，新加入的节点的右端点，作为新的节点插入即可。也就是删除 [1 6]，加入 [1 7] 到合并好的列表中。



3. 如下图，新加入的节点只有右端点在之前的一个节点之内，这样的话将这个节点删除，使用删除的节点的右端点，新加入的节点的左端点，作为新的节点插入即可。也就是删除 [8 12]，加入 [7 12] 到合并好的列表中。

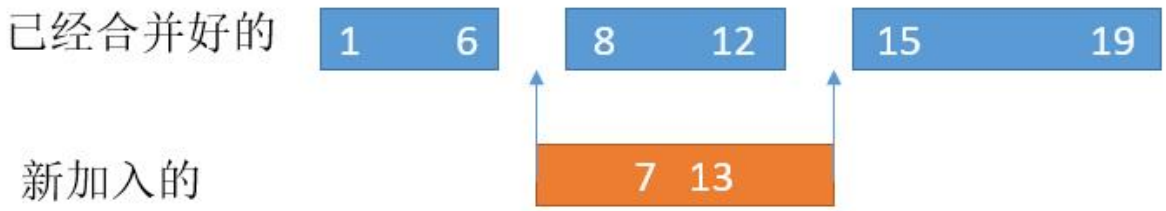


4. 如下图，新加入的节点的左端点和右端点在之前的一个节点之内，这样的话新加入的节点舍弃就可以了。



5. 如下图，新加入的节点没有在任何一个节点之内，那么将它直接作为新的节点加入到合并好的节点之内就可以了。

6. 如下图，还有一种情况，就是新加入的节点两个端点，将之前的节点囊括其中，这种的话，我们只需要将囊括的节点删除，把新节点加入即可。把 [8 12] 删除，将 7 13 加入即可。并且，新加入的节点可能会囊括多个旧节点，比如新加入的节点是 [1 100]，那么下边的三个节点就都包括了，就需要都删除掉。



以上就是所有的情况了，可以开始写代码了。

```
public class Interval {
    int start;
    int end;

    Interval() {
        start = 0;
        end = 0;
    }

    Interval(int s, int e) {
        start = s;
        end = e;
    }
```

```
}

public List<Interval> merge(List<Interval> intervals) {
    List<Interval> ans = new ArrayList<>();
    if (intervals.size() == 0)
        return ans;
    //将第一个节点加入，作为合并好的节点列表
    ans.add(new Interval(intervals.get(0).start, intervals.get(0).end));
    //遍历其他的每一个节点
    for (int i = 1; i < intervals.size(); i++) {
        Interval start = null;
        Interval end = null;
        //新加入节点的左端点
        int i_start = intervals.get(i).start;
        //新加入节点的右端点
        int i_end = intervals.get(i).end;
        int size = ans.size();
        //情况 6，保存囊括的节点，便于删除
        List<Interval> in = new ArrayList<>();
        //遍历合并好的每一个节点
        for (int j = 0; j < size; j++) {
            //找到左端点在哪个节点内
            if (i_start >= ans.get(j).start && i_start <= ans.get(j).end) {
                start = ans.get(j);
            }
            //找到右端点在哪个节点内
            if (i_end >= ans.get(j).start && i_end <= ans.get(j).end) {
                end = ans.get(j);
            }
            //判断新加入的节点是否囊括当前旧节点，对应情况 6
            if (i_start < ans.get(j).start && i_end >ans.get(j).end) {
                in.add(ans.get(j));
            }

        }
        //删除囊括的节点
        if (in.size() != 0) {
            for (int index = 0; index < in.size(); index++) {
                ans.remove(in.get(index));
            }
        }
        //equals 函数作用是在 start 和 end 有且只有一个 null，或者 start 和 end 是同一个
节点返回 true，相当于情况 2 3 4 中的一种
        if (equals(start, end)) {
            //如果 start 和 end 都不等于 null 就代表情况 4
```

```
            // start 等于 null 的话相当于情况 3
            int s = start == null ? i_start : start.start;
            // end 等于 null 的话相当于情况 2
            int e = end == null ? i_end : end.end;
            ans.add(new Interval(s, e));
            // start 和 end 不是同一个节点，相当于情况 1
        } else if (start!= null && end!=null) {
            ans.add(new Interval(start.start, end.end));
            // start 和 end 都为 null，相当于情况 5 和 情况 6 ，加入新节点
        }else if (start == null) {
            ans.add(new Interval(i_start, i_end));
        }
        //将旧节点删除
        if (start != null) {
            ans.remove(start);
        }
        if (end != null) {
            ans.remove(end);
        }

    }
    return ans;
}

private boolean equals(Interval start, Interval end) {
    if (start == null && end == null) {
        return false;
    }
    if (start == null || end == null) {
        return true;
    }
    if (start.start == end.start && start.end == end.end) {
        return true;
    }
    return false;
}
```

时间复杂度：O（n^2）。

空间复杂度：O （n），用来存储结果。

解法二

参考这里的解法二。

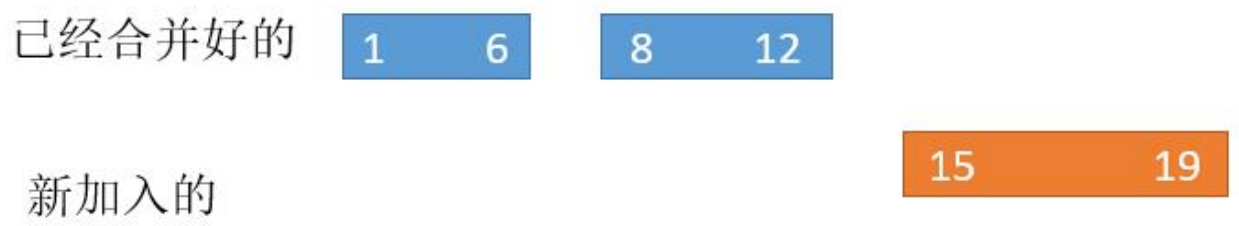
在解法一中，我们每次对于新加入的节点，都用一个 for 循环去遍历已经合并好的列表。如果我们把之前的列表，按照左端点进行从小到大排序了。这样的话，每次添加新节点的话，我们只需要和合并好的列表最后一个节点对比就可以了。

```
[(1, 9), (2, 5), (19, 20), (10, 11), (12, 20), (0, 3), (0, 1), (0, 2)]
```

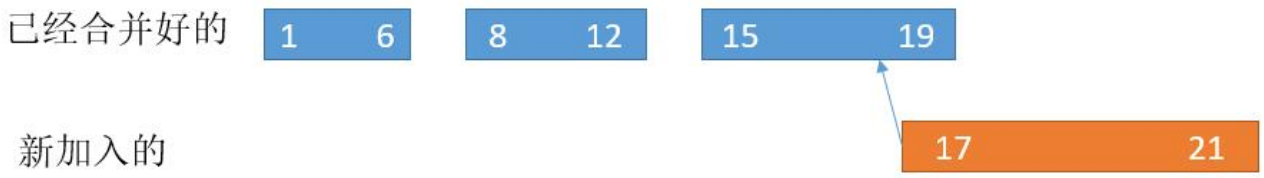

```
[(0, 3), (0, 1), (0, 2), (1, 9), (2, 5), (10, 11), (12, 20), (19, 20)]
```

排好序后我们只需要把新加入的节点和最后一个节点比较就够了。

情况 1，如果新加入的节点的左端点大于合并好的节点列表的最后一个节点的右端点，那么我们只需要把新节点直接加入就可以了。



情况 2，如果新加入的节点的左端点不大于合并好的节点列表的最后一个节点的右端点，那么只需要判断新加入的节点的右端点和最后一个节点的右端点哪个大，然后更新最后一个节点的右端点就可以了。




```
private class IntervalComparator implements Comparator<Interval> {
    @Override
    public int compare(Interval a, Interval b) {
        return a.start < b.start ? -1 : a.start == b.start ? 0 : 1;
    }
}

public List<Interval> merge(List<Interval> intervals) {
    Collections.sort(intervals, new IntervalComparator());

    LinkedList<Interval> merged = new LinkedList<Interval>();
```

```
    for (Interval interval : intervals) {
        //最开始是空的，直接加入
        //然后对应情况 1，新加入的节点的左端点大于最后一个节点的右端点
        if (merged.isEmpty() || merged.getLast().end < interval.start) {
            merged.add(interval);
        }
        //对于情况 2 ，更新最后一个节点的右端点
        else {
            merged.getLast().end = Math.max(merged.getLast().end, interval.end);
        }
    }

    return merged;
}
```

时间复杂度：O（n log（n））， 排序算法。

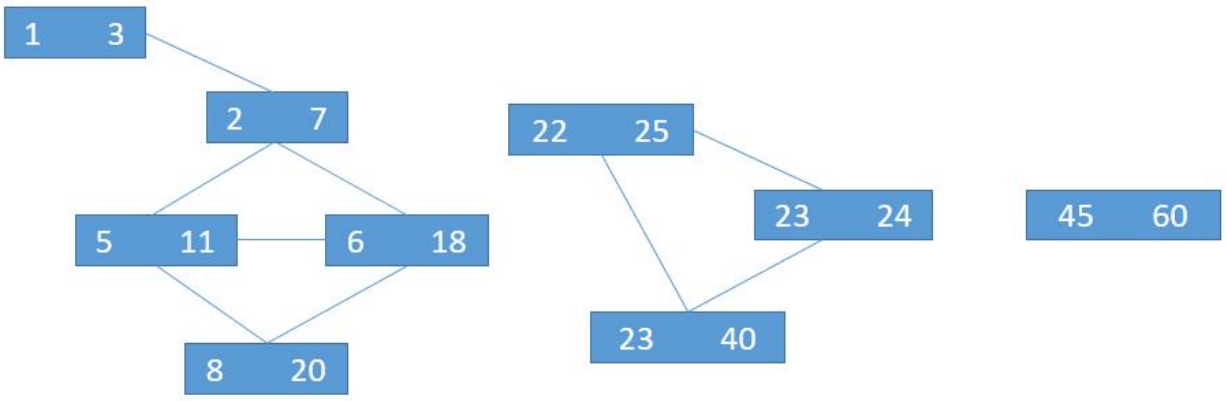
空间复杂度：O（n），存储结果。另外排序算法也可能需要。

解法三

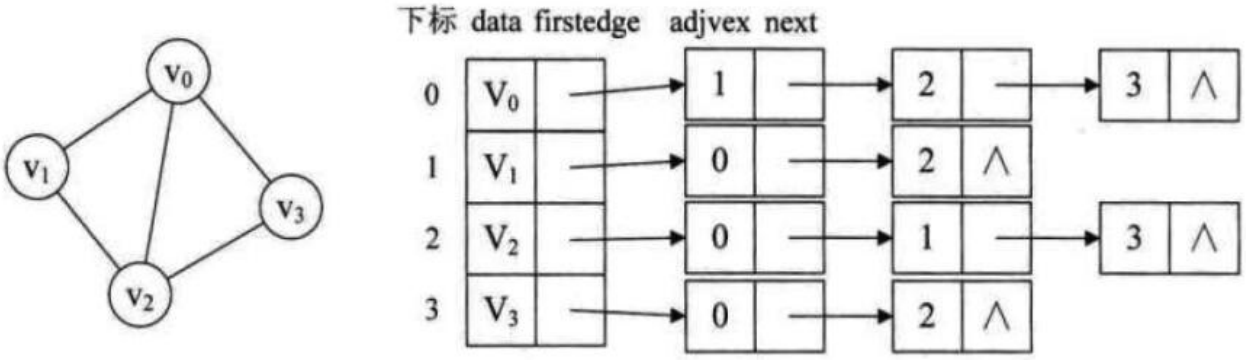
参考这里的解法 1。

刷这么多题，第一次利用图去解决问题，这里分享下作者的思路。

如果每个节点如果有重叠部分，就用一条边相连。



我们用一个 HashMap，用邻接表的结构来实现图，类似于下边的样子。



图存起来以后，可以发现，最后有几个连通图，最后合并后的列表就有几个。我们需要把每个连通图保存起来，然后在每个连通图中找最小和最大的端点作为一个节点加入到合并后的列表中就可以了。最后，我们把每个连通图就转换成下边的图了。


```
class Solution {
    private Map<Interval, List<Interval> > graph; //存储图
    private Map<Integer, List<Interval> > nodesInComp; ///存储每个有向图
    private Set<Interval> visited;

   //主函数
    public List<Interval> merge(List<Interval> intervals) {
        buildGraph(intervals); //建立图
        buildComponents(intervals); //单独保存每个有向图


        List<Interval> merged = new LinkedList<>();
        //遍历每个有向图，将有向图中最小最大的节点加入到列表中
        for (int comp = 0; comp < nodesInComp.size(); comp++) {
            merged.add(mergeNodes(nodesInComp.get(comp)));
        }

        return merged;
    }

    // 判断两个节点是否有重叠部分
    private boolean overlap(Interval a, Interval b) {
        return a.start <= b.end && b.start <= a.end;
    }
```

```
    //利用邻接表建立图
    private void buildGraph(List<Interval> intervals) {
        graph = new HashMap<>();
        for (Interval interval : intervals) {
            graph.put(interval, new LinkedList<>());
        }

        for (Interval interval1 : intervals) {
            for (Interval interval2 : intervals) {
                if (overlap(interval1, interval2)) {
                    graph.get(interval1).add(interval2);
                    graph.get(interval2).add(interval1);
                }
            }
        }
    }

    // 将每个连接图单独存起来
    private void buildComponents(List<Interval> intervals) {
        nodesInComp = new HashMap();
        visited = new HashSet();
        int compNumber = 0;

        for (Interval interval : intervals) {
            if (!visited.contains(interval)) {
                markComponentDFS(interval, compNumber);
                compNumber++;
            }
        }
    }

    //利用深度优先遍历去找到所有互相相连的边
    private void markComponentDFS(Interval start, int compNumber) {
        Stack<Interval> stack = new Stack<>();
        stack.add(start);

        while (!stack.isEmpty()) {
            Interval node = stack.pop();
            if (!visited.contains(node)) {
                visited.add(node);

                if (nodesInComp.get(compNumber) == null) {
                    nodesInComp.put(compNumber, new LinkedList<>());
                }
                nodesInComp.get(compNumber).add(node);
```

```
                for (Interval child : graph.get(node)) {
                    stack.add(child);
                }
            }
        }
    }


    // 找出每个有向图中最小和最大的端点
    private Interval mergeNodes(List<Interval> nodes) {
        int minStart = nodes.get(0).start;
        for (Interval node : nodes) {
            minStart = Math.min(minStart, node.start);
        }

        int maxEnd = nodes.get(0).end;
        for (Interval node : nodes) {
            maxEnd= Math.max(maxEnd, node.end);
        }

        return new Interval(minStart, maxEnd);
    }
}
```

时间复杂度：

空间复杂度：O（n^2），最坏的情况，每个节点都互相重合，这样每个都与其他节点相连，就会是 n^2 的空间存储图。

可惜的是，这种解法在 leetcode 会遇到超时错误。

总

开始的时候，使用最常用的思路，将大问题化解为小问题，然后用递归或者直接用迭代实现。解法二中，先对列表进行排序，从而优化了时间复杂度，也不是第一次看到了。解法三中，利用图解决问题很新颖，是我刷题第一次遇到的，又多了一种解题思路。

57、题目描述（困难难度）

57. Insert Interval

Hard 726 92 Favorite Share

Given a set of *non-overlapping* intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Example 1:

```
Input: intervals = [[1,3],[6,9]], newInterval = [2,5]
Output: [[1,5],[6,9]]
```

Example 2:

```
Input: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
Output: [[1,2],[3,10],[12,16]]
Explanation: Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10].
```

和上一道可以说是一个问题，只不过这个是给一个已经合并好的列表，然后给一个新的节点依据规则加入到合并好的列表。

解法一

对应 56 题的解法一，没看的话，可以先过去看一下。这个问题其实就是我们解法中的一个子问题，所以直接加过来就行了。

```
public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
      Interval start = null;
      Interval end = null;
      int i_start = newInterval.start;
      int i_end =newInterval.end;
      int size = intervals.size();
      List<Interval> in = new ArrayList<>();
            //遍历合并好的列表
      for (int j = 0; j < size; j++) {
        if (i_start >= intervals.get(j).start && i_start <= intervals.get(j).end)
{
          start = intervals.get(j);
        }
        if (i_end >= intervals.get(j).start && i_end <= intervals.get(j).end) {
          end = intervals.get(j);
        }
        if (i_start < intervals.get(j).start && i_end >intervals.get(j).end) {
          in.add(intervals.get(j));
        }

      }
      if (in.size() != 0) {
```

```
        for (int index = 0; index < in.size(); index++) {
          intervals.remove(in.get(index));
        }
      }
      Interval interval = null;
        //根据不同的情况，生成新的节点
      if (equals(start, end)) {
        int s = start == null ? i_start : start.start;
        int e = end == null ? i_end : end.end;
        interval = new Interval(s, e);
      } else if (start!= null && end!=null) {
        interval = new Interval(start.start, end.end);
      }else if (start == null) {
        interval = new Interval(i_start, i_end);
      }
      if (start != null) {
        intervals.remove(start);
      }
      if (end != null) {
        intervals.remove(end);
      }
           //不同之处是生成的节点，要遍历原节点，根据 start 插入到对应的位置，虽然题目没说，但这里如果不按顺序插入的话，leetcode 过不了。
      for(int i = 0;i<intervals.size();i++){
        if(intervals.get(i).start>interval.start){
          intervals.add(i,interval);
          return intervals;
        }
      }
      intervals.add(interval);
      return intervals;
      }
private boolean equals(Interval start, Interval end) {
    if (start == null && end == null) {
        return false;
    }
    if (start == null || end == null) {
        return true;
    }
    if (start.start == end.start && start.end == end.end) {
        return true;
    }
    return false;
}
```

时间复杂度：O（n）。

空间复杂度： O（n），里边的 in 变量用来存储囊括的节点时候耗费的。

我们其实可以利用迭代器，一边遍历，一边删除，这样就不需要 in 变量了。

```
public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
    Interval start = null;
    Interval end = null;
    int i_start = newInterval.start;
    int i_end = newInterval.end;
    //利用迭代器遍历
    for (Iterator<Interval> it = intervals.iterator(); it.hasNext();) {
      Interval inter = it.next();
      if (i_start >= inter.start && i_start <= inter.end) {
        start = inter;
      }
      if (i_end >= inter.start && i_end <= inter.end) {
        end = inter;
      }
      if (i_start < inter.start && i_end > inter.end) {

        it.remove();
      }
    }
    Interval interval = null;
    if (equals(start, end)) {
      int s = start == null ? i_start : start.start;
      int e = end == null ? i_end : end.end;
      interval = new Interval(s, e);
    } else if (start != null && end != null) {
      interval = new Interval(start.start, end.end);
    } else if (start == null) {
      interval = new Interval(i_start, i_end);
    }
    if (start != null) {
      intervals.remove(start);
    }
    if (end != null) {
      intervals.remove(end);
    }
    for (int i = 0; i < intervals.size(); i++) {
      if (intervals.get(i).start > interval.start) {
        intervals.add(i, interval);
        return intervals;
      }
    }
    intervals.add(interval);
```

```
    return intervals;
  }

private boolean equals(Interval start, Interval end) {
    if (start == null && end == null) {
        return false;
    }
    if (start == null || end == null) {
        return true;
    }
    if (start.start == end.start && start.end == end.end) {
        return true;
    }
    return false;
}
```

时间复杂度：O（n）。

空间复杂度： O（1）。

解法二

对应 56 题的解法二，考虑到它给定的合并的列表是有序的，和解法二是一个思想。只不过这里不能直接从末尾添加，而是根据新节点的 start 来找到它应该在的位置，然后再利用之前的想法就够了。

这里把 leetcode 里的两种写法，贴过来，大家可以参考一下。

第一种。

```
public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
    List<Interval> result = new LinkedList<>();
    int i = 0;
    // 将新节点之前的节点加到结果中
    while (i < intervals.size() && intervals.get(i).end < newInterval.start)
        result.add(intervals.get(i++));
    // 和新节点判断是否重叠，更新新节点
    while (i < intervals.size() && intervals.get(i).start <= newInterval.end) {
        newInterval = new Interval(
                Math.min(newInterval.start, intervals.get(i).start),
                Math.max(newInterval.end, intervals.get(i).end));
        i++;
    }
    //将新节点加入
    result.add(newInterval);
    ///剩下的全部加进来
    while (i < intervals.size()) result.add(intervals.get(i++));
```

```
    return result;
}
```

第二种。和之前是一样的思想，只不过更加的简洁，可以参考一下。

```
public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
    List<Interval> result = new ArrayList<Interval>();
    for (Interval i : intervals) {
        //新加的入的节点在当前节点后边
        if (newInterval == null || i.end < newInterval.start)
            result.add(i);
        //新加入的节点在当前节点的前边
        else if (i.start > newInterval.end) {
            result.add(newInterval);
            result.add(i);
            newInterval = null;
        //新加入的节点和当前节点有重合，更新节点
        } else {
            newInterval.start = Math.min(newInterval.start, i.start);
            newInterval.end = Math.max(newInterval.end, i.end);
        }
    }
    if (newInterval != null)
        result.add(newInterval);
    return result;
}
```

总的来说，上边两个写法本质是一样的，就是依据他们是有序的，先把新节点前边的节点加入，然后开始判断是否重合，当前节点加入后，把后边的加入就可以了。

时间复杂度：O（n）。

空间复杂度：O（n），存储最后的结果。

总

总的来说，这道题可以看做上道题的一些变形，本质上是一样的。由于用 for 循环不能一边遍历列表，一边删除某个元素，所以利用迭代器实现边遍历，边删除，自己也是第一次用。此外，解法一更加通用些，它不要求给定的列表有序。

58、题目描述（简单难度）

58. Length of Last Word

Easy 340 1408 Favorite Share

Given a string *s* consists of upper/lower-case alphabets and empty space characters `' '`, return the length of last word in the string.

If the last word does not exist, return 0.

Note: A word is defined as a character sequence consists of non-space characters only.

Example:

```
Input: "Hello World"
Output: 5
```

输出最后一个单词的长度。

解法一

直接从最后一个字符往前遍历，遇到空格停止就可以了。不过在此之前要过滤到末尾的空格。

```
public int lengthOfLastWord(String s) {
    int count = 0;
    int index = s.length() - 1;
    //过滤空格
    while (true) {
        if (index < 0 || s.charAt(index) != ' ')
            break;
        index--;
    }
    //计算最后一个单词的长度
    for (int i = index; i >= 0; i--) {
        if (s.charAt(i) == ' ') {
            break;
        }
        count++;
    }
    return count;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

总

时隔多天，又遇到了一个简单的题，没什么好说的，就是遍历一遍，没有 get 到考点。

59、题目描述（中等难度）

59. Spiral Matrix II

Medium 405 74 Favorite Share

Given a positive integer *n*, generate a square matrix filled with elements from 1 to n^2 in spiral order.

Example:

```
Input: 3
Output:
[
 [ 1, 2, 3 ],
 [ 8, 9, 4 ],
 [ 7, 6, 5 ]
]
```

和 54题 差不多，54 题按照螺旋状遍历，这个是按照螺旋状生成二维数组。

解法一

直接按照 54题，贪吃蛇的走法来写，如果没做过可以看一下。

```
/*
   * direction 0 代表向右, 1 代表向下, 2 代表向左, 3 代表向上
   */
public int[][] generateMatrix(int n) {
    int[][] ans = new int[n][n];
    int start_x = 0, start_y = 0, direction = 0, top_border = -1, // 上边界
    right_border = n, // 右边界
    bottom_border = n, // 下边界
    left_border = -1; // 左边界
    int count = 1;
    while (true) {
        // 全部遍历完结束
        if (count == n * n + 1) {
            return ans;
        }
        // 注意 y 方向写在前边，x 方向写在后边
        ans[start_y][start_x] = count;
        count++;
        switch (direction) {
                // 当前向右
            case 0:
                // 继续向右是否到达边界
                // 到达边界就改变方向，并且更新上边界
```

```
                if (start_x + 1 == right_border) {
                    direction = 1;
                    start_y += 1;
                    top_border += 1;
                } else {
                    start_x += 1;
                }
                break;
                // 当前向下
            case 1:
                // 继续向下是否到达边界
                // 到达边界就改变方向，并且更新右边界
                if (start_y + 1 == bottom_border) {
                    direction = 2;
                    start_x -= 1;
                    right_border -= 1;
                } else {
                    start_y += 1;
                }
                break;
            case 2:
                if (start_x - 1 == left_border) {
                    direction = 3;
                    start_y -= 1;
                    bottom_border -= 1;
                } else {
                    start_x -= 1;
                }
                break;
            case 3:
                if (start_y - 1 == top_border) {
                    direction = 0;
                    start_x += 1;
                    left_border += 1;
                } else {
                    start_y -= 1;
                }
                break;
        }
    }

}
```

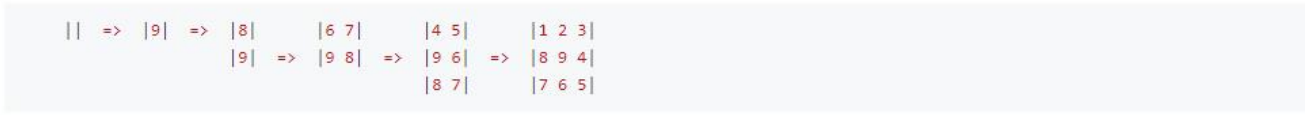
时间复杂度：O（n^2）。

空间复杂度：O（1）。

解法二

这里看到了一个与众不同的想法，分享一下。

Start with the empty matrix, add the numbers in reverse order until we added the number 1. Always rotate the matrix clockwise and add a top row:



矩阵先添加 1 个元素，然后顺时针旋转矩阵，然后再在矩阵第一行添加元素，再顺时针旋转矩阵，再在第一行添加元素，直到变成 n * n 的矩阵。

之前在 48题 做过旋转矩阵的算法，但是当时是 n * n，这个 n * m 就更复杂些了，然后由于 JAVA 的矩阵定义的时候就固定死了，每次添加新的一行又得 new 新的数组，这样整个过程就会很浪费空间，综上，用 JAVA 不适合去实现这个算法，就不实现了，哈哈哈哈哈，看一下作者的 python 代码吧。

总

基本上和 54题 差不多，依旧是理解题意，然后模仿遍历过程就可以了。

60、题目描述（中等难度）

60. Permutation Sequence

Medium 763 208 Favorite Share

The set $[1, 2, 3, \ldots, n]$ contains a total of $n!$ unique permutations.

By listing and labeling all of the permutations in order, we get the following sequence for $n = 3$:

1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

Given n and k, return the k^{th} permutation sequence.

Note:

- Given n will be between 1 and 9 inclusive.
- Given k will be between 1 and $n!$ inclusive.

Example 1:

```
Input: n = 3, k = 3
Output: "213"
```

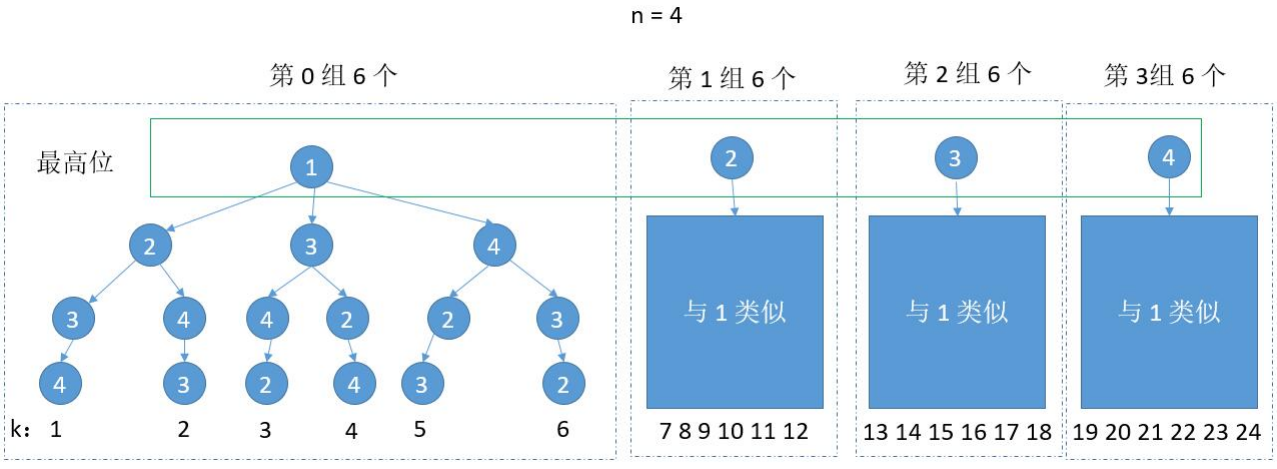
Example 2:

```
Input: n = 4, k = 9
Output: "2314"
```

又是一道全排列的题，之前在31题，46题，也讨论过全排列问题的一些解法。这道题的话，是给一个 n，不是输出它的全排列，而是把所有组合从从小到大排列后，输出第 k 个。

解法一

以 n = 4 为例，可以结合下图看一下。因为是从小到大排列，那么最高位一定是从 1 到 4。然后可以看成一组一组的，我们只需要求出组数，就知道最高位是多少了。而每组的个数就是 n - 1 的阶乘，也就是 3 的阶乘 6。



算组数的时候， 1 到 5 除以 6 是 0，6 除以 6 是 1，而 6 是属于第 0 组的，所有要把 k 减去 1。这样做除法结果就都是 0 了。

```
int perGroupNum = factorial(n - 1);
int groupNum = (k - 1) / perGroupNum;
```

当然，还有一个问题下次 k 是多少了。求组数用的除法，余数就是下次的 k 了。因为 k 是从 1 计数的，所以如果 k 刚好等于了 perGroupNum 的倍数，此时得到的余数是 0 ，而其实由于我们求 groupNum 的时候减 1 了，所以此时 k 应该更新为 perGroupNum。

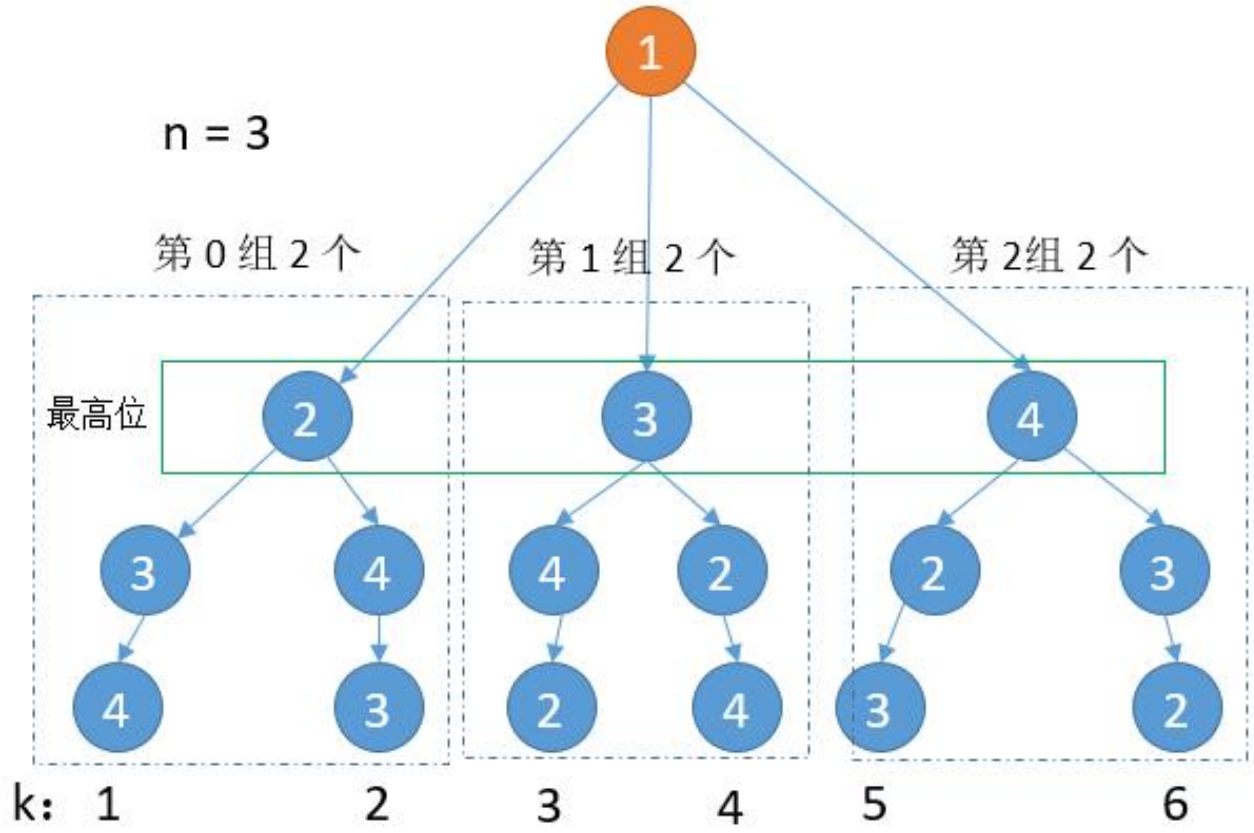
```
k = k % perGroupNum;
k = k == 0 ? perGroupNum : k;
```

举个例子，如果 k = 6，那么 groupNum = (k - 1) / 6 = 0， k % perGroupNum = 6 % 6 = 0，而下次的 k ，可以结合上图，很明显是 perGroupNum ，依旧是 6。

结合下图，确定了最高位属于第 0 组，下边就和上边的情况一样了。唯一不同的地方是最高位是 2 3 4，没有了 1。所有得到 groupNum 怎么得到最高位需要考虑下。

我们可以用一个 list 从小到大保存 1 到 n，每次选到一个就去掉，这样就可以得到 groupNum 对应的数字了。

```
List<Integer> nums = new ArrayList<Integer>();
for (int i = 0; i < n; i++) {
    nums.add(i + 1);
}
int perGroupNum = factorial(n - 1);
int groupNum = (k - 1) / perGroupNum;
int num = nums.get(groupNum); //根据 groupNum 得到当前位
nums.remove(groupNum);//去掉当前数字
```



综上，我们把它们整合在一起。

```java
public String getPermutation(int n, int k) {
    List<Integer> nums = new ArrayList<Integer>();
    for (int i = 0; i < n; i++) {
        nums.add(i + 1);
    }
    return getAns(nums, n, k);
}

private String getAns(List<Integer> nums, int n, int k) {
    if (n == 1) {
        //把剩下的最后一个数字返回就可以了
        return nums.get(0) + "";
    }
    int perGroupNum = factorial(n - 1); //每组的个数
    int groupNum = (k - 1) / perGroupNum;
    int num = nums.get(groupNum);
    nums.remove(groupNum);
    k = k % perGroupNum; //更新下次的 k
    k = k == 0 ? perGroupNum : k;
    return num + getAns(nums, n - 1, k);
}
```

```
public int factorial(int number) {
    if (number <= 1)
        return 1;
    else
        return number * factorial(number - 1);
}
```

时间复杂度：

空间复杂度：

这是最开始自己的想法，有 3 点可以改进一下。

第 1 点，更新 k 的时候，有一句

```
k = k % perGroupNum; //更新下次的 k
k = k == 0 ? perGroupNum : k;
```

很不优雅了，问题的根源就在于问题给定的 k 是从 1 编码的。我们只要把 k - 1 % perGroupNum，这样得到的结果就是 k 从 0 编码的了。然后求 groupNum = (k - 1) / perGroupNum; 这里 k 也不用减 1 了。

第 2 点，这个算法很容易改成改成迭代的写法，只需要把递归的函数参数， 在每次迭代更新就够了。

第 3 点，我们求 perGroupNum 的时候，每次都调用了求迭代的函数，其实没有必要的，我们只需要一次循环求出 n 的阶乘。然后在每次迭代中除以 nums 的剩余个数就够了。

综上，看一下优化过的代码吧。

```
public String getPermutation(int n, int k) {
    List<Integer> nums = new ArrayList<Integer>();
    int factorial = 1;
    for (int i = 0; i < n; i++) {
        nums.add(i + 1);
        if (i != 0) {
            factorial *= i;
        }
    }
    factorial *= n; //先求出 n 的阶乘
    StringBuilder ans = new StringBuilder();
    k = k - 1; // k 变为 k - 1
    for (int i = n; i > 0; i--) {
        factorial /= (nums.size()); //更新为 n - 1 的阶乘
        int groupNum = k / factorial;
        int num = nums.get(groupNum);
        nums.remove(groupNum);
        k = k % factorial;
        ans.append(num);
```

```
    }
    return ans.toString();
}
```

时间复杂度：O（n），当然如果 remove 函数的时间是复杂度是 O（n），那么整体上就是 O（n^2）。

空间复杂度：O（1）。

总

这道题其实如果写出来，也不算难，优化的思路可以了解一下。

61、题目描述（中等难度）

61. Rotate List

Medium 555 794 Favorite Share

Given a linked list, rotate the list to the right by *k* places, where *k* is non-negative.

Example 1:

```
Input: 1->2->3->4->5->NULL, k = 2
Output: 4->5->1->2->3->NULL
Explanation:
rotate 1 steps to the right: 5->1->2->3->4->NULL
rotate 2 steps to the right: 4->5->1->2->3->NULL
```

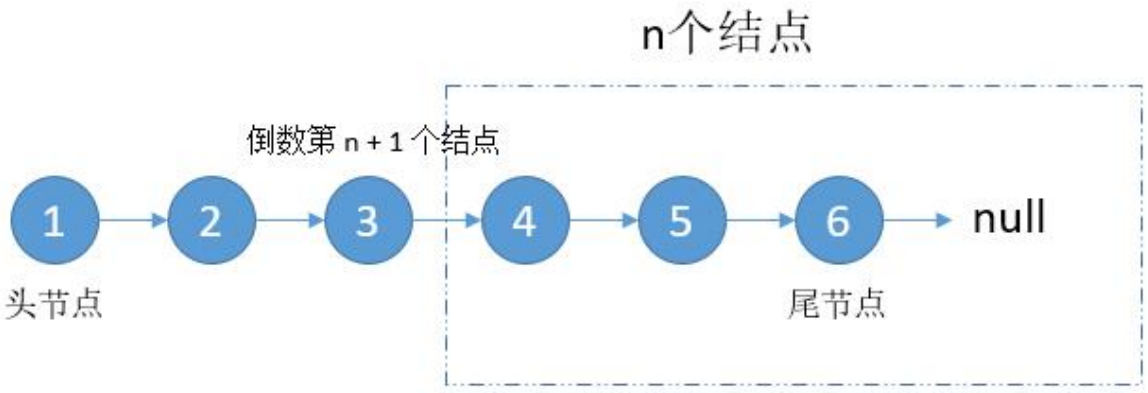
Example 2:

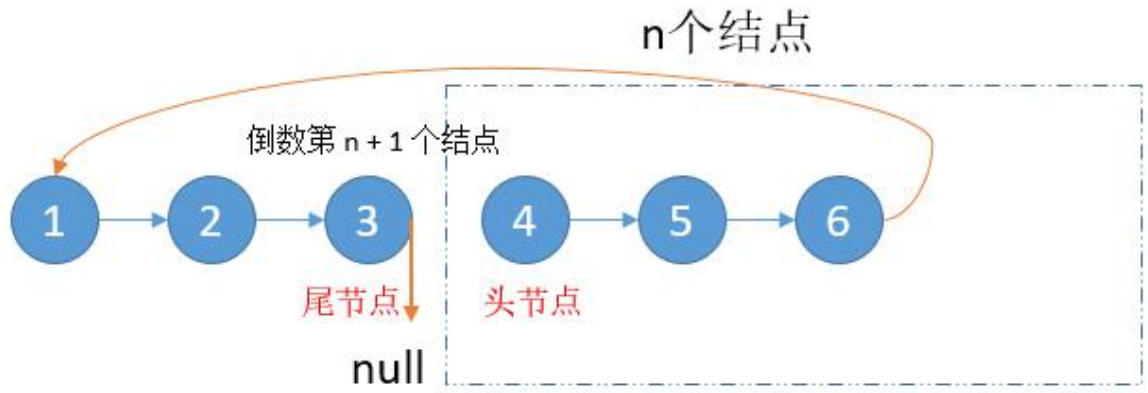
```
Input: 0->1->2->NULL, k = 4
Output: 2->0->1->NULL
Explanation:
rotate 1 steps to the right: 2->0->1->NULL
rotate 2 steps to the right: 1->2->0->NULL
rotate 3 steps to the right: 0->1->2->NULL
rotate 4 steps to the right: 2->0->1->NULL
```

将最后一个链表节点移到最前边，然后重复这个过程 k 次。

解法一

很明显我们不需要真的一个一个移，如果链表长度是 len， n = k % len，我们只需要将末尾 n 个链表节点整体移动到最前边就可以了。可以结合下边的图看一下，我们只需要找到倒数 n + 1 个节点的指针把它指向 null，以及末尾的指针指向头结点就可以了。找倒数 n 个结点，让我想到了 19题，利用快慢指针。






```java
public ListNode rotateRight(ListNode head, int k) {
    if (head == null || k == 0) {
        return head;
    }
    int len = 0;
    ListNode h = head;
    ListNode tail = null;
    //求出链表长度，保存尾指针
    while (h != null) {
        h = h.next;
        len++;
        if (h != null) {
            tail = h;
        }
    }
    //求出需要整体移动多少个节点
    int n = k % len;
    if (n == 0) {
```

```
            return head;
        }

        //利用快慢指针找出倒数 n + 1 个节点的指针，用 slow 保存
        ListNode fast = head;
        while (n >= 0) {
            fast = fast.next;
            n--;
        }
        ListNode slow = head;
        while (fast != null) {
            slow = slow.next;
            fast = fast.next;
        }
        //尾指针指向头结点
        tail.next = head;
        //头指针更新为倒数第 n 个节点
        head = slow.next;
        //尾指针置为 null
        slow.next = null;
        return head;
    }
```

时间复杂度：O(n)。

空间复杂度：O（1）。

这里我们用到的快慢指针其实没有必要，快慢指针的一个优点是，不需要知道链表长度就可以找到倒数第 n 个节点。而这个算法中，我们在之前已经求出了 len ，所以我们其实可以直接找倒数第 n + 1 个节点。

```
ListNode slow = head;
for (int i = 1; i < len - n; i++) {
    slow = slow.next;
}
```

总

这道题也没有什么技巧，只要对链表很熟，把题理解了，很快就解出来了。

62、题目描述（中等难度）

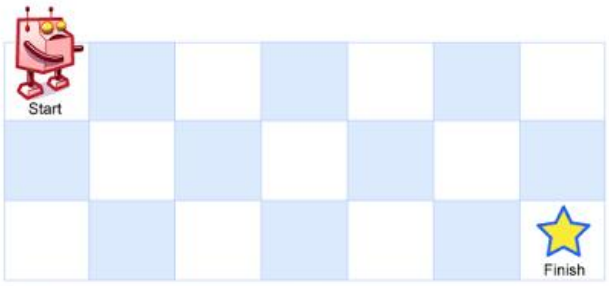
62. Unique Paths

Medium 1436 102 Favorite Share

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?



Above is a 7 x 3 grid. How many possible unique paths are there?

Note: *m* and *n* will be at most 100.

Example 1:

```
Input: m = 3, n = 2
Output: 3
Explanation:
From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:
1. Right -> Right -> Down
2. Right -> Down -> Right
3. Down -> Right -> Right
```

Example 2:

```
Input: m = 7, n = 3
Output: 28
```

机器人从左上角走到右下角，只能向右或者向下走。输出总共有多少种走法。

解法一 递归

求 (0 , 0) 点到（ m - 1 , n - 1） 点的走法。

（0，0）点到（m - 1 , n - 1） 点的走法等于（0，0）点右边的点 （1，0）到（m - 1 , n - 1）的走法加上（0，0）点下边的点（0，1）到（m - 1 , n - 1）的走法。

而左边的点（1，0）点到（m - 1 , n - 1） 点的走法等于（2，0） 点到（m - 1 , n - 1）的走法加上（1，1）点到（m - 1 , n - 1）的走法。

下边的点（0，1）点到（m - 1 , n - 1） 点的走法等于（1，1）点到（m - 1 , n - 1）的走法加上（0，2）点到（m - 1 , n - 1）的走法。

然后一直递归下去，直到 （m - 1 , n - 1） 点到（m - 1 , n - 1） ，返回 1。

```
public int uniquePaths(int m, int n) {
    HashMap<String, Integer> visited = new HashMap<>();
    return getAns(0, 0, m - 1, n - 1, 0);

}

private int getAns(int x, int y, int m, int n, int num) {
    if (x == m && y == n) {
        return 1;
    }
    int n1 = 0;
    int n2 = 0;
    //向右探索的所有解
    if (x + 1 <= m) {
        n1 = getAns(x + 1, y, m, n, num);
    }
    //向左探索的所有解
    if (y + 1 <= n) {
        n2 = getAns(x, y + 1, m, n, num);
    }
    //加起来
    return n1 + n2;
}
```

时间复杂度：

空间复杂度：

遗憾的是，这个算法在 LeetCode 上超时了。我们可以优化下，问题出在当我们求点 （x，y）到（m - 1 , n - 1） 点的走法的时候，递归求了点 （x，y）点右边的点 （x + 1，0）到（m - 1 , n - 1）的走法和（x，y）下边的点（x，y + 1）到（m - 1 , n - 1）的走法。而没有考虑到（x + 1，0）到（m - 1 , n - 1）的走法和点（x，y + 1）到（m - 1 , n - 1）的走法是否是之前已经求过了。事实上，很多点求的时候后边的的点已经求过了，所以再进行递归是没有必要的。基于此，我们可以用 visited 保存已经求过的点。

```
public int uniquePaths(int m, int n) {
    HashMap<String, Integer> visited = new HashMap<>();
    return getAns(0, 0, m - 1, n - 1, 0, visited);

}
private int getAns(int x, int y, int m, int n, int num, HashMap<String, Integer>
visited) {
    if (x == m && y == n) {
        return 1;
    }
    int n1 = 0;
    int n2 = 0;
```

```
    String key = x + 1 + "@" + y;
    //判断当前点是否已经求过了
    if (!visited.containsKey(key)) {
        if (x + 1 <= m) {
            n1 = getAns(x + 1, y, m, n, num, visited);
        }
    } else {
        n1 = visited.get(key);
    }
    key = x + "@" + (y + 1);
    if (!visited.containsKey(key)) {
        if (y + 1 <= n) {
            n2 = getAns(x, y + 1, m, n, num, visited);
        }
    } else {
        n2 = visited.get(key);
    }
    //将当前点加入 visited 中
    key = x + "@" + y;
    visited.put(key, n1+n2);
    return n1 + n2;
}
```

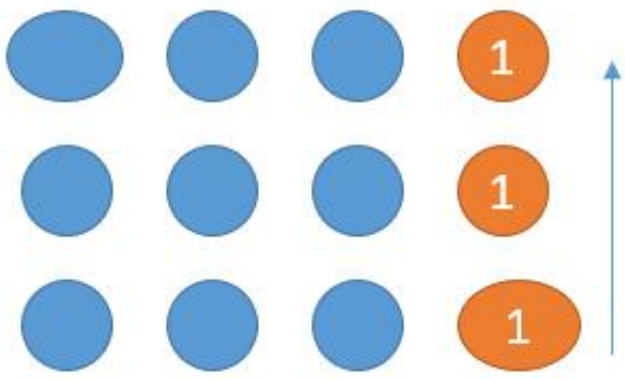
时间复杂度：

空间复杂度：

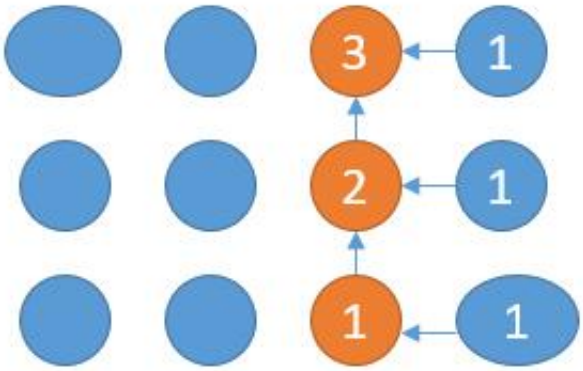
解法二 动态规划

解法一是基于递归的，压栈浪费了很多时间。我们来分析一下，压栈的过程，然后我们其实完全可以省略压栈的过程，直接用迭代去实现。

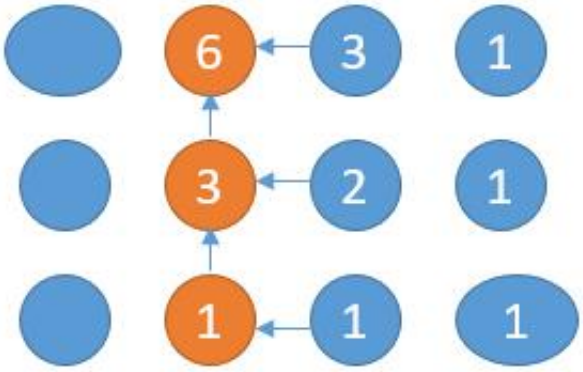
如下图，如果是递归的话，根据上边的代码，从 （0，0）点向右压栈，向右压栈，到最右边后，就向下压栈，向下压栈，到最下边以后，就开始出栈。出栈过程就是橙色部分。

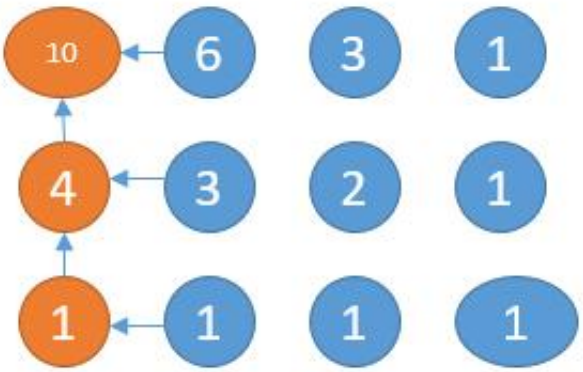


然后根据代码，继续压栈前一列，下图的橙色部分，然后到最下边后，然后开始出栈，根据它的右边的点和下边的点计算当前的点的走法。



接下来两步同理，压栈，出栈。





我们现在要做的就是要省略压栈的过程，直接出栈。很明显可以做到的，只需要初始化最后一列为 1 ，然后 1 列，1 列的向前更新就可以了。有一些动态规划的思想了。

```
public int uniquePaths(int m, int n) {
```

```
        int[] dp = new int[m];
        //初始化最后一列
        for (int i = 0; i < m; i++) {
            dp[i] = 1;
        }
        //从右向左更新所有列
        for (int i = n - 2; i >= 0; i--) {
            //最后一行永远是 1，所以从倒数第 2 行开始
            //从下向上更新所有行
            for (int j = m - 2; j >= 0; j--) {
                //右边的和下边的更新当前元素
                dp[j] = dp[j] + dp[j + 1];
            }
        }
        return dp[0];
    }
```

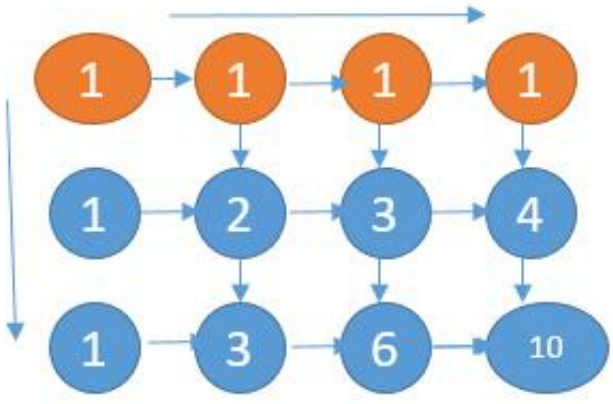
时间复杂度：O（m * n）。

空间复杂度：O（m）。

这里也有一个类似的想法。不过他是正向考虑的，和上边的想法刚好相反。如果把 dp [i] [j] 表示为从点（0，0）到点 (i，j）的走法。

上边解法公式就是 dp [i] [j] = dp [i + 1] [j] + dp [i] [j +1]。

这里的话就是 dp [i] [j] = dp [i - 1] [j] + dp [i] [j - 1]。就是用它左边的和上边的更新，可以结合下图。



这样的话，就是从左向右，从上到下一行一行更新（当前也可以一列一列更新）。

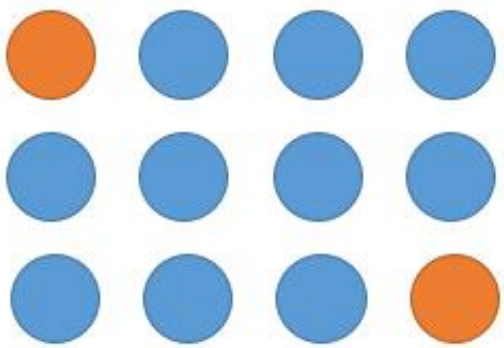
```
public int uniquePaths(int m, int n) {
    int[] dp = new int[n];
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
    }

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[j] = dp[j] + dp[j - 1];
        }
    }
    return dp[(n - 1)];
}
```

时间复杂度：O（m * n）。

空间复杂度：O（n）。

解法三 公式

参考这里。



我们用 R 表示向右，D 表示向下，然后把所有路线写出来，就会发现神奇的事情了。

R R R D D

R R D D R

R D R D R

……

从左上角，到右下角，总会是 3 个 R，2 个 D，只是出现的顺序不一样。所以求解法，本质上是求了组合数，N = m + n - 2，也就是总共走的步数。 k = m - 1，也就是向下的步数，D 的个数。所以总共的解就是 $$C^k_n = n!/(k!(n-k)!) = (n*(n-1)*(n-2)*...(n-k+1))/k!$$。

```
public int uniquePaths(int m, int n) {
    int N = n + m - 2;
    int k = m - 1;
    long res = 1;
    for (int i = 1; i <= k; i++)
        res = res * (N - k + i) / i;
    return (int) res;
}
```

时间复杂度：O（m）。

空间复杂度：O（1）。

总

从递归，到递归改迭代，之前的题也都遇到过了，本质上就是省去压栈的过程。解法三的公式法，直接到问题的本质，很厉害。

63、题目描述（中等难度）

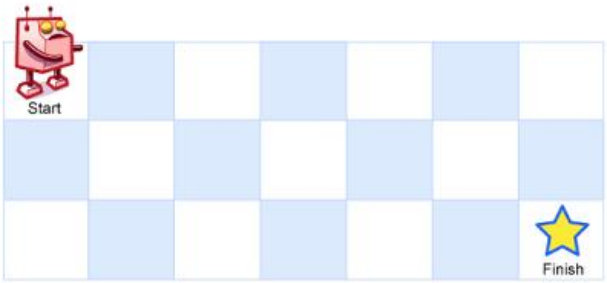
63. Unique Paths II

Medium 802 98 Favorite Share

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

Now consider if some obstacles are added to the grids. How many unique paths would there be?



An obstacle and empty space is marked as `1` and `0` respectively in the grid.

Note: *m* and *n* will be at most 100.

Example 1:

```
Input:
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
Output: 2
Explanation:
There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right
```

对62题的变体，增加了一些不能走的格子，用 1 表示。还是输出从左上角到右下角总共有多少种走法。

没做过62题的话可以先看一下，62 题总结的很详细了，我直接在 62 题的基础上改了。

解法一 递归

```java
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    int m = obstacleGrid.length;
    int n = obstacleGrid[0].length;
    HashMap<String, Integer> visited = new HashMap<>();
    //起点是障碍，直接返回 0
    if (obstacleGrid[0][0] == 1)
        return 0;
    return getAns(0, 0, m - 1, n - 1, 0, visited, obstacleGrid);
}
```

```
private int getAns(int x, int y, int m, int n, int num, HashMap<String, Integer>
visited, int[][] obstacleGrid) {
    // TODO Auto-generated method stub
    if (x == m && y == n) {
        return 1;
    }
    int n1 = 0;
    int n2 = 0;
    String key = x + 1 + "@" + y;
    if (!visited.containsKey(key)) {
        //与 62 题不同的地方，增加了判断是否是障碍
        if (x + 1 <= m && obstacleGrid[x + 1][y] == 0) {
            n1 = getAns(x + 1, y, m, n, num, visited, obstacleGrid);
        }
    } else {
        n1 = visited.get(key);
    }
    key = x + "@" + (y + 1);
    if (!visited.containsKey(key)) {
        //与 62 题不同的地方，增加了判断是否是障碍
        if (y + 1 <= n && obstacleGrid[x][y + 1] == 0) {
            n2 = getAns(x, y + 1, m, n, num, visited, obstacleGrid);
        }
    } else {
        n2 = visited.get(key);
    }
    //将当前点加入 visited 中
    key = x + "@" + y;
    visited.put(key, n1+n2);
    return n1 + n2;
}
```

时间复杂度：

空间复杂度：

解法二 动态规划

在62题解法二最后个想法上改。

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
    int m = obstacleGrid.length;
    int n = obstacleGrid[0].length;
    //起点是障碍，直接返回 0
    if (obstacleGrid[0][0] == 1)
```

```
        return 0;
    int[] dp = new int[n];
    int i = 0;
    //初始化第一行和 62 题不一样了
    //这里的话不是全部初始化 1 了，因为如果有障碍的话当前列和后边的列就都走不过了，初始化为 0
    for (; i < n; i++) {
        if (obstacleGrid[0][i] == 1) {
            dp[i] = 0;
            break;
        } else {
            dp[i] = 1;
        }
    }
    //障碍后边的列全部初始化为 0
    for (; i < n; i++) {
        dp[i] = 0;
    }
    for (i = 1; i < m; i++) {
        //这里改为从 0 列开始了，因为 62 题中从 1 开始是因为第 0 列永远是 1 不需要更新
        //这里的话，如果第 0 列是障碍的话，需要更新为 0
        for (int j = 0; j < n; j++) {
            if (obstacleGrid[i][j] == 1) {
                dp[j] = 0;
            } else {
                //由于从第 0 列开始了，更新公式有 j - 1，所以 j = 0 的时候不更新
                if (j != 0)
                    dp[j] = dp[j] + dp[j - 1];
            }
        }
    }
    return dp[(n - 1)];
}
```

时间复杂度：O（m * n）。

空间复杂度：O（n）。

总

和 62 题改动不大，就是在障碍的地方，更新的时候需要注意一下。

64、题目描述（中等难度）

64. Minimum Path Sum

Medium 1252 35 Favorite Share

Given a *m* x *n* grid filled with non-negative numbers, find a path from top left to bottom right which *minimizes* the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example:

```
Input:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
Output: 7
Explanation: Because the path 1→3→1→1→1 minimizes the sum.
```

依旧是62题的扩展，这个是输出从左上角到右下角，路径的数字加起来和最小是多少。

依旧在62题代码的基础上改，大家可以先看下 62 题。

解法一 递归

62 题中我们把递归 getAns 定义为，输出 （x，y）到 （m ，n ） 的路径数，如果记做 dp[x][y]。

那么递推式就是 dp[x][y] = dp[x][y+1] + dp[x+1][y]。

这道题的话，把递归 getAns 定义为，输出 （x，y）到 （m，n ） 的路径和最小是多少。同样如果记做 dp[x][y]。这样的话， dp[x][y] = Min（dp[x][y+1] + dp[x+1][y]）+ grid[x][y]。很好理解，就是当前点的右边和下边取一个和较小的，然后加上当前点的权值。

```
public int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    HashMap<String, Integer> visited = new HashMap<>();
    return getAns(0, 0, m - 1, n - 1, 0, visited, grid);
}

private int getAns(int x, int y, int m, int n, int num, HashMap<String, Integer>
visited, int[][] grid) {
    // 到了终点，返回终点的权值
    if (x == m && y == n) {
        return grid[m][n];
    }
    int n1 = Integer.MAX_VALUE;
    int n2 = Integer.MAX_VALUE;
    String key = x + 1 + "@" + y;
```

```
    if (!visited.containsKey(key)) {
        if (x + 1 <= m) {
            n1 = getAns(x + 1, y, m, n, num, visited, grid);
        }
    } else {
        n1 = visited.get(key);
    }
    key = x + "@" + (y + 1);
    if (!visited.containsKey(key)) {
        if (y + 1 <= n) {
            n2 = getAns(x, y + 1, m, n, num, visited, grid);
        }
    } else {
        n2 = visited.get(key);
    }
    // 将当前点加入 visited 中
    key = x + "@" + y;
    visited.put(key, Math.min(n1, n2) + grid[x][y]);
    //返回两个之间较小的，并且加上当前权值
    return Math.min(n1, n2) + grid[x][y];
}
```

时间复杂度：

空间复杂度：

解法二

这里我们直接用 grid 覆盖存，不去 new 一个 n 的空间了。

```
public int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;
    //由于第一行和第一列不能用我们的递推式，所以单独更新
    //更新第一行的权值
    for (int i = 1; i < n; i++) {
        grid[0][i] = grid[0][i - 1] + grid[0][i];
    }
    //更新第一列的权值
    for (int i = 1; i < m; i++) {
        grid[i][0] = grid[i - 1][0] + grid[i][0];
    }
    //利用递推式更新其它的
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            grid[i][j] = Math.min(grid[i][j - 1], grid[i - 1][j]) + grid[i][j];
```

```
            }
        }
        return grid[m - 1][n - 1];
    }
```

时间复杂度：O（m * n）。

空间复杂度：O（1）。

总

依旧是62题的扩展，理解了 62 题的话，很快就写出来了。

65、题目描述（困难难度）

65. Valid Number

Hard 402 3028 Favorite Share

Validate if a given string can be interpreted as a decimal number.

Some examples:
`"0" => true`
`" 0.1 " => true`
`"abc" => false`
`"1 a" => false`
`"2e10" => true`
`" -90e3 " => true`
`" 1e" => false`
`"e3" => false`
`" 6e-1" => true`
`" 99e2.5 " => false`
`"53.5e93" => true`
`" --6 " => false`
`"-+3" => false`
`"95a54e53" => false`

Note: It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one. However, here is a list of characters that can be in a valid decimal number:

- Numbers 0-9
- Exponent - "e"
- Positive/negative sign - "+"/"-"
- Decimal point - "."

Of course, the context of these characters also matters in the input.

给定一个字符串，判断它是否代表合法数字，当然题目描述的样例不够多，会使得设计算法中出现很多遗漏的地方，这里直接参考评论区@yeelan0319给出的更多测试样例。

```
test(1, "123", true);
```

```
test(2, " 123 ", true);
test(3, "0", true);
test(4, "0123", true);  //Cannot agree
test(5, "00", true);  //Cannot agree
test(6, "-10", true);
test(7, "-0", true);
test(8, "123.5", true);
test(9, "123.000000", true);
test(10, "-500.777", true);
test(11, "0.0000001", true);
test(12, "0.00000", true);
test(13, "0.", true);  //Cannot be more disagree!!!
test(14, "00.5", true);  //Strongly cannot agree
test(15, "123e1", true);
test(16, "1.23e10", true);
test(17, "0.5e-10", true);
test(18, "1.0e4.5", false);
test(19, "0.5e04", true);
test(20, "12 3", false);
test(21, "1a3", false);
test(22, "", false);
test(23, "     ", false);
test(24, null, false);
test(25, ".1", true); //Ok, if you say so
test(26, ".", false);
test(27, "2e0", true);  //Really?!
test(28, "+.8", true);
test(29, " 005047e+6", true);  //Damn = =|||
```

解法一 直接法

什么叫直接法呢，就是没有什么通用的方法，直接分析题目，然后写代码，直接贴两个 leetcode Disscuss 的代码吧，供参考。

想法一。

把当前的输入分成几类，再用几个标志位来判断当前是否合法。

```
public boolean isNumber(String s) {
    s = s.trim();

    boolean pointSeen = false;
    boolean eSeen = false;
    boolean numberSeen = false;
    boolean numberAfterE = true;
    for(int i=0; i<s.length(); i++) {
```

```
        if('0' <= s.charAt(i) && s.charAt(i) <= '9') {
            numberSeen = true;
            numberAfterE = true;
        } else if(s.charAt(i) == '.') {
            if(eSeen || pointSeen) {
                return false;
            }
            pointSeen = true;
        } else if(s.charAt(i) == 'e') {
            if(eSeen || !numberSeen) {
                return false;
            }
            numberAfterE = false;
            eSeen = true;
        } else if(s.charAt(i) == '-' || s.charAt(i) == '+') {
            if(i != 0 && s.charAt(i-1) != 'e') {
                return false;
            }
        } else {
            return false;
        }
    }

    return numberSeen && numberAfterE;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

想法二，遍历过程中，把遇到不符合的都返回 false，到最后成功到达末尾就返回 true。C++ 的代码，可以参考一下思想。

```
bool isNumber(const char *s)
{
    int i = 0;

    // skip the whilespaces
    for(; s[i] == ' '; i++) {}

    // check the significand
    if(s[i] == '+' || s[i] == '-') i++; // skip the sign if exist

    int n_nm, n_pt;
    for(n_nm=0, n_pt=0; (s[i]<='9' && s[i]>='0') || s[i]=='.'; i++)
        s[i] == '.' ? n_pt++:n_nm++;
```

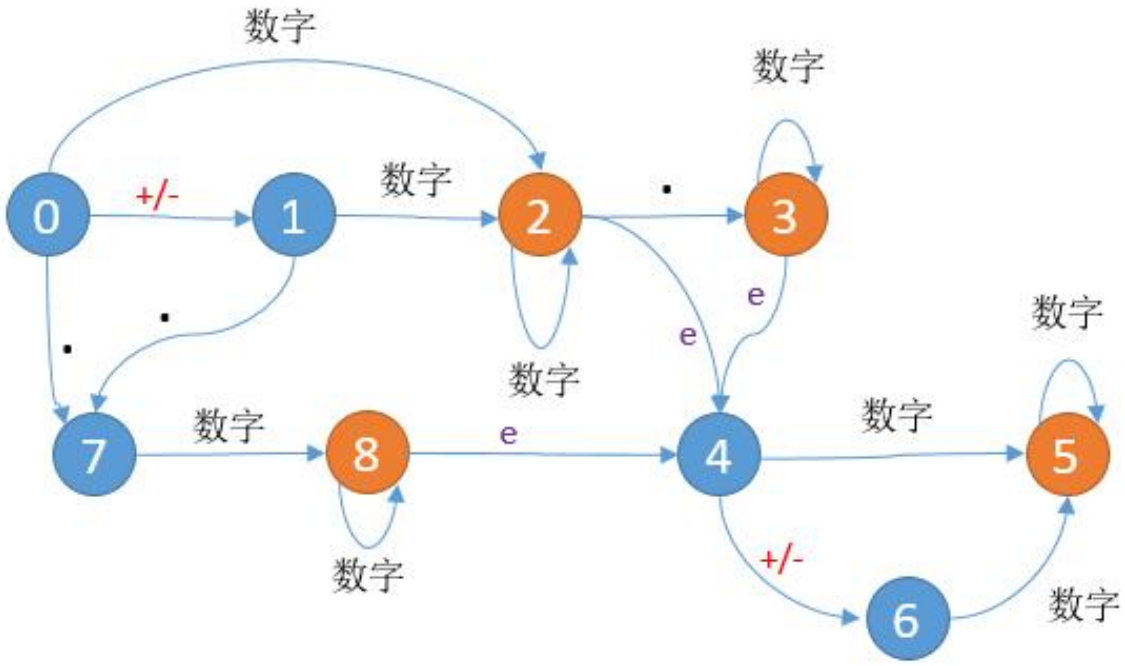
```
    if(n_pt>1 || n_nm<1) // no more than one point, at least one digit
        return false;

    // check the exponent if exist
    if(s[i] == 'e') {
        i++;
        if(s[i] == '+' || s[i] == '-') i++; // skip the sign

        int n_nm = 0;
        for(; s[i]>='0' && s[i]<='9'; i++, n_nm++) {}
        if(n_nm<1)
            return false;
    }

    // skip the trailing whitespaces
    for(; s[i] == ' '; i++) {}

    return s[i]==0;  // must reach the ending 0 of the string
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

解法二 自动机

自己最开始想到的就是这个，编译原理时候在学到的自动机，就是一些状态转移。这一块内容很多，自己也很多东西都忘了，但不影响我们写算法，主要参考这里。



如上图，从 0 开始总共有 9 个状态，橙色代表可接受状态，也就是表示此时是合法数字。总共有四大类输入，数字，小数点，e 和 正负号。我们只需要将这个图实现就够了。

```
public boolean isNumber(String s) {
    int state = 0;
    s = s.trim();//去除头尾的空格
    //遍历所有字符，当做输入
    for (int i = 0; i < s.length(); i++) {
        switch (s.charAt(i)) {
             //输入正负号
            case '+':
            case '-':
                if (state == 0) {
                    state = 1;
                } else if (state == 4) {
                    state = 6;
                } else {
                    return false;
                }
                break;
            //输入数字
            case '0':
            case '1':
            case '2':
            case '3':
```

```
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                //根据当前状态去跳转
                switch (state) {
                    case 0:
                    case 1:
                    case 2:
                        state = 2;
                        break;
                    case 3:
                        state = 3;
                        break;
                    case 4:
                    case 5:
                    case 6:
                        state = 5;
                        break;
                    case 7:
                        state = 8;
                        break;
                    case 8:
                        state = 8;
                        break;
                    default:
                        return false;
                }
                break;
            //小数点
            case '.':
                switch (state) {
                    case 0:
                    case 1:
                        state = 7;
                        break;
                    case 2:
                        state = 3;
                        break;
                    default:
                        return false;
                }
                break;
            //e
```

```
            case 'e':
                switch (state) {
                    case 2:
                    case 3:
                    case 8:
                        state = 4;
                        break;
                    default:
                        return false;
                }
                break;
            default:
                return false;

        }
    }
    //橙色部分的状态代表合法数字
    return state == 2 || state == 3 || state == 5 || state == 8;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

解法三 责任链模式

解法二看起来已经很清晰明了了，只需要把状态图画出来，然后实现代码就很简单了。但是缺点是，如果状态图少考虑了东西，再改起来就会很麻烦。

这里作者提出来，利用责任链的设计模式，会使得写出的算法扩展性以及维护性更高。这里用到的思想就是，每个类只判断一种类型。比如判断是否是正数的类，判断是否是小数的类，判断是否是科学计数法的类，这样每个类只关心自己的部分，出了问题很好排查，而且互不影响。

```
//每个类都实现这个接口
interface NumberValidate {
    boolean validate(String s);
}
//定义一个抽象类，用来检查一些基础的操作，是否为空，去掉首尾空格，去掉 +/-
//doValidate 交给子类自己去实现
abstract class  NumberValidateTemplate implements NumberValidate{

    public boolean validate(String s)
    {
        if (checkStringEmpty(s))
        {
            return false;
```

```
        }

        s = checkAndProcessHeader(s);

        if (s.length() == 0)
        {
            return false;
        }

        return doValidate(s);
    }

    private boolean checkStringEmpty(String s)
    {
        if (s.equals(""))
        {
            return true;
        }

        return false;
    }

    private String checkAndProcessHeader(String value)
    {
        value = value.trim();

        if (value.startsWith("+") || value.startsWith("-"))
        {
            value = value.substring(1);
        }


        return value;
    }



    protected abstract boolean doValidate(String s);
}

//实现 doValidate 判断是否是整数
class IntegerValidate extends NumberValidateTemplate{

    protected boolean doValidate(String integer)
    {
        for (int i = 0; i < integer.length(); i++)
```

```
        {
            if(Character.isDigit(integer.charAt(i)) == false)
            {
                return false;
            }
        }

        return true;
    }
}

//实现 doValidate 判断是否是科学计数法
class SienceFormatValidate extends NumberValidateTemplate{

    protected boolean doValidate(String s)
    {
        s = s.toLowerCase();
        int pos = s.indexOf("e");
        if (pos == -1)
        {
            return false;
        }

        if (s.length() == 1)
        {
            return false;
        }

        String first = s.substring(0, pos);
        String second = s.substring(pos+1, s.length());

        if (validatePartBeforeE(first) == false || validatePartAfterE(second) ==
false)
        {
            return false;
        }


        return true;
    }

    private boolean validatePartBeforeE(String first)
    {
        if (first.equals("") == true)
        {
            return false;
```

```
        }

        if (checkHeadAndEndForSpace(first) == false)
        {
            return false;
        }

        NumberValidate integerValidate = new IntegerValidate();
        NumberValidate floatValidate = new FloatValidate();
        if (integerValidate.validate(first) == false &&
floatValidate.validate(first) == false)
        {
            return false;
        }

        return true;
    }

    private boolean checkHeadAndEndForSpace(String part)
    {

        if (part.startsWith(" ") ||
            part.endsWith(" "))
        {
            return false;
        }

        return true;
    }

    private boolean validatePartAfterE(String second)
    {
        if (second.equals("") == true)
        {
            return false;
        }

        if (checkHeadAndEndForSpace(second) == false)
        {
            return false;
        }

        NumberValidate integerValidate = new IntegerValidate();
        if (integerValidate.validate(second) == false)
        {
            return false;
```

```
        }

        return true;
    }
}
//实现 doValidate 判断是否是小数
class FloatValidate extends NumberValidateTemplate{

    protected boolean doValidate(String floatVal)
    {
        int pos = floatVal.indexOf(".");
        if (pos == -1)
        {
            return false;
        }

        if (floatVal.length() == 1)
        {
            return false;
        }

        NumberValidate nv = new IntegerValidate();
        String first = floatVal.substring(0, pos);
        String second = floatVal.substring(pos + 1, floatVal.length());

        if (checkFirstPart(first) == true && checkFirstPart(second) == true)
        {
            return true;
        }

        return false;
    }

    private boolean checkFirstPart(String first)
    {
        if (first.equals("") == false && checkPart(first) == false)
        {
            return false;
        }

        return true;
    }

    private boolean checkPart(String part)
    {
        if (Character.isDigit(part.charAt(0)) == false ||
```

```
            Character.isDigit(part.charAt(part.length() - 1)) == false)
        {
            return false;
        }

        NumberValidate nv = new IntegerValidate();
        if (nv.validate(part) == false)
        {
            return false;
        }

        return true;
    }
}
//定义一个执行者，我们把之前实现的各个类加到一个数组里，然后依次调用
class NumberValidator implements NumberValidate {

    private ArrayList<NumberValidate> validators = new ArrayList<NumberValidate>
();

    public NumberValidator()
    {
        addValidators();
    }

    private  void addValidators()
    {
        NumberValidate nv = new IntegerValidate();
        validators.add(nv);

        nv = new FloatValidate();
        validators.add(nv);

        nv = new HexValidate();
        validators.add(nv);

        nv = new SienceFormatValidate();
        validators.add(nv);
    }

    @Override
    public boolean validate(String s)
    {
        for (NumberValidate nv : validators)
        {
            if (nv.validate(s) == true)
```

```
            {
                return true;
            }
        }

        return false;
    }


}
public boolean isNumber(String s) {
    NumberValidate nv = new NumberValidator();
    return nv.validate(s);
}
```

时间复杂度：

空间复杂度：

总

解法二中自动机的应用，会使得自己的思路更清晰。而解法三中，作者提出的对设计模式的应用，使自己眼前一亮，虽然代码变多了，但是维护性，扩展性变的很强了。比如，题目新增了一种情况，"0x123" 16 进制也算是合法数字。这样的话，解法一和解法二就没什么用了，完全得重新设计。但对于解法三，我们只需要新增一个类，专门判断这种情况，然后加到执行者的数组里就够了，很棒！

66、题目描述（简单难度）

Given a **non-empty** array of digits representing a non-negative integer, plus one to the integer.

The digits are stored such that the most significant digit is at the head of the list, and each element in the array contain a single digit.

You may assume the integer does not contain any leading zero, except the number 0 itself.

Example 1:

```
Input: [1,2,3]
Output: [1,2,4]
Explanation: The array represents the integer 123.
```

Example 2:

```
Input: [4,3,2,1]
Output: [4,3,2,2]
Explanation: The array represents the integer 4321.
```

数组代表一个数字，[1, 2, 3] 就代表 123，然后给它加上 1，输出新的数组。数组每个位置只保存 1 位，也就是 0 到 9。

解法一 递归

先用递归，好理解一些。

```
public int[] plusOne(int[] digits) {
    return plusOneAtIndex(digits, digits.length - 1);
}

private int[] plusOneAtIndex(int[] digits, int index) {
    //说明每一位都是 9
    if (index < 0) {
        //新建一个更大的数组，最高位赋值为 1
        int[] ans = new int[digits.length + 1];
        ans[0] = 1;
        //其他位赋值为 0，因为 java 里默认是 0，所以不需要管了
        return ans;
    }
    //如果当前位小于 9，直接加 1 返回
    if (digits[index] < 9) {
        digits[index] += 1;
        return digits;
    }

    //否则的话当前位置为 0，
    digits[index] = 0;
    //考虑给前一位加 1
    return plusOneAtIndex(digits, index - 1);

}
```

时间复杂度：O（n）。

空间复杂度：

解法二 迭代

上边的递归，属于尾递归，完全没必要压栈，直接改成迭代的形式吧。

```
public int[] plusOne(int[] digits) {
    //从最低位遍历
    for (int i = digits.length - 1; i >= 0; i--) {
        //小于 9 的话，直接加 1，结束循环
        if (digits[i] < 9) {
            digits[i] += 1;
            break;
```

```
            }
            //否则的话置为 0
            digits[i] = 0;
        }
        //最高位如果置为 0 了，说明最高位产生了进位
        if (digits[0] == 0) {
            int[] ans = new int[digits.length + 1];
            ans[0] = 1;
            digits = ans;
        }
        return digits;
    }
```

时间复杂度：O（n）。

空间复杂度：

总

很简单的一道题，理解题意就可以了。有的编译器不进行尾递归优化，他遇到这种尾递归还是不停压栈压栈压栈，所以这种简单的我们直接用迭代就行。

67、题目描述（简单难度）

67. Add Binary

Easy 907 183 Favorite Share

Given two binary strings, return their sum (also a binary string).

The input strings are both **non-empty** and contains only characters `1` or `0`.

Example 1:

```
Input: a = "11", b = "1"
Output: "100"
```

Example 2:

```
Input: a = "1010", b = "1011"
Output: "10101"
```

两个二进制数相加，返回结果，要注意到字符串的最低位代表着数字的最高位。例如 "100" 最高位（十进制中的百位的位置）是 1，但是对应的字符串的下标是 0。

解法一

开始的时候以为会有什么特殊的方法，然后想着不管了，先按第二题两个十进制数相加的想法写吧。

```
public String addBinary(String a, String b) {
    StringBuilder ans = new StringBuilder();
    int i = a.length() - 1;
    int j = b.length() - 1;
    int carry = 0;
    while (i >= 0 || j >= 0) {
        int num1 = i >= 0 ? a.charAt(i) - 48 : 0;
        int num2 = j >= 0 ? b.charAt(j) - 48 : 0;
        int sum = num1 + num2 + carry;
        carry = 0;
        if (sum >= 2) {
            sum = sum % 2;
            carry = 1;
        }
        ans.insert(0, sum);
        i--;
        j--;

    }
    if (carry == 1) {
        ans.insert(0, 1);
    }
    return ans.toString();
}
```

时间复杂度：O（max （m，n））。m 和 n 分别是字符串 a 和 b 的长度。

空间复杂度：O（1）。

然后写完以后，在 Discuss 里逛了逛，找找其他的解法。发现基本都是这个思路，但是奇怪的是我的解法，时间上只超过了 60% 的人。然后，点开了超过 100% 的人的解法。

```
public String addBinary2(String a, String b) {
    char[] charsA = a.toCharArray(), charsB = b.toCharArray();
    char[] sum = new char[Math.max(a.length(), b.length()) + 1];
    int carry = 0, index = sum.length - 1;
    for (int i = charsA.length - 1, j = charsB.length - 1; i >= 0 || j >= 0; i--,
j--)    {
        int aNum = i < 0 ? 0 : charsA[i] - '0';
        int bNum  = j < 0 ? 0 : charsB[j] - '0';

        int s = aNum + bNum + carry;
        sum[index--] = (char) (s % 2 + '0');
        carry = s / 2;
    }
    sum[index] = (char) ('0' + carry);
```

```
    return carry == 0 ? new String(sum, 1, sum.length - 1) : new String(sum);
}
```

和我的思路是一样的，区别在于它提前申请了 sum 的空间，然后直接 index 从最后向 0 依次赋值。

因为 String .charAt (0) 代表的是数字的最高位，而我们计算是从最低位开始的，也就是 lenght - 1开始的，所以在之前的算法中每次得到一个结果我们用的是 ans.insert(0, sum) ，在 0 位置插入新的数。我猜测是这里耗费了很多时间，因为插入的话，会导致数组的后移。

我们如果把 insert 换成 append ，然后再最后的结果中再倒置，就会快一些了。

```
public String3 addBinary(String a, String b) {
    StringBuilder ans = new StringBuilder();
    int i = a.length() - 1;
    int j = b.length() - 1;
    int carry = 0;
    while (i >= 0 || j >= 0) {
        int num1 = i >= 0 ? a.charAt(i) - 48 : 0;
        int num2 = j >= 0 ? b.charAt(j) - 48 : 0;
        int sum = num1 + num2 + carry;
        carry = 0;
        if (sum >= 2) {
            sum = sum % 2;
            carry = 1;
        }
        ans.append(sum);
        i--;
        j--;

    }
    if (carry == 1) {
        ans.append(1);
    }
    return ans.reverse().toString();
}
```

总

这里看出来多次 insert 会很耗费时间，不如最后直接 reverse。另外提前申请空间，直接根据下标赋值，省去了倒置的时间，很 cool。

68、题目描述（困难难度）

68. Text Justification

Description | Hints | Submissions | Discuss | Solution

Pick One

Given an array of words and a width *maxWidth*, format the text such that each line has exactly *maxWidth* characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces `' '` when necessary so that each line has exactly *maxWidth* characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no **extra** space is inserted between words.

Note:

- A word is defined as a character sequence consisting of non-space characters only.
- Each word's length is guaranteed to be greater than 0 and not exceed *maxWidth*.
- The input array `words` contains at least one word.

Example 1:

```
Input:
words = ["This", "is", "an", "example", "of", "text", "justification."]
maxWidth = 16
Output:
[
   "This    is    an",
   "example  of text",
   "justification.  "
]
```

Example 2:

```
Input:
words = ["What","must","be","acknowledgment","shall","be"]
maxWidth = 16
Output:
[
  "What   must   be",
  "acknowledgment  ",
  "shall be        "
]
Explanation: Note that the last line is "shall be    " instead of "shall     be",
             because the last line must be left-justified instead of fully-justified.
             Note that the second line is also left-justified becase it contains only one word.
```

Example 3:

```
Input:
words = ["Science","is","what","we","understand","well","enough","to","explain",
         "to","a","computer.","Art","is","everything","else","we","do"]
maxWidth = 20
Output:
[
  "Science  is  what we",
  "understand      well",
  "enough to explain to",
  "a  computer.  Art is",
  "everything  else  we",
  "do                  "
]
```

]

一个句子，和一个长度表示一行最长的长度，然后对齐文本，有下边几个规则。

1. 同一个单词只能出现在一行中，不能拆分
2. 一行如果只能放下一个单词，该单词放在最左边，然后空格补齐，例如 "acknowledgement#####"，这里只是我为了直观，# 表示空格，题目并没有要求。
3. 一行如果有多个单词，最左边和最右边不能有空格，每个单词间隙尽量平均，如果无法平均，把剩余的空隙从左边开始分配。例如，"enough###to###explain##to"，3 个间隙，每个 2 个空格的话，剩下 2 个空格，从左边依次添加一个空格。
4. 最后一行执行左对齐，单词间一个空格，末尾用空格补齐。

解法一

这道题关键就是理解题目，然后就是一些细节的把控了，我主要是下边的想法。

一行一行计算该行可以放多少个单词，然后计算单词间的空隙是多少，然后把它添加到结果中。

```
public List<String> fullJustify(String[] words, int maxWidth) {
    List<String> ans = new ArrayList<>();
    //当前行单词已经占用的长度
    int currentLen = 0;
    //保存当前行的单词
    List<String> row = new ArrayList<>();
    //遍历每个单词
    for (int i = 0; i < words.length;) {
        //判断加入该单词是否超过最长长度
        //分了两种情况，一种情况是加入第一个单词，不需要多加 1
        //已经有单词的话，再加入单词，需要多加个空格，所以多加了 1
        if (currentLen == 0 && currentLen + words[i].length() <= maxWidth
            || currentLen > 0 && currentLen + 1 + words[i].length() <= maxWidth)
{
            row.add(words[i]);
            if (currentLen == 0) {
                currentLen = currentLen + words[i].length();
            } else {
                currentLen = currentLen + 1 + words[i].length();
            }
            i++;
        //超过的最长长度，对 row 里边的单词进行处理
        } else {
            //计算有多少剩余，也就是总共的空格数，因为之前计算 currentLen 多算了一个空格，这
里加回来
            int sub = maxWidth - currentLen + row.size() - 1;
            //如果只有一个单词，那么就直接单词加空格就可以
            if (row.size() == 1) {
```

```
                String blank = getStringBlank(sub);
                ans.add(row.get(0) + blank);
            } else {
                //用来保存当前行的结果
                StringBuilder temp = new StringBuilder();
                //将第一个单词加进来
                temp.append(row.get(0));
                //计算平均空格数
                int averageBlank = sub / (row.size() - 1);
                //如果除不尽，计算剩余空格数
                int missing = sub - averageBlank * (row.size() - 1);
                //前 missing 的空格数比平均空格数多 1
                String blank = getStringBlank(averageBlank + 1);
                int k = 1;
                for (int j = 0; j < missing; j++) {
                    temp.append(blank + row.get(k));
                    k++;
                }
                //剩下的空格数就是求得的平均空格数
                blank = getStringBlank(averageBlank);
                for (; k < row.size(); k++) {
                    temp.append(blank + row.get(k));
                }
                //将当前结果加入
                ans.add(temp.toString());

            }
            //清空以及置零
            row = new ArrayList<>();
            currentLen = 0;

        }
    }
    //单独考虑最后一行，左对齐
    StringBuilder temp = new StringBuilder();
    temp.append(row.get(0));
    for (int i = 1; i < row.size(); i++) {
        temp.append(" " + row.get(i));
    }
    //剩余部分用空格补齐
    temp.append(getStringBlank(maxWidth - currentLen));
    //最后一行加入到结果中
    ans.add(temp.toString());
    return ans;
}
//得到 n 个空白
```

```
private String getStringBlank(int n) {
    StringBuilder str = new StringBuilder();
    for (int i = 0; i < n; i++) {
        str.append(" ");
    }
    return str.toString();
}
```

时间复杂度：

空间复杂度：

但是这个算法，在 leetcode 跑，速度只打败了 30% 多的人，1 ms。然后在 discuss 里转了一圈寻求原因，发现大家思路都是这样子，然后找了一个人的跑了下，链接。

```
public List<String> fullJustify(String[] words, int maxWidth) {
    int left = 0; List<String> result = new ArrayList<>();

    while (left < words.length) {
        int right = findRight(left, words, maxWidth);
        result.add(justify(left, right, words, maxWidth));
        left = right + 1;
    }

    return result;
}

//找到当前行最右边的单词下标
private int findRight(int left, String[] words, int maxWidth) {
    int right = left;
    int sum = words[right++].length();

    while (right < words.length && (sum + 1 + words[right].length()) <= maxWidth)
        sum += 1 + words[right++].length();

    return right - 1;
}

//根据不同的情况添加不同的空格
private String justify(int left, int right, String[] words, int maxWidth) {
    if (right - left == 0) return padResult(words[left], maxWidth);

    boolean isLastLine = right == words.length - 1;
    int numSpaces = right - left;
    int totalSpace = maxWidth - wordsLength(left, right, words);
```

```
    String space = isLastLine ? " " : blank(totalSpace / numSpaces);
    int remainder = isLastLine ? 0 : totalSpace % numSpaces;

    StringBuilder result = new StringBuilder();
    for (int i = left; i <= right; i++)
        result.append(words[i])
        .append(space)
        .append(remainder-- > 0 ? " " : "");

    return padResult(result.toString().trim(), maxWidth);
}

//当前单词的长度
private int wordsLength(int left, int right, String[] words) {
    int wordsLength = 0;
    for (int i = left; i <= right; i++) wordsLength += words[i].length();
    return wordsLength;
}

private String padResult(String result, int maxWidth) {
    return result + blank(maxWidth - result.length());
}

private String blank(int length) {
    return new String(new char[length]).replace('\0', ' ');
}
```

看了下，发现思想和自己也是一样的。但是这个速度却打败了 100% ，0 ms。考虑了下，差别应该在我的算法里使用了一个叫做 row 的 list 用来保存当前行的单词，用了很多 row.get (index)，而上边的算法只记录了 left 和 right 下标，取单词直接用的 words 数组。然后尝试着在我之前的算法上改了一下，去掉 row，用两个变量 start 和 end 保存当前行的单词范围。主要是 (end - start) 代替了之前的 row.size ( )，words [start + k] 代替了之前的 row.get (k)。

```
public List<String> fullJustify2(String[] words, int maxWidth) {
    List<String> ans = new ArrayList<>();
    int currentLen = 0;
    int start = 0;
    int end = 0;
    for (int i = 0; i < words.length;) {
        //判断加入该单词是否超过最长长度
        //分了两种情况，一种情况是加入第一个单词，不需要多加 1
        //已经有单词的话，再加入单词，需要多加个空格，所以多加了 1
        if (currentLen == 0 && currentLen + words[i].length() <= maxWidth
            || currentLen > 0 && currentLen + 1 + words[i].length() <= maxWidth)
{
```

```
            end++;
            if (currentLen == 0) {
                currentLen = currentLen + words[i].length();
            } else {
                currentLen = currentLen + 1 + words[i].length();
            }
            i++;
        } else {
            int sub = maxWidth - currentLen + (end - start) - 1;
            if (end - start == 1) {
                String blank = getStringBlank(sub);
                ans.add(words[start] + blank);
            } else {
                StringBuilder temp = new StringBuilder();
                temp.append(words[start]);
                int averageBlank = sub / ((end - start) - 1);
                 //如果除不尽，计算剩余空格数
                int missing = sub - averageBlank * ((end - start) - 1);
                String blank = getStringBlank(averageBlank + 1);
                int k = 1;
                for (int j = 0; j < missing; j++) {
                    temp.append(blank + words[start+k]);
                    k++;
                }
                blank = getStringBlank(averageBlank);
                for (; k <(end - start); k++) {
                    temp.append(blank + words[start+k]);
                }
                ans.add(temp.toString());

            }
            start = end;
            currentLen = 0;

        }
    }
    StringBuilder temp = new StringBuilder();
    temp.append(words[start]);
    for (int i = 1; i < (end - start); i++) {
        temp.append(" " + words[start+i]);
    }
    temp.append(getStringBlank(maxWidth - currentLen));
    ans.add(temp.toString());
    return ans;
}
//得到 n 个空白
```

```java
private String getStringBlank(int n) {
    StringBuilder str = new StringBuilder();
    for (int i = 0; i < n; i++) {
        str.append(" ");
    }
    return str.toString();
}
```

果然，速度也到了打败 100%，0 ms。

总

充分说明 list 的读取还是没有数组的直接读取快呀，还有就是要向上边的作者学习，多封装几个函数，思路会更加清晰，代码也会简明。

69、题目描述（简单难度）

69. Sqrt(x)

Easy 724 1313 Favorite Share

Implement `int sqrt(int x)`.

Compute and return the square root of *x*, where *x* is guaranteed to be a non-negative integer.

Since the return type is an integer, the decimal digits are truncated and only the integer part of the result is returned.

Example 1:

```
Input: 4
Output: 2
```

Example 2:

```
Input: 8
Output: 2
Explanation: The square root of 8 is 2.82842..., and since
             the decimal part is truncated, 2 is returned.
```

求一个数的平方根，不要求近似解，只需要整数部分。

解法一 二分法

本科的时候上计算方法的时候，讲过这个题的几个解法，二分法， 牛顿法，牛顿下山法，不同之处是之前是求近似解，类似误差是 0.0001 这样的。而这道题，只要求整数部分，所以先忘掉之前的解法，重新考虑一下。

求 n 的平方根的整数部分，所以平方根一定是 1，2，3 ... n 中的一个数。从一个有序序列中找一个数，像极了二分查找。先取中点 mid，然后判断 mid * mid 是否等于 n，小于 n 的话取左半部分，大于 n 的话取右半部分，等于 n 的话 mid 就是我们要找的了。

```
public int mySqrt(int x) {
    int L = 1, R = x;
    while (L <= R) {
        int mid =  (L + R) / 2;
        int square = mid * mid;
        if (square == x) {
            return mid;
        } else if (square < x) {
            L = mid + 1;
        } else {
            R = mid - 1;
        }
    }
    return ?;
}
```

正常的 2 分法，如果最后没有找到就返回 -1。但这里肯定是不行的，那应该返回什么呢?

对于平方数 4 9 16... 肯定会进入 square == x 然后结束掉。但是非平方数呢？例如 15。我们知道 15 的根，一定是 3 点几。因为 15 在 9 和 16 之间，9 的根是 3，16 的根是 4。所以对于 15，3 在这里就是我们要找的。 3 * 3 < 15，所以在上边算法中，最后的解是流向 square < x 的，所以我们用一个变量保存它，最后返回就可以了。

```
public int mySqrt(int x) {
    int L = 1, R = x;
    int ans = 0; //保存最后的解
    while (L <= R) {
        int mid =  (L + R) / 2;
        int square = mid * mid;
        if (square == x) {
            return mid;
        } else if (square < x) {
            ans = mid; //存起来以便返回
            L = mid + 1;
        } else {
            R = mid - 1;
        }
    }
```

```
    return ans;
}
```

看起来很完美了，但如果 x = Integer.MAX_VALUE 的话，下边两句代码是会溢出的。

```
int mid =  (L + R) / 2;
int square = mid * mid;
```

当然，我们把变量用 long 存就解决了，这里有一个更优雅的解法。利用数学的变形。

```
int mid = L + (R - L) / 2;
int div = x / mid;
```

当然相应的 if 语句也需要改变。

```
else if (square < x)
mid * mid < x
mid < x / mid
mid < div
```

全部加进去就可以了。

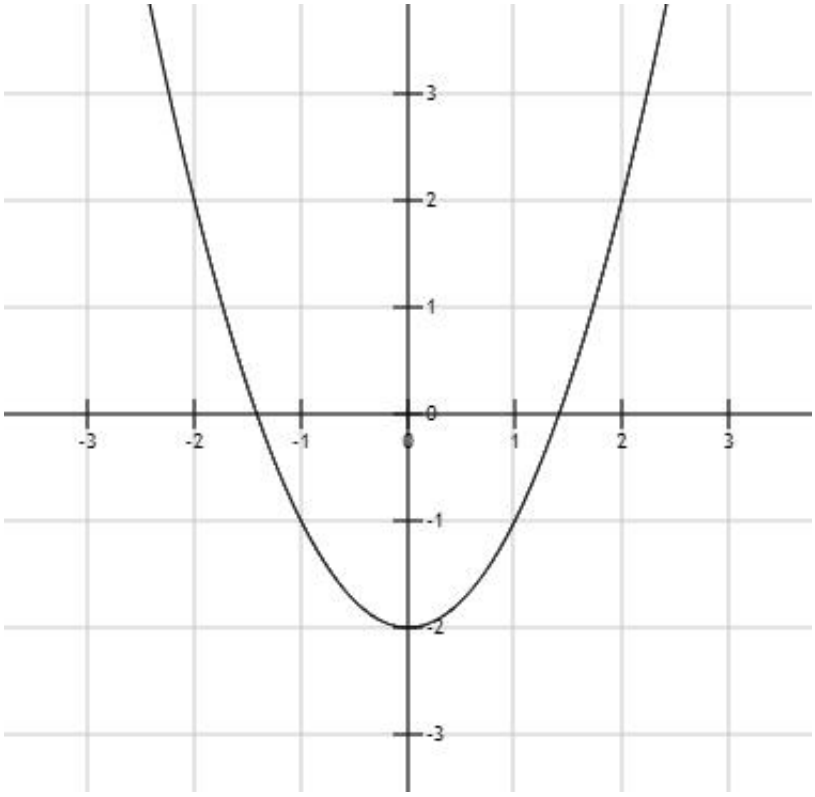
```
public int mySqrt(int x) {
    int L = 1, R = x;
    int ans = 0;
    while (L <= R) {
        int mid = L + (R - L) / 2;
        int div = x / mid;
        if (div == mid) {
            return mid;
        } else if (mid < div) {
            ans = mid;
            L = mid + 1;
        } else {
            R = mid - 1;
        }
    }
    return ans;
}
```

时间复杂度：O（log(x)）。

空间复杂度：O（1）。

解法二 二分法求精确解

把求根转换为求函数的零点，求 n 的平方根，也就是求函数 $f(x) = x^2 - n$ 的零点。这是一个二次曲线，与 x 轴有两个交点，我们要找的是那个正值。



这里基于零点定理，去写算法。

> 如果函数 $y = f(x)$ 在区间 $[a, b]$ 上的图像是连续不断的一条曲线，并且有$f(a) \cdot f(b) < 0$， 那么，函数$y = f(x)$ 在区间 (a, b) 内有零点，即存在 $c \in (a, b)$, 使得 $f(c) = 0$ ，这个 c 也就是方程 $f(x) = 0$ 的根。

简单的说，如果曲线上两点的值正负号相反，其间必定存在一个根。

这样我们就可以用二分法，找出中点，然后保留与中点的函数值符号相反的一段，丢弃另一段，然后继续找中点，直到符合我们的误差。

由于题目要求的是整数部分，所以我们需要想办法把我们的精确解转成整数。

四舍五入？由于我们求的是近似解，利用我们的算法我们求出的 8 的立方根会是 2.8125，直接四舍五入就是 3 了，很明显这里 8 的平方根应该是 2。

直接舍弃小数？由于我们是近似解，所有 9 的平方根可能会是 2.999， 舍弃小数变成 2 ，很明显也是不对的。

这里我想到一个解法。

我们先采取四舍五入变成 ans，然后判断 ans * ans 是否大于 n，如果大于 n 了，ans 减 1。如果小于等于，那么 ans 不变。这样的话，求 8 的平方根的例子就被我们解决了。

```
int ans = (int) Math.round(mid); //先采取四舍五入
if ((long) ans * ans > n) {
    ans--;
}
// 可以不用 long
if (ans > n / ans) {
    ans--;
}
```

合起来就可以了。

```
//计算 x² - n
public double fx(double x, double n) {
    return x * x - n;
}

public int mySqrt(int n) {
    double low = 0;
    double high = n;
    double mid = (low + high) / 2;
    //函数值小于 0.1 的时候结束
    while (Math.abs(fx(mid, n)) > 0.1) {
        //左端点的函数值
        double low_f = fx(low, n);
        //中间节点的函数值
        double low_mid = fx(mid, n);
        //判断哪一段的函数值是异号的
        if (low_f * low_mid < 0) {
            high = mid;
        } else {
            low = mid;
        }
        mid = (low + high) / 2;
    }
    //先进行四舍五入
    int ans = (int) Math.round(mid);
    if (ans != 0 && ans > n / ans) {
        ans--;
    }
    return ans;
}
```

时间复杂度：

空间复杂度：O（1）。

解法三 牛顿法

具体解释可以参考下这篇文章，或者搜一下， 有很多讲解的，代码的话根据下边的迭代式进行写。

$$x_{k+1}=x_k- f(x_k)/f^{'}(x_k)$$。

这里的话，$$f(x_n) = x^2-n$$

$$x_{k+1}=x_k-(x_k^2-n)/2x_k=(x_k^2+n)/2x_k = (x_k + n /x_k)/2$$。

```
public int mySqrt(int n) {
    double t = n; // 赋一个初值
    while (Math.abs(t * t - n) > 0.1) {
        t = (n / t + t) / 2.0;
    }
    //先进行四舍五入
    int ans = (int) Math.round(t);
    //判断是否超出
    if ((long) ans * ans > n) {
        ans--;
    }
    return ans;
}
```

时间复杂度：

空间复杂度：O（1）。

总

首先用了正常的二分法，求出整数解。然后用常规的二分法、牛顿法求近似根，然后利用一个技巧转换为整数解。

70、题目描述（简单难度）

You are climbing a stair case. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Note: Given *n* will be a positive integer.

Example 1:

```
Input: 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

爬楼梯，每次走 1 个或 2 个台阶，n 层的台阶，总共有多少种走法。

解法一 暴力解法

用递归的思路想一下，要求 n 层的台阶的走法，由于一次走 1 或 2 个台阶，所以上到第 n 个台阶之前，一定是停留在第 n - 1 个台阶上，或者 n - 2 个台阶上。所以如果用 f (n) 代表 n 个台阶的走法。那么，

f (n) = f (n - 1) + f (n - 2)。

f (1) = 1，f (2) = 2 。

发现个神奇的事情，这就是斐波那契数列（Fibonacci sequence）。

直接暴力一点，利用递归写出来。

```java
public int climbStairs(int n) {
    return climbStairsN(n);
}

private int climbStairsN(int n) {
    if (n == 1) {
        return 1;
    }
    if (n == 2) {
```

```
        return 2;
    }
    return climbStairsN(n - 1) + climbStairsN(n - 2);
}
```

时间复杂度：是一个树状图，$$O(2^n)$$。

空间复杂度：

解法二 暴力解法优化

解法一很慢，leetcode 上报了超时，原因就是先求 climbStairsN (n - 1)，然后求 climbStairsN (n - 2) 的时候，其实很多解已经有了，但是它依旧进入了递归。优化方法就是把求出的解都存起来，后边求的时候直接使用，不用再进入递归了。叫做 memoization 技术。

```
public int climbStairs(int n) {
    return climbStairsN(n, new HashMap<Integer, Integer>());
}

private int climbStairsN(int n, HashMap<Integer, Integer> hashMap) {
    if (n == 1) {
        return 1;
    }
    if (n == 2) {
        return 2;
    }
    int n1 = 0;
    if (!hashMap.containsKey(n - 1)) {
        n1 = climbStairsN(n - 1, hashMap);
        hashMap.put(n - 1, n1);
    } else {
        n1 = hashMap.get(n - 1);
    }
    int n2 = 0;
    if (!hashMap.containsKey(n - 2)) {
        n2 = climbStairsN(n - 2, hashMap);
        hashMap.put(n - 2, n1);
    } else {
        n2 = hashMap.get(n - 2);
    }
    return n1 + n2;
}
```

时间复杂度：

空间复杂度：

当然由于 key 都是整数，我们完全可以用一个数组去存储，不需要 Hash。

```
public int climbStairs(int n) {
    int memo[] = new int[n + 1];
    return climbStairsN(n, memo);
}
private int climbStairsN(int n, int[] memo) {
    if (n == 1) {
        return 1;
    }
    if (n == 2) {
        return 2;
    }
    int n1 = 0;
    //数组的默认值是 0
    if (memo[n - 1] == 0) {
        n1 = climbStairsN(n - 1, memo);
        memo[n - 1] = n1;
    } else {
        n1 = memo[n - 1];
    }
    int n2 = 0;
    if (memo[n - 2] == 0) {
        n2 = climbStairsN(n - 2, memo);
        memo[n - 2] = n2;

    } else {
        n2 = memo[n - 2];
    }
    return n1 + n2;
}
```

解法三 迭代

当然递归可以解决，我们可以直接迭代，省去递归压栈的过程。初始值 f(1) 和 f(2)，然后可以求出 f(3)，然后求出 f(4) ... 直到 f(n)，一个循环就够了。其实就是动态规划的思想了。

```
public int climbStairs(int n) {
    int n1 = 1;
    int n2 = 2;
    if (n == 1) {
        return n1;
    }
```

```
    if (n == 2) {
        return n2;
    }
    //n1、n2 都后移一个位置
    for (int i = 3; i <= n; i++) {
        int temp = n2;
        n2 = n1 + n2;
        n1 = temp;
    }
    return n2;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

以上都是比较常规的方法，下边分享一下 Solution 里给出的其他解法。

解法四 矩阵相乘

Solution5叫做 Binets Method，它利用数学归纳法证明了一下，这里就直接用了，至于怎么想出来的，我也不清楚了。

定义一个矩阵 $$Q = \begin{matrix} 1 & 1 \\ 1 & 0 \end{matrix} $$ ，然后求 f (n) 话，我们先让 Q 矩阵求幂，然后取第一行第一列的元素就可以了，也就是 $$f(n)=Q^n[0][0]$$。

至于怎么更快的求幂，可以看 50 题的解法三。

```
public int climbStairs(int n) {
    int[][] Q = {{1, 1}, {1, 0}};
    int[][] res = pow(Q, n);
    return res[0][0];
}
public int[][] pow(int[][] a, int n) {
    int[][] ret = {{1, 0}, {0, 1}};
    while (n > 0) {
        //最后一位是 1，加到累乘结果里
        if ((n & 1) == 1) {
            ret = multiply(ret, a);
        }
        //n 右移一位
        n >>= 1;
        //更新 a
        a = multiply(a, a);
    }
    return ret;
}
```

```
public int[][] multiply(int[][] a, int[][] b) {
    int[][] c = new int[2][2];
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j];
        }
    }
    return c;
}
```

时间复杂度：O（log （n））。

空间复杂度：O（1）。

解法五 公式法

直接套用公式

$$F_n = 1/\sqrt{5}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

```
public int climbStairs(int n) {
    double sqrt5=Math.sqrt(5);
    double fibn=Math.pow((1+sqrt5)/2,n+1)-Math.pow((1-sqrt5)/2,n+1);
    return (int)(fibn/sqrt5);
}
```

时间复杂度：耗在了求幂的时候，O（log（n））。

空间复杂度：O（1）。

总

这道题把递归，动态规划的思想都用到了，很经典。此外，矩阵相乘的解法是真的强，直接将时间复杂度优化到 log 层面。

71、题目描述（中等难度）

71. Simplify Path

Medium 411 1151 Favorite Share

Given an **absolute path** for a file (Unix-style), simplify it. Or in other words, convert it to the **canonical path**.

In a UNIX-style file system, a period `.` refers to the current directory. Furthermore, a double period `..` moves the directory up a level. For more information, see: Absolute path vs relative path in Linux/Unix

Note that the returned canonical path must always begin with a slash `/`, and there must be only a single slash `/` between two directory names. The last directory name (if it exists) **must not** end with a trailing `/`. Also, the canonical path must be the **shortest** string representing the absolute path.

Example 1:

```
Input: "/home/"
Output: "/home"
Explanation: Note that there is no trailing slash after the last directory name.
```

Example 2:

```
Input: "/../"
Output: "/"
Explanation: Going one level up from the root directory is a no-op, as the root level is the highest level you can go.
```

Example 3:

```
Input: "/home//foo/"
Output: "/home/foo"
Explanation: In the canonical path, multiple consecutive slashes are replaced by a single one.
```

生成一个绝对路径，把相对路径中 ".." 和 "." 都转换为实际的路径，此外，"///" 多余的 "/" 要去掉，开头要有一个 "/"，末尾不要 "/"。

解法一

这道题，只要理解了题意，然后理一下就出来了。下面代码就不考虑空间复杂度了，多创建几个数组，代码会简洁一些。

```java
public String simplifyPath(String path) {
    //先利用 "/" 将字符串分割成一个一个单词
    String[] wordArr = path.split("/");
    //将空字符串（由类似这种"/a//c"的字符串产生）和 "." （"."代表当前目录不影响路径）去掉，保存到 wordList
    ArrayList<String> wordList = new ArrayList<String>();
    for (int i = 0; i < wordArr.length; i++) {
        if (wordArr[i].isEmpty() || wordArr[i].equals(".")) {
            continue;
        }
        wordList.add(wordArr[i]);
    }
    //wordListSim 保存简化后的路径
    ArrayList<String> wordListSim = new ArrayList<String>();
    //遍历 wordList
    for (int i = 0; i < wordList.size(); i++) {
        //如果遇到 "..", wordListSim 就删除末尾的单词
        if (wordList.get(i).equals("..")) {
            if (!wordListSim.isEmpty()) {
                wordListSim.remove(wordListSim.size() - 1);
            }
        //否则的话就加入 wordListSim
        } else {
            wordListSim.add(wordList.get(i));
        }
    }
    //将单词用 "/" 连接
    String ans = String.join("/", wordListSim);
    //开头补上 "/"
    ans = "/" + ans;
    return ans;

}
```

时间复杂度：

空间复杂度：

总

这道题就是理清思路就可以，没有用到什么技巧。

72、题目描述（困难难度）

72. Edit Distance

Hard 2005 30 Favorite Share

Given two words *word1* and *word2*, find the minimum number of operations required to convert *word1* to *word2*.

You have the following 3 operations permitted on a word:

1. Insert a character
2. Delete a character
3. Replace a character

Example 1:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

Example 2:

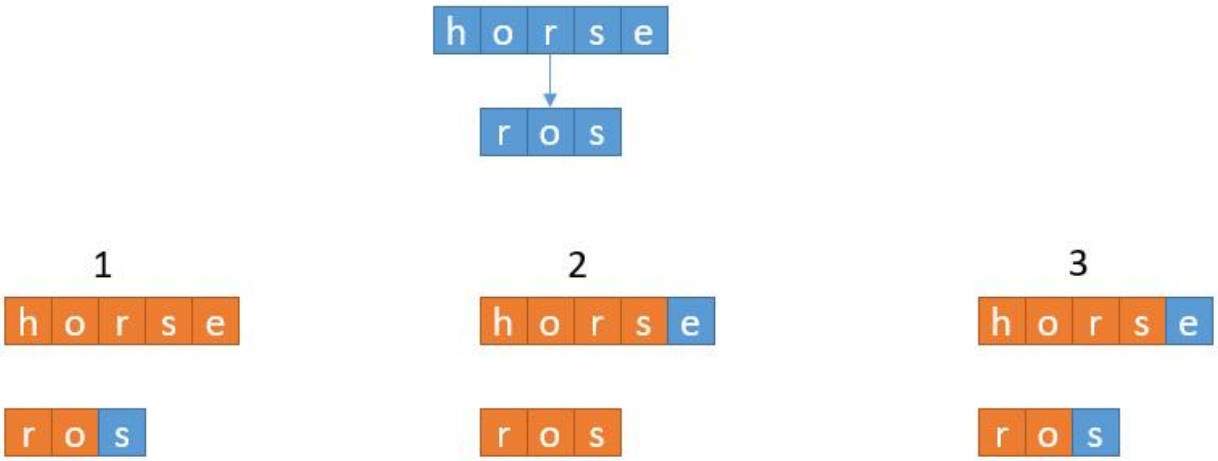
```
Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')
```

由一个字符串变为另一个字符串的最少操作次数，可以删除一个字符，替换一个字符，插入一个字符，也叫做最小编辑距离。

解法一 递归

我们可以发现删除一个字符和插入一个字符是等效的，对于变换次数并没有影响。例如 "a" 和 "ab" ，既可以 "a" 加上一个字符 "b" 变成 "ab"，也可以是 "ab" 去掉一个字符 "b" 变成 "a"。所以下边的算法可以只考虑删除和替换。

首先，以递归的思想去考虑问题，思考如何将大问题化解为小问题。例如 horse 变为 ros，其实我们有三种可选方案。



第一种，先把 horse 变为 ro ，求出它的最短编辑距离，假如是 x，然后 hosre 变成 ros 的编辑距离就可以是 x + 1。因为 horse 已经变成了 ro，然后我们可以把 ros 的 s 去掉，两个字符串就一样了，也就是再进行一次删除操作，所以加 1。

第二种，先把 hors 变为 ros，求出它的最短编辑距离，假如是 y，然后 hosre 变成 ros 的编辑距离就可以是 y + 1。因为 hors 变成了 ros，然后我们可以把 horse 的 e 去掉，两个字符串就一样了，也就是再进行一次删除操作，所以加 1。

第三种，先把 hors 变为 ro，求出它的最短编辑距离，假如是 z，然后我们再把 e 换成 s，两个字符串就一样了，hosre 变成 ros 的编辑距离就可以是 z + 1。当然，如果是其它的例子，最后一个字符是一样的，比如是 hosrs 和 ros ，此时我们直接取 z 作为编辑距离就可以了。

最后，我们从上边所有可选的编辑距离中，选一个最小的就可以了。

上边的第一种情况，假设了 horse 变为 ro 的最短编辑距离是 x，但其实我们并不知道 x 是多少，这个怎么求呢？类似的思路，也分为三种情况，然后选最小的就可以了！当然，上边的第二种，第三种情况也是类似的。然后一直递归下去。

最后，字符串长度不断地减少，直到出现了空串，这也是我们的递归出口了，如果是一个空串，一个不是空串，假如它的长度是 l，那么这两个字符串的最小编辑距离就是 l。如果是两个空串，那么最小编辑距离当然就是 0 了。

上边的分析，很容易就写出递归的写法了。

```
public int minDistance(String word1, String word2) {
    if (word1.length() == 0 && word2.length() == 0) {
        return 0;
    }
    if (word1.length() == 0) {
        return word2.length();
    }
    if (word2.length() == 0) {
        return word1.length();
```

```
    }
    int x = minDistance(word1, word2.substring(0, word2.length() - 1)) + 1;
    int y = minDistance(word1.substring(0, word1.length() - 1), word2) + 1;
    int z = minDistance(word1.substring(0, word1.length() - 1),
word2.substring(0, word2.length() - 1));
    if(word1.charAt(word1.length()-1)!=word2.charAt(word2.length()-1)){
        z++;
    }
    return Math.min(Math.min(x, y), z);
}
```

解法二 动态规划

上边的算法缺点很明显，先进行了压栈，浪费了很多时间，其次很多字符串的最小编辑距离都进行了重复计算。对于这种，很容易想到动态规划的思想去优化。

假设两个字符串是 word1 和 word2。

ans[i][j] 来表示字符串 word1[0, i ）（word1 的第 0 到 第 i - 1个字符）和 word2[0, j - 1) 的最小编辑距离。然后状态转移方程就出来了。

if (word1[m] == word2[n])

```
ans\[m\]\[n\] = Math.min ( ans[m]\[n-1\] + 1,  ans[m-1]\[n\] + 1,  ans[m-1]\[n-
1\])
```

if (word1[m] != word2[n])

```
ans\[m\]\[n\] = Math.min ( ans[m]\[n-1\] + 1,  ans[m-1]\[n\] + 1,  ans[m-1]\[n-
1\] + 1)
```

然后两层 for 循环，直接一层一层的更新数组就够了。

```
public int minDistance(String word1, String word2) {
    if (word1.length() == 0 && word2.length() == 0) {
        return 0;
    }
    if (word1.length() == 0) {
        return word2.length();
    }
    if (word2.length() == 0) {
        return word1.length();
    }
    int[][] ans = new int[word1.length() + 1][word2.length() + 1];
```

```
    //把有空串的情况更新了
    for (int i = 0; i <= word1.length(); i++) {
        ans[i][0] = i;
    }
    for (int i = 0; i <= word2.length(); i++) {
        ans[0][i] = i;
    }
    int n1 = word1.length();
    int n2 = word2.length();
    //从 1 开始遍历，从 0 开始的话，按照下边的算法取了 i - 1 会越界
    for (int i = 1; i <= n1; i++) {
        for (int j = 1; j <= n2; j++) {
            int min_delete = Math.min(ans[i - 1][j], ans[i][j - 1]) + 1;
            int replace = ans[i - 1][j - 1];
            if (word1.charAt(i - 1) != word2.charAt(j - 1)) {
                replace++;
            }
            ans[i][j] = Math.min(min_delete, replace);
        }
    }
    return ans[n1][n2];
}
```

时间复杂度：O（mn）。

空间复杂度：O（mn）。

如果你是顺序刷题的话，做到这里，一定会想到空间复杂度的优化，例如5题，10题，53题等等。主要想法是，看上边的算法，我们再求 ans[i][*] 的时候，我们只用到 ans[i - 1][*] 的情况，所以我们完全只用两个数组就够了。

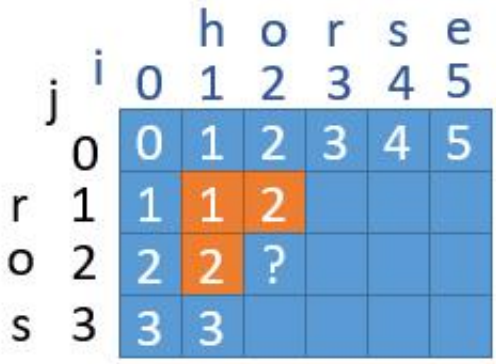
```
public int minDistance(String word1, String word2) {
    if (word1.length() == 0 && word2.length() == 0) {
        return 0;
    }
    if (word1.length() == 0) {
        return word2.length();
    }
    if (word2.length() == 0) {
        return word1.length();
    }
    int[][] ans = new int[2][word2.length() + 1];

    for (int i = 0; i <= word2.length(); i++) {
        ans[0][i] = i;
    }
```

```
    int n1 = word1.length();
    int n2 = word2.length();
    for (int i = 1; i <= n1; i++) {
        //由于只用了两个数组，所以不能向以前一样一次性初始化空串，在这里提前更新 j = 0 的情况
        ans[i % 2][0] = ans[(i - 1) % 2][0] + 1;
        for (int j = 1; j <= n2; j++) {
            int min_delete = Math.min(ans[(i - 1) % 2][j], ans[i % 2][j - 1]) +
1;
            int replace = ans[(i - 1) % 2][j - 1];
            if (word1.charAt(i - 1) != word2.charAt(j - 1)) {
                replace++;
            }
            ans[i % 2][j] = Math.min(min_delete, replace);
        }
    }
    return ans[n1 % 2][n2];
}
```

时间复杂度：O（mn）。

空间复杂度：O（n）。

再直接点，其实连两个数组我们都不需要，只需要一个数组。改写这个可能有些不好理解，可以结合一下图示。



在更新二维数组的时候，我们都是一列一列的更新。在更新？位置的时候，我们需要橙色位置的信息，也就是当前列的上一个位置，和上一列的当前位置，和上一列的上一个位置。如果我们用一个数组，当前列的上一个位置已经把上一列的上一个位置的数据覆盖掉了，所以我们要用一个变量提前保存上一列的上一个位置以便使用。

```
public int minDistance(String word1, String word2) {
    if (word1.length() == 0 && word2.length() == 0) {
        return 0;
    }
    if (word1.length() == 0) {
```

```
            return word2.length();
        }
        if (word2.length() == 0) {
            return word1.length();
        }
        int[] ans = new int[word2.length() + 1];

        for (int i = 0; i <= word2.length(); i++) {
            ans[i] = i;
        }
        int n1 = word1.length();
        int n2 = word2.length();
        for (int i = 1; i <= n1; i++) {
            int temp = ans[0];
            ans[0] = ans[0] + 1;
            for (int j = 1; j <= n2; j++) {
                int min_delete = Math.min(ans[j], ans[j - 1]) + 1;
                //上一列的上一个位置，直接用 temp
                int replace = temp;
                if (word1.charAt(i - 1) != word2.charAt(j - 1)) {
                    replace++;
                }
                //保存当前列的信息
                temp = ans[j];
                //再进行更新
                ans[j] = Math.min(min_delete, replace);
            }
        }
        return ans[n2];
    }
```

时间复杂度：O（mn）。

空间复杂度：O（n）。

总

动态规划的一系列操作，先递归，利用动态规划省略压栈的过程，然后空间复杂度的优化，很经典了。此外，对于动态规划数组的含义的定义也是很重要，开始的时候自己将 ans[i][j] 表示为 字符串 word1[0, i]（word1 的第 0 到 第 i 个字符）和 word2[0, j - 1] 的最小编辑距离。和上边解法的区别只是包含了末尾的字符。这造成了初始化 ans[0][*] 和 ans[*][0] 的时候，会比较复杂，看到了这里的解法，才有一种柳暗花明的感觉，思路是一样的，但更新ans[0][*] 和 ans[*][0] 却简单了很多。

73、题目描述（中等难度）

73. Set Matrix Zeroes

Medium 1024 190 Favorite Share

Given a *m* x *n* matrix, if an element is 0, set its entire row and column to 0. Do it **in-place**.

Example 1:

```
Input:
[
  [1,1,1],
  [1,0,1],
  [1,1,1]
]
Output:
[
  [1,0,1],
  [0,0,0],
  [1,0,1]
]
```

Example 2:

```
Input:
[
  [0,1,2,0],
  [3,4,5,2],
  [1,3,1,5]
]
Output:
[
  [0,0,0,0],
  [0,4,5,0],
  [0,3,1,0]
]
```

给定一个矩阵，然后找到所有含有 0 的地方，把该位置所在行所在列的元素全部变成 0。

解法一

暴力解法，用一个等大的空间把给定的矩阵存起来，然后遍历这个矩阵，遇到 0 就把原矩阵的当前行，当前列全部变作 0，然后继续遍历。

```java
public void setZeroes(int[][] matrix) {
    int row = matrix.length;
    int col = matrix[0].length;
    int[][] matrix_copy = new int[row][col];
    //复制矩阵
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
```

```
            matrix_copy[i][j] = matrix[i][j];
        }
    }
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            //找到 0 的位置
            if (matrix_copy[i][j] == 0) {
                //将当前行，当前列置为 0
                setRowZeroes(matrix, i);
                setColZeroes(matrix, j);
            }

        }
    }
}
//第 col 列全部置为 0
private void setColZeroes(int[][] matrix, int col) {
    for (int i = 0; i < matrix.length; i++) {
        matrix[i][col] = 0;
    }
}
//第 rol 行全部置为 0
private void setRowZeroes(int[][] matrix, int row) {
    for (int i = 0; i < matrix[row].length; i++) {
        matrix[row][i] = 0;
    }
}
```

时间复杂度：O (mn)。

空间复杂度：O（mn）。m 和 n 分别是矩阵的行数和列数。

解法二

空间复杂度可以优化一下，我们可以把哪一行有 0 ，哪一列有 0 都记录下来，然后最后统一把这些行，这些列置为 0。

```
public void setZeroes(int[][] matrix) {
    int row = matrix.length;
    int col = matrix[0].length;
    //用两个 bool 数组标记当前行和当前列是否需要置为 0
    boolean[] row_zero = new boolean[row];
    boolean[] col_zero = new boolean[col];
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            //找到 0 的位置
```

```
            if (matrix[i][j] == 0) {
                row_zero[i] = true;
                col_zero[j] = true;
            }
        }
    }
    //将行标记为 true 的行全部置为 0
    for (int i = 0; i < row; i++) {
        if (row_zero[i]) {
            setRowZeroes(matrix, i);
        }
    }
    //将列标记为 false 的列全部置为 0
    for (int i = 0; i < col; i++) {
        if (col_zero[i]) {
            setColZeroes(matrix, i);
        }
    }
}
//第 col 列全部置为 0
private void setColZeroes(int[][] matrix, int col) {
    for (int i = 0; i < matrix.length; i++) {
        matrix[i][col] = 0;
    }
}
//第 rol 行全部置为 0
private void setRowZeroes(int[][] matrix, int row) {
    for (int i = 0; i < matrix[row].length; i++) {
        matrix[row][i] = 0;
    }
}
```

时间复杂度：O (mn)。

空间复杂度：O（m + n）。m 和 n 分别是矩阵的行数和列数。

顺便说一下 leetcode 解法二说的解法，思想是一样的，只不过它没有用 bool 数组去标记，而是用两个 set 去存行和列。

```
class Solution {
    public void setZeroes(int[][] matrix) {
        int R = matrix.length;
        int C = matrix[0].length;
        Set<Integer> rows = new HashSet<Integer>();
        Set<Integer> cols = new HashSet<Integer>();
```

```
        // 将元素为 0 的地方的行和列存起来
        for (int i = 0; i < R; i++) {
            for (int j = 0; j < C; j++) {
                if (matrix[i][j] == 0) {
                    rows.add(i);
                    cols.add(j);
                }
            }
        }

        //将存储的 Set 拿出来，然后将当前行和列相应的元素置零
        for (int i = 0; i < R; i++) {
            for (int j = 0; j < C; j++) {
                if (rows.contains(i) || cols.contains(j)) {
                    matrix[i][j] = 0;
                }
            }
        }
    }
}
```

这里，有一个比自己巧妙的地方时，自己比较直接的用两个函数去将行和列分别置零，但很明显自己的算法会使得一些元素重复置零。而上边提供的算法，每个元素只遍历一次就够了，很棒。

解法三

继续优化空间复杂度，接下来用的思想之前也用过，例如41题解法二和47题解法二，就是用给定的数组去存我们需要的数据，只要保证原来的数据不丢失就可以。

按 47题解法二 的思路，就是假设我们对问题足够的了解，假设存在一个数，矩阵中永远不会存在，然后我们就可以把需要变成 0 的位置先变成这个数，也就是先标记一下，最后再统一把这个数变成 0。直接贴下leetcode解法二的代码。

```
class Solution {
    public void setZeroes(int[][] matrix) {
        int MODIFIED = -1000000; //假设这个数字不存在于矩阵中
        int R = matrix.length;
        int C = matrix[0].length;

        for (int r = 0; r < R; r++) {
            for (int c = 0; c < C; c++) {
                //找到等于 0 的位置
                if (matrix[r][c] == 0) {
                    // 将需要变成 0 的行和列改为之前定义的数字
                    // 如果是 0 不要管，因为我们要找 0 的位置
```

```
                    for (int k = 0; k < C; k++) {
                        if (matrix[r][k] != 0) {
                            matrix[r][k] = MODIFIED;
                        }
                    }
                    for (int k = 0; k < R; k++) {
                        if (matrix[k][c] != 0) {
                            matrix[k][c] = MODIFIED;
                        }
                    }
                }
            }
        }

        for (int r = 0; r < R; r++) {
            for (int c = 0; c < C; c++) {
                // 将是定义的数字的位置变成 0
                if (matrix[r][c] == MODIFIED) {
                    matrix[r][c] = 0;
                }
            }
        }
    }
}
```

时间复杂度：O (mn)。

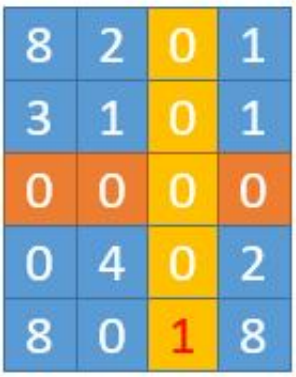
空间复杂度：O（1）。m 和 n 分别是矩阵的行数和列数。

当然，这个解法局限性很强，很依赖于样例的取值，我们继续想其他的方法。

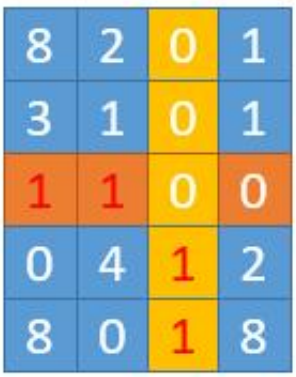
回想一下解法二，我们用了两个 bool 数组去标记当前哪些行和那些列需要置零，我们能不能在矩阵中找点儿空间去存我们的标记呢?

可以的！因为当我们找到第一个 0 的时候，这个 0 所在行和所在列就要全部更新成 0，所以它之前的数据是什么就不重要了，所以我们可以把这一行和这一列当做标记位，0 当做 false，1 当做 true，最后像解法二一样，统一更新就够了。

如上图，找到第一个 0 出现的位置，把橙色当做解法二的列标志位，黄色当做解法二的行标志位。



如上图，我们首先需要初始化为 0，并且遇到之前是 0 的位置我们需要把它置为 1，代表当前行（或者列）最终要值为 0。



如上图，继续遍历找 0 的位置，找到后将对应的位置置为 1 即可。橙色部分的数字为 1 代表当前列要置为 0，黄色部分的数字为 1 代表当前行要置为 0。

看下代码吧。

```
public void setZeroes(int[][] matrix) {
```

```
    int row = matrix.length;
    int col = matrix[0].length;
    int free_row = -1; //记录第一个 0 出现的行
    int free_col = -1; //记录第一个 0 出现的列
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            //如果是当前作为标记的列，就跳过
            if (j == free_col) {
                continue;
            }
            if (matrix[i][j] == 0) {
                //判断是否是第一个 0
                if (free_row == -1) {
                    free_row = i;
                    free_col = j;
                    //初始化行标记位为 0，如果之前是 0 就置为 1
                    for (int k = 0; k < matrix.length; k++) {
                        if (matrix[k][free_col] == 0) {
                            matrix[k][free_col] = 1;
                        } else {
                            matrix[k][free_col] = 0;
                        }

                    }
                    //初始化列标记位为 0，如果之前是 0 就置为 1
                    for (int k = 0; k < matrix[free_row].length; k++) {
                        if (matrix[free_row][k] == 0) {
                            matrix[free_row][k] = 1;
                        } else {
                            matrix[free_row][k] = 0;
                        }
                    }
                    break;
                 //找 0 的位置，将相应的标志置 1
                } else {
                    matrix[i][free_col] = 1;
                    matrix[free_row][j] = 1;
                }
            }

        }
    }
    if (free_row != -1) {
        //将标志位为 1 的所有列置为 0
        for (int i = 0; i < col; i++) {
            if (matrix[free_row][i] == 1) {
```

```
                setColZeroes(matrix, i);
            }
        }
        //将标志位为 1 的所有行置为 0
        for (int i = 0; i < row; i++) {
            if (matrix[i][free_col] == 1) {
                setRowZeroes(matrix, i);
            }
        }
    }
}

private void setColZeroes(int[][] matrix, int col) {
    for (int i = 0; i < matrix.length; i++) {
        matrix[i][col] = 0;
    }
}

private void setRowZeroes(int[][] matrix, int row) {
    for (int i = 0; i < matrix[row].length; i++) {
        matrix[row][i] = 0;
    }
}
```

时间复杂度：O (mn)。

空间复杂度：O（1）。

leetcode解法三和我的思想是一样的，它标记位直接用第一行和第一列，由于第一行和第一列不一定会被置为 0，所以需要用 isCol 变量来标记第一列是否需要置为 0，用 matrix[0][0] 标记第一行是否需要置为 0。它是将用 0 表示当前行（列）需要置 0，这一点也很巧妙，相比我上边的算法就不需要初始化标记位了。

```
class Solution {
    public void setZeroes(int[][] matrix) {
        Boolean isCol = false;
        int R = matrix.length;
        int C = matrix[0].length;
        for (int i = 0; i < R; i++) {
            //判断第 1 列是否需要置为 0
            if (matrix[i][0] == 0) {
                isCol = true;
            }
            //找 0 的位置，将相应标记置 0
            for (int j = 1; j < C; j++) {
                if (matrix[i][j] == 0) {
                    matrix[0][j] = 0;
```

```
                    matrix[i][0] = 0;
                }
            }
        }
        //根据标志，将元素置 0
        for (int i = 1; i < R; i++) {
            for (int j = 1; j < C; j++) {
                if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }

        //判断第一行是否需要置 0
        if (matrix[0][0] == 0) {
            for (int j = 0; j < C; j++) {
                matrix[0][j] = 0;
            }
        }

        //判断第一列是否需要置 0
        if (isCol) {
            for (int i = 0; i < R; i++) {
                matrix[i][0] = 0;
            }
        }
    }
}
```

总

这道题如果对空间复杂度没有要求就很简单了，对于空间复杂度的优化，充分利用给定的空间的思想很经典了。

74、题目描述（中等难度）

74. Search a 2D Matrix

Medium 811 97 Favorite Share

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

Example 1:

```
Input:
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 3
Output: true
```

Example 2:

```
Input:
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 13
Output: false
```

判断一个矩阵中是否存在某个数，矩阵是有序的。

解法一 二分法

看到了有序序列，啥都不用想直接二分，只需要考虑到怎么把二分时候的下标转换为矩阵的行、列下标就可以了，很简单，用除法和求余就够了。

```
public boolean searchMatrix(int[][] matrix, int target) {
    int rows = matrix.length;
    if (rows == 0) {
        return false;
    }
    int cols = matrix[0].length;
    int left = 0;
    int right = rows * cols - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        int temp = matrix[mid / cols][mid % cols];
```

```
            if (temp == target) {
                return true;
            } else if (temp < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return false;
    }
```

时间复杂度：O (log (n))。

空间复杂度：O (1)。

总

这道题的二分法，比较简单，大家可以看下33题，相信对二分法会有一个更深刻的理解。

75、题目描述（中等难度）

75. Sort Colors

Medium 1596 154 Favorite Share

Given an array with *n* objects colored red, white or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note: You are not suppose to use the library's sort function for this problem.

Example:

```
Input: [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
```

Follow up:

- A rather straight forward solution is a two-pass algorithm using counting sort.
  First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.
- Could you come up with a one-pass algorithm using only constant space?

给一个数组，含有的数只可能 0，1，2 中的一个，然后把这些数字从小到大排序。

解法一

题目下边的 Follow up 提到了一个解法，遍历一次数组，统计 0 出现的次数，1 出现的次数，2 出现的次数，然后再遍历数组，根据次数，把数组的元素改成相应的值。当然我们只需要记录 0 的次数，和 1 的次数，剩下的就是 2 的次数了。

```
public void sortColors(int[] nums) {
    int zero_count = 0;
    int one_count = 0;
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] == 0) {
            zero_count++;
        }
        if (nums[i] == 1) {
            one_count++;
        }
    }
    for (int i = 0; i < nums.length; i++) {
        if (zero_count > 0) {
            nums[i] = 0;
            zero_count--;
        } else if (one_count > 0) {
            nums[i] = 1;
            one_count--;
        } else {
            nums[i] = 2;
        }
    }
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

解法二

上边的算法，我们遍历了两次数组，让我们想一想只遍历一次的方法。我们假设一种简单的情况，如果只含有两个数 0 和 1，该怎么做呢?

假设原数组是 1 0 1 1 0，我们可以用一个指针，zero_position，含义是该指针指向的位置，前边的位置全部存 0 。然后再用一个指针 i 遍历这个数组，找到 0 就把 0 放到当前 zero_position 指向的位置，zero_position 后移。用 Z 代表 zero_position，看下边的遍历过程。

```
1 0 1 1 0   初始化 z,i 指向第 0 个位置，i 后移
^
Z
i
```

```
1 0 1 1 0   发现 0，把 z 的位置置为 0，并且把 z 的位置的数字交换过来，z 后移一位
^ ^
Z i

0 1 1 1 0   i 后移一位
  ^
  i
  Z

0 1 1 1 0  i 继续后移
  ^ ^
  Z i

0 1 1 1 0  i 继续后移
  ^   ^
  Z   i

0 1 1 1 0  遇到 0，把 z 的位置置为 0，并且把 z 的位置的数字交换过来，z 后移一位
  ^     ^
  Z     i

0 0 1 1 1  遍历结束
    ^   ^
    Z   i
```

回到我们当前这道题，我们有 3 个数字，那我们可以用两个指针，一个是 zero_position，和之前一样，它前边的位置全部存 0。再来一个指针，two_position，注意这里是，它**后边**的位置全部存 2。然后遍历整个数组就行了。

下边画一个遍历过程中的图，理解一下，Z 前边全存 0，T 后边全存 2。

```
0 1 0 2 1 2 2 2
  ^ ^   ^
  Z i   T
```

```java
public void sortColors(int[] nums) {
    int zero_position = 0;
    int two_position = nums.length - 1;
    for (int i = 0; i <= two_position; i++) {
        if (nums[i] == 0) {
            //将当前位置的数字保存
            int temp = nums[zero_position];
```

```
                //把 0 存过来
                nums[zero_position] = 0;
                //把之前的数换过来
                nums[i] = temp;
                //当前指针后移
                zero_position++;
            } else if (nums[i] == 2) {
                //将当前位置的数字保存
                int temp = nums[two_position];
                //把 2 存过来
                nums[two_position] = 2;
                //把之前的数换过来
                nums[i] = temp;
                //当前指针前移
                two_position--;
                //这里一定要注意，因为我们把后边的数字换到了第 i 个位置，
                //这个数字我们还没有判断它是多少，外层的 for 循环会使得 i++ 导致跳过这个元素
                //所以要 i--
                //而对于上边 zero_position 的更新不需要考虑，因为它是从前边换过来的数字
                //在之前已经都判断过了
                i--;
            }
        }
    }
```

时间复杂度：O（n）。

空间复杂度：O（1）。

解法三

解法二中总共有三种数，然后很自然可以分成三部分，用两个指针作为间隔，但是，如果有 5 种数呢，解法二恐怕就不适用了。在 leetcode 发现另一种解法，参考这里的解法二，用了大问题化小的思想。

我们用三个指针 n0，n1，n2，分别代表已排好序的数组当前 0 的末尾，1 的末尾，2 的末尾。

```
0  0  1  2  2  2  0  2  1
   ^  ^        ^  ^
  n0 n1       n2  i
```

然后当前遍历到 i 的位置，等于 0，我们只需要把 n2 指针后移并且将当前数字置为 2，将 n1 指针后移并且将当前数字置为 1，将 n0 指针后移并且将当前数字置为 0。

```
0  0  1  2  2  2  2  2  1   n2 后移后的情况
   ^  ^           ^
```

```
   n0 n1          i
                  n2

0  0  1  1  2  2  2  2  1  n1 后移后的情况
   ^     ^        ^
   n0    n1       i
                  n2

0  0  0  1  2  2  2  2  1  n0 后移后的情况
      ^  ^        ^
      n0 n1       i
                  n2
```

然后就达到了将 i 指向的 0 插入到当前排好序的 0 的位置的末尾。

原因的话，由于前边插入了新的数字，势必造成数字的覆盖，指针后移后要把对应的指针位置置为对应的数，n2 指针后移后置为 2，n1 指针后移后置为 1，例如，假如之前有 3 个 2，由于前边插入一个数字，所以会导致 1 个 2 被覆盖掉，所以要加 1 个 2。

```
public void sortColors(int[] nums) {
    int n0 = -1, n1 = -1, n2 = -1;
    int n = nums.length;
    for (int i = 0; i < n; i++) {
        if (nums[i] == 0) {
            n2++;
            nums[n2] = 2;
            n1++;
            nums[n1] = 1;
            n0++;
            nums[n0] = 0;
        } else if (nums[i] == 1) {
            n2++;
            nums[n2] = 2;
            n1++;
            nums[n1] = 1;
        } else if (nums[i] == 2) {
            n2++;
            nums[n2] = 2;
        }
    }
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

总

解法二利用指针，在原来的空间存东西很经典。解法三，其实本质是我们常用的递归思想，先假设一个小问题解决了，然后假如再来一个数该怎么操作。

76、题目描述（困难难度）

76. Minimum Window Substring

Hard 2237 156 Favorite Share

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

Example:

```
Input: S = "ADOBECODEBANC", T = "ABC"
Output: "BANC"
```

Note:

- If there is no such window in S that covers all characters in T, return the empty string `""`.
- If there is such window, you are guaranteed that there will always be only one unique minimum window in S.

给两个字符串，S 和 T，在 S 中找出包含 T 中所有字母的最短字符串，不考虑顺序。

解法一 滑动窗口

没有想出来，直接看来了题解，这里总结一下。

用双指针 left 和 right 表示一个窗口。

1. right 向右移增大窗口，直到窗口包含了所有要求的字母。进行第二步。
2. 记录此时的长度，left 向右移动，开始减少长度，每减少一次，就更新最小长度。直到当前窗口不包含所有字母，回到第 1 步。

```
S = "ADOBECODEBANC", T = "ABC"
A D O B E C O D E B A N C //l 和 r 初始化为 0
^
l
r
```

```
A D O B E C O D E B A N C //向后移动 r，扩大窗口
^ ^
l r

A D O B E C O D E B A N C //向后移动 r，扩大窗口
^   ^
l   r

A D O B E C O D E B A N C //向后移动 r，扩大窗口
^     ^
l     r

A D O B E C O D E B A N C //向后移动 r，扩大窗口
^       ^
l       r


//此时窗口中包含了所有字母（ABC），停止移动 r，记录此时的 l 和 r，然后开始移动 l
A D O B E C O D E B A N C
^         ^
l         r

//向后移动 l，减小窗口，此时窗口中没有包含所有字母（ABC），重新开始移动 r，扩大窗口
A D O B E C O D E B A N C
  ^       ^
  l       r

//移动 r 直到窗口包含了所有字母（ABC），
//和之前的长度进行比较，如果窗口更小，则更新 l 和 r
//然后开始移动 l，开始缩小窗口
A D O B E C O D E B A N C
  ^               ^
  l               r

//此时窗口内依旧包含所有字母
//和之前的长度进行比较，如果窗口更小，则更新 l 和 r
//继续移动 l，继续缩小窗口
A D O B E C O D E B A N C
    ^             ^
    l             r

//此时窗口内依旧包含所有字母
//和之前的长度进行比较，如果窗口更小，则更新 l 和 r
//继续移动 l，继续缩小窗口
A D O B E C O D E B A N C
      ^           ^
```

```
      l             r

//继续减小 l，直到窗口中不再包含所有字母，然后开始移动 r，不停的重复上边的过程，直到全部遍历完
```

思想有了，其实这里需要解决的问题只有一个，怎么来判断当前窗口包含了所有字母。

判断字符串相等，并且不要求顺序，之前已经用过很多次了，利用 HashMap，对于两个字符串 S = "ADOBECODEBANC", T = "ABCB"，用 map 统计 T 的每个字母的出现次数，然后遍历 S，遇到相应的字母，就将相应字母的次数减 1，如果此时 map 中所有字母的次数都小于等于 0，那么此时的窗口一定包含了所有字母。

```
public String minWindow(String s, String t) {
    Map<Character, Integer> map = new HashMap<>();
    //遍历字符串 t，初始化每个字母的次数
    for (int i = 0; i < t.length(); i++) {
        char char_i = t.charAt(i);
        map.put(char_i, map.getOrDefault(char_i, 0) + 1);
    }
    int left = 0; //左指针
    int right = 0; //右指针
    int ans_left = 0; //保存最小窗口的左边界
    int ans_right = -1; //保存最小窗口的右边界
    int ans_len = Integer.MAX_VALUE; //当前最小窗口的长度
    //遍历字符串 s
    while (right < s.length()) {
        char char_right = s.charAt(right);
        //判断 map 中是否含有当前字母
        if (map.containsKey(char_right)) {
            //当前的字母次数减一
            map.put(char_right, map.get(char_right) - 1);
            //开始移动左指针，减小窗口
            while (match(map)) { //如果当前窗口包含所有字母，就进入循环
                //当前窗口大小
                int temp_len = right - left + 1;
                //如果当前窗口更小，则更新相应变量
                if (temp_len < ans_len) {
                    ans_left = left;
                    ans_right = right;
                    ans_len = temp_len;
                }
                //得到左指针的字母
                char key = s.charAt(left);
                //判断 map 中是否有当前字母
                if (map.containsKey(key)) {
                    //因为要把当前字母移除，所有相应次数要加 1
                    map.put(key, map.get(key) + 1);
```

```
                }
                left++; //左指针右移
            }
        }
        //右指针右移扩大窗口
        right++;
    }
    return s.substring(ans_left, ans_right+1);
}

//判断所有的 value 是否为 0
private boolean match(Map<Character, Integer> map) {
    for (Integer value : map.values()) {
        if (value > 0) {
            return false;
        }
    }
    return true;
}
```

时间复杂度：O（nm），n 是 S 的长度，match 函数消耗 O（m）。

空间复杂度：O（m），m 是 T 的长度。

参考这里，由于字符串中只有字母，我们其实可以不用 hashmap，可以直接用一个 int 数组，字母的 ascii 码值作为下标，保存每个字母的次数。

此外，判断当前窗口是否含有所有字母，我们除了可以判断所有字母的次数是否小于等于 0，还可以用一个计数变量 count，把 count 初始化为 t 的长度，然后每次找到一个满足条件的字母，count 就减 1，如果 count 等于了 0，就代表包含了所有字母。这样的话，可以把之前的 match(map) 优化到 O（1）。

```
public String minWindow(String s, String t) {
    int[] map = new int[128];
    // 遍历字符串 t，初始化每个字母的次数
    for (int i = 0; i < t.length(); i++) {
        char char_i = t.charAt(i);
        map[char_i]++;
    }
    int left = 0; // 左指针
    int right = 0; // 右指针
    int ans_left = 0; // 保存最小窗口的左边界
    int ans_right = -1; // 保存最小窗口的右边界
    int ans_len = Integer.MAX_VALUE; // 当前最小窗口的长度
    int count = t.length();
    // 遍历字符串 s
    while (right < s.length()) {
        char char_right = s.charAt(right);
```

```
            // 当前的字母次数减一
            map[char_right]--;

            //代表当前符合了一个字母
            if (map[char_right] >= 0) {
                count--;
            }
            // 开始移动左指针，减小窗口
            while (count == 0) { // 如果当前窗口包含所有字母，就进入循环
                // 当前窗口大小
                int temp_len = right - left + 1;
                // 如果当前窗口更小，则更新相应变量
                if (temp_len < ans_len) {
                    ans_left = left;
                    ans_right = right;
                    ans_len = temp_len;
                }
                // 得到左指针的字母
                char key = s.charAt(left);
                // 因为要把当前字母移除，所有相应次数要加 1
                map[key]++;
                //此时的 map[key] 大于 0 了，表示缺少当前字母了，count++
                if (map[key] > 0) {
                    count++;
                }
                left++; // 左指针右移
            }
            // 右指针右移扩大窗口
            right++;
        }
        return s.substring(ans_left, ans_right + 1);
    }
```

总

开始自己的思路偏了，一直往递归，动态规划的思想走，导致没想出来。对滑动窗口的应用的少，这次加深了印象。

77、题目描述（中等难度）

77. Combinations

Medium 765 42 Favorite Share

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.

Example:

```
Input: n = 4, k = 2
Output:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

给定 n ， k ，表示从 { 1, 2, 3 ... n } 中选 k 个数，输出所有可能，并且选出数字从小到大排列，每个数字只能用一次。

解法一 回溯法

这种选数字很经典的回溯法问题了，先选一个数字，然后进入递归继续选，满足条件后加到结果中，然后回溯到上一步，继续递归。直接看代码吧，很好理解。

```
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> ans = new ArrayList<>();
    getAns(1,n, k, new ArrayList<Integer>(), ans);
    return ans;
}

private void getAns(int start, int n, int k, ArrayList<Integer>
temp,List<List<Integer>> ans) {
    //如果 temp 里的数字够了 k 个，就把它加入到结果中
    if(temp.size() == k){
        ans.add(new ArrayList<Integer>(temp));
        return;
    }
    //从 start 到 n
    for (int i = start; i <= n; i++) {
        //将当前数字加入 temp
        temp.add(i);
        //进入递归
        getAns(i+1, n, k, temp, ans);
```

```
            //将当前数字删除，进入下次 for 循环
            temp.remove(temp.size() - 1);
        }
    }
```

一个 for 循环，添加，递归，删除，很经典的回溯框架了。在这里发现了一个优化方法。for 循环里 i 从 start 到 n，其实没必要到 n。比如，n = 5，k = 4，temp.size( ) == 1，此时代表我们还需要（4 - 1 = 3）个数字，如果 i = 4 的话，以后最多把 4 和 5 加入到 temp 中，而此时 temp.size() 才等于 1 + 2 = 3，不够 4 个，所以 i 没必要等于 4，i 循环到 3 就足够了。

所以 for 循环的结束条件可以改成， i <= n - (k - temp.size ( )) + 1，k - temp.size ( ) 代表我们还需要的数字个数。因为我们最后取到了 n，所以还要加 1。

```
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> ans = new ArrayList<>();
    getAns(1,n, k, new ArrayList<Integer>(), ans);
    return ans;
}

private void getAns(int start, int n, int k, ArrayList<Integer> temp,
List<List<Integer>> ans) {
    if(temp.size() == k){
        ans.add(new ArrayList<Integer>(temp));
        return;
    }
    for (int i = start; i <= n - (k -temp.size()) + 1; i++) {
        temp.add(i);
        getAns(i+1, n, k, temp, ans);
        temp.remove(temp.size() - 1);
    }
}
```

虽然只改了一句代码，速度却快了很多。

解法二 迭代

参考这里，完全按照解法一回溯的思想改成迭代。我们思考一下，回溯其实有三个过程。

- for 循环结束，也就是 i == n + 1，然后回到上一层的 for 循环
- temp.size() == k，也就是所需要的数字够了，然后把它加入到结果中。
- 每个 for 循环里边，进入递归，添加下一个数字

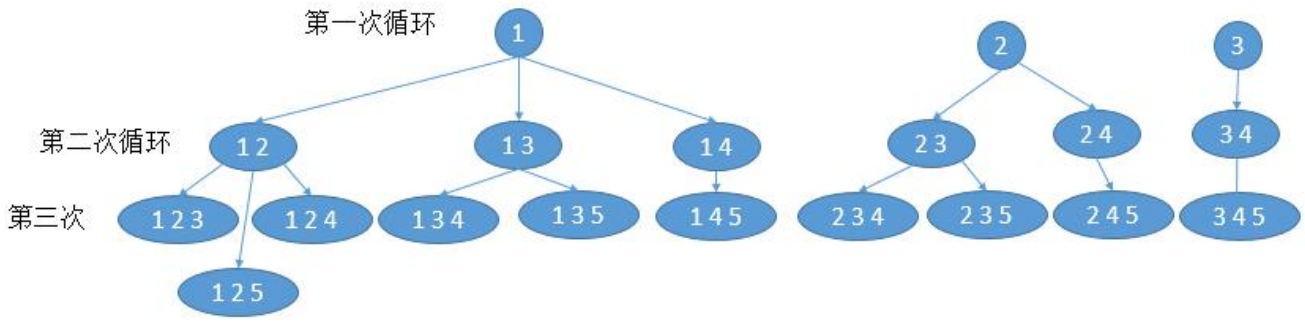
```
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> ans = new ArrayList<>();
    List<Integer> temp = new ArrayList<>();
```

```
        for(int i = 0;i<k;i++){
            temp.add(0);
        }
        int i = 0;
        while (i >= 0) {
            temp.set(i, temp.get(i)+ 1); //当前数字加 1
            //当前数字大于 n，对应回溯法的 i == n + 1，然后回到上一层
            if (temp.get(i) > n) {
                i--;
            // 当前数字个数够了
            } else if (i == k - 1) {
                ans.add(new ArrayList<>(temp));
            //进入更新下一个数字
            }else {
                i++;
                //把下一个数字置为上一个数字，类似于回溯法中的 start
                temp.set(i, temp.get(i-1));
            }
        }
        return ans;
    }
```

解法三 迭代法2

解法二的迭代法是基于回溯的思想，还有一种思想，参考这里。类似于46题的解法一，找 k 个数，我们可以先找出 1 个的所有结果，然后在 1 个的所有结果再添加 1 个数，变成 2 个，然后依次迭代，直到有 k 个数。

比如 n = 5， k = 3



第 1 次循环，我们找出所有 1 个数的可能 [1]，[2]，[3]。4 和 5 不可能，解法一分析过了，因为总共需要 3 个数，4，5 全加上才 2 个数。

第 2 次循环，在每个 list 添加 1 个数， [1] 扩展为 [1 , 2]，[1 , 3]，[1 , 4]。[1 , 5] 不可能，因为 5 后边没有数字了。 [2] 扩展为 [2 , 3]，[2 , 4]。[3] 扩展为 [3 , 4];

第 3 次循环，在每个 list 添加 1 个数，[1，2] 扩展为[1，2，3]，[1，2，4]，[1，2，5]；[1，3] 扩展为 [1，3，4]，[1，3，5]；[1，4] 扩展为 [1，4，5]；[2，3] 扩展为 [2，3，4]，[2，3，5]；[2，4] 扩展为 [2，4，5]；[3，4] 扩展为 [3，4，5]；

最后结果就是，[[1，2，3]，[1，2，4]，[1，2，5]，[1，3，4]，[1，3，5]，[1，4，5]，[2，3，4]，[2，3，5]，[2，4，5]，[3，4，5]]。

上边分析很明显了，三个循环，第一层循环是 1 到 k，代表当前有多少个数。第二层循环就是遍历之前的所有结果。第三次循环就是将当前结果扩展为多个。

```
public List<List<Integer>> combine(int n, int k) {
    if (n == 0 || k == 0 || k > n) return Collections.emptyList();
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    //个数为 1 的所有可能
    for (int i = 1; i <= n + 1 - k; i++) res.add(Arrays.asList(i));
    //第一层循环，从 2 到 k
    for (int i = 2; i <= k; i++) {
        List<List<Integer>> tmp = new ArrayList<List<Integer>>();
        //第二层循环，遍历之前所有的结果
        for (List<Integer> list : res) {
            //第三次循环，对每个结果进行扩展
            //从最后一个元素加 1 开始，然后不是到 n ，而是和解法一的优化一样
            //(k - (i - 1)) 代表当前已经有的个数，最后再加 1 是因为取了 n
            for (int m = list.get(list.size() - 1) + 1; m <= n - (k - (i - 1)) +
1; m++) {
                List<Integer> newList = new ArrayList<Integer>(list);
                newList.add(m);
                tmp.add(newList);
            }
        }
        res = tmp;
    }
    return res;
}
```

解法四 递归

参考这里。基于这个公式 C(n, k) = C(n - 1, k - 1) + C(n - 1, k) 所用的思想，这个思想之前刷题也用过，但忘记是哪道了。

从 n 个数字选 k 个，我们把所有结果分为两种，包含第 n 个数和不包含第 n 个数。这样的话，就可以把问题转换成

- 从 n - 1 里边选 k - 1 个，然后每个结果加上 n
- 从 n - 1 个里边直接选 k 个。

把上边两个的结果合起来就可以了。

```
public List<List<Integer>> combine(int n, int k) {
    if (k == n || k == 0) {
        List<Integer> row = new LinkedList<>();
        for (int i = 1; i <= k; ++i) {
            row.add(i);
        }
        return new LinkedList<>(Arrays.asList(row));
    }
    // n - 1 里边选 k - 1 个
    List<List<Integer>> result = combine(n - 1, k - 1);
    //每个结果加上 n
    result.forEach(e -> e.add(n));
    //把 n - 1 个选 k 个的结果也加入
    result.addAll(combine(n - 1, k));
    return result;
}
```

解法五 动态规划

参考这里，既然有了解法四的递归，那么一定可以有动态规划。递归就是压栈压栈压栈，然后到了递归出口，开始出栈出栈出栈。而动态规划一个好处就是省略了出栈的过程，我们直接从递归出口网上走。

```
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>>[][] dp = new List[n + 1][k + 1];
    //更新 k = 0 的所有情况
    for (int i = 0; i <= n; i++) {
        dp[i][0] = new ArrayList<>();
        dp[i][0].add(new ArrayList<Integer>());
    }
    // i 从 1 到 n
    for (int i = 1; i <= n; i++) {
        // j 从 1 到 i 或者 k
        for (int j = 1; j <= i && j <= k; j++) {
            dp[i][j] = new ArrayList<>();
            //判断是否可以从 i - 1 里边选 j 个
            if (i > j){
                dp[i][j].addAll(dp[i - 1][j]);
            }
            //把 i - 1 里边选 j - 1 个的每个结果加上 i
            for (List<Integer> list: dp[i - 1][j - 1]) {
                List<Integer> tmpList = new ArrayList<>(list);
                tmpList.add(i);
                dp[i][j].add(tmpList);
            }
```

```
        }
    }
    return dp[n][k];
}
```

这里遇到个神奇的问题，提一下，开始的的时候，最里边的 for 循环是这样写的

```
for (List<Integer> list: dp[i - 1][j - 1]) {
    List<Integer> tmpList = new LinkedList<>(list);
    tmpList.add(i);
    dp[i][j].add(tmpList);
}
```

就是 List 用的 Linked，而不是 Array，看起来没什么大问题，在 leetcode 上竟然报了超时。看了下 java 的源码。

```
//ArrayList
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
//LinkedList
public boolean add(E e) {
    linkLast(e);
    return true;
}
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

猜测原因可能是因为 linked 每次 add 的时候，都需要 new 一个节点对象，而我们进行了很多次 add，所以这里造成了时间的耗费，导致了超时。所以刷题的时候还是优先用 ArrayList 吧。

接下来就是动态规划的常规操作了，空间复杂度的优化，我们注意到更新 dp [i] [*] 的时候，只用到dp [i - 1] [*] 的情况，所以我们可以只用一个一维数组就够了。和72题解法二，以及5题，10题，53题等等优化思路一样，这里不详细说了。

```java
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>>[] dp = new ArrayList[k + 1];
    // i 从 1 到 n
    dp[0] = new ArrayList<>();
    dp[0].add(new ArrayList<Integer>());
    for (int i = 1; i <= n; i++) {
        // j 从 1 到 i 或者 k
        List<List<Integer>> temp = new ArrayList<>(dp[0]);
        for (int j = 1; j <= i && j <= k; j++) {
            List<List<Integer>> last = temp;
            if(dp[j]!=null){
                temp = new ArrayList<>(dp[j]);
            }
            // 判断是否可以从 i - 1 里边选 j 个
            if (i <= j) {
                dp[j] = new ArrayList<>();
            }
            // 把 i - 1 里边选 j - 1 个的每个结果加上 i
            for (List<Integer> list : last) {
                List<Integer> tmpList = new ArrayList<>(list);
                tmpList.add(i);
                dp[j].add(tmpList);
            }
        }
    }
    return dp[k];
}
```

总

开始的时候直接用了动态规划，然后翻了一些 Discuss 感觉发现了新世界，把目前为止常用的思路都用到了，回溯，递归，迭代，动态规划，这道题也太经典了！值得细细回味。

78、题目描述（中等难度）

78. Subsets

Medium 1895 50 Favorite Share

Given a set of **distinct** integers, *nums*, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

Example:

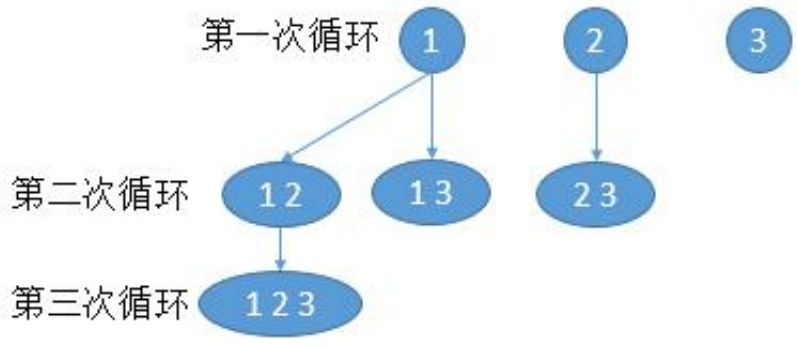
```
Input: nums = [1,2,3]
Output:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

给一个数组，输出这个数组的所有子数组。

解法一 迭代一

和 77 题解法三一个思想，想找出数组长度 1 的所有解，然后再在长度为 1 的所有解上加 1 个数字变成长度为 2 的所有解，同样的直到 n。

假如 nums = [1, 2, 3]，参照下图。



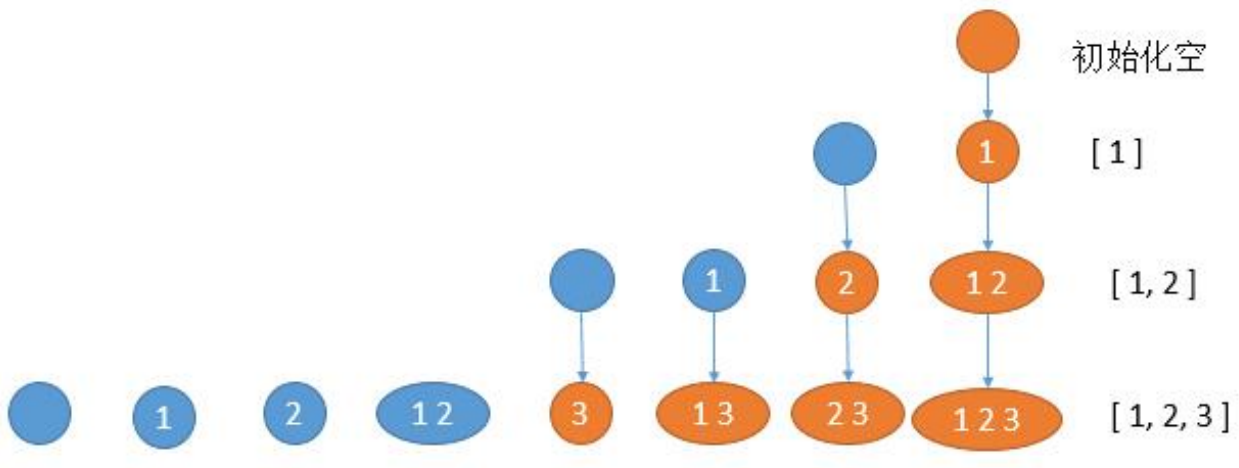

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<List<Integer>> ans = new ArrayList<List<Integer>>();
    ans.add(new ArrayList<Integer>());
```

```
        res.add(new ArrayList<Integer>());
        int n = nums.length;
        // 第一层循环，子数组长度从 1 到 n
        for (int i = 1; i <= n; i++) {
            // 第二层循环，遍历上次的所有结果
            List<List<Integer>> tmp = new ArrayList<List<Integer>>();
            for (List<Integer> list : res) {
                // 第三次循环，对每个结果进行扩展
                for (int m = 0; m < n; m++) {
                    //只添加比末尾数字大的数字，防止重复
                    if (list.size() > 0 && list.get(list.size() - 1) >= nums[m])
                        continue;
                    List<Integer> newList = new ArrayList<Integer>(list);
                    newList.add(nums[m]);
                    tmp.add(newList);
                    ans.add(newList);
                }
            }
            res = tmp;
        }
        return ans;
    }
```

解法二 迭代法2

参照这里。解法一的迭代法，是直接从结果上进行分类，将子数组的长度分为长度是 1 的，2 的 .... n 的。我们还可以从条件上入手，先只考虑给定数组的 1 个元素的所有子数组，然后再考虑数组的 2 个元素的所有子数组 ... 最后再考虑数组的 n 个元素的所有子数组。求 k 个元素的所有子数组，只需要在 k - 1 个元素的所有子数组里边加上 nums [k] 即可。

例如 nums [1 , 2, 3] 的遍历过程。



```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> ans = new ArrayList<>();
    ans.add(new ArrayList<>());//初始化空数组
    for(int i = 0;i<nums.length;i++){
        List<List<Integer>> ans_tmp = new ArrayList<>();
        //遍历之前的所有结果
        for(List<Integer> list : ans){
            List<Integer> tmp = new ArrayList<>(list);
            tmp.add(nums[i]); //加入新增数字
            ans_tmp.add(tmp);
        }
        ans.addAll(ans_tmp);
    }
    return ans;
}
```

解法三 回溯法

参考这里。同样是很经典的回溯法例子，添加一个数，递归，删除之前的数，下次循环。

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> ans = new ArrayList<>();
    getAns(nums, 0, new ArrayList<>(), ans);
    return ans;
}

private void getAns(int[] nums, int start, ArrayList<Integer> temp,
List<List<Integer>> ans) {
    ans.add(new ArrayList<>(temp));
    for (int i = start; i < nums.length; i++) {
        temp.add(nums[i]);
        getAns(nums, i + 1, temp, ans);
        temp.remove(temp.size() - 1);
    }
}
```

解法四 位操作

前方高能！！！！这个方法真的是太太太牛了。参考这里。

数组的每个元素，可以有两个状态，**在**子数组中和**不在**子数组中，所有状态的组合就是所有子数组了。

例如，nums = [1, 2 , 3]。1 代表在，0 代表不在。

```
1 2 3
0 0 0 -> [      ]
0 0 1 -> [    3]
0 1 0 -> [  2   ]
0 1 1 -> [  2 3]
1 0 0 -> [1     ]
1 0 1 -> [1   3]
1 1 0 -> [1 2   ]
1 1 1 -> [1 2 3]
```

所以我们只需要遍历 0 0 0 到 1 1 1，也就是 0 到 7，然后判断每个比特位是否是 1，是 1 的话将对应数字加入即可。如果数组长度是 n，那么每个比特位是 2 个状态，所有总共就是 2 的 n 次方个子数组。遍历 00 ... 0 到 11 ... 1 即可。

```java
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> ans = new ArrayList<>();
    int bit_nums = nums.length;
    int ans_nums = 1 << bit_nums; //执行 2 的 n 次方
    for (int i = 0; i < ans_nums; i++) {
        List<Integer> tmp = new ArrayList<>();
        int count = 0; //记录当前对应数组的哪一位
        int i_copy = i; //用来移位
        while (i_copy != 0) {
            if ((i_copy & 1) == 1) { //判断当前位是否是 1
                tmp.add(nums[count]);
            }
            count++;
            i_copy = i_copy >> 1;//右移一位
        }
        ans.add(tmp);

    }
    return ans;
}
```

总

同样是很经典的一道题，回溯，迭代，最后的位操作真的是太强了，每次遇到关于位操作的解法就很惊叹。

79、题目描述（中等难度）

79. Word Search

Medium 1745 84 Favorite Share

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example:

```
board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]

Given word = "ABCCED", return true.
Given word = "SEE", return true.
Given word = "ABCB", return false.
```

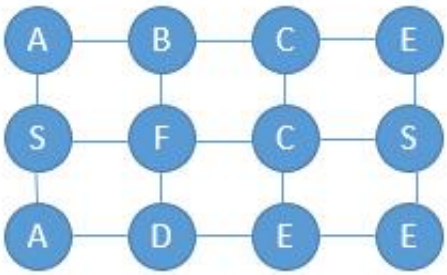
意思就是从某个字符出发，然后它可以向左向右向上向下移动，走过的路径构成一个字符串，判断是否能走出给定字符串的 word ，还有一个条件就是走过的字符不能够走第二次。

比如 SEE，从第二行最后一列的 S 出发，向下移动，再向左移动，就走出了 SEE。

ABCB，从第一行第一列的 A 出发，向右移动，再向右移动，到达 C 以后，不能向左移动回到 B ，并且也没有其他的路径走出 ABCB 所以返回 false。

解法一 DFS

我们可以把矩阵看做一个图，然后利用图的深度优先遍历 DFS 的思想就可以了。



我们需要做的就是，在深度优先遍历过程中，判断当前遍历元素是否对应 word 元素，如果不匹配就结束当前的遍历，返回上一次的元素，尝试其他路径。当然，和普通的 dfs 一样，我们需要一个 visited 数组标记元素是否访问过。

```
public boolean exist(char[][] board, String word) {
```

```
    int rows = board.length;
    if (rows == 0) {
        return false;
    }
    int cols = board[0].length;
    boolean[][] visited = new boolean[rows][cols];
    //从不同位置开始
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            //从当前位置开始符合就返回 true
            if (existRecursive(board, i, j, word, 0, visited)) {
                return true;
            }
        }
    }
    return false;
}

private boolean existRecursive(char[][] board, int row, int col, String word, int
index, boolean[][] visited) {
    //判断是否越界
    if (row < 0 || row >= board.length || col < 0 || col >= board[0].length) {
        return false;
    }
    //当前元素访问过或者和当前 word 不匹配返回 false
    if (visited[row][col] || board[row][col] != word.charAt(index)) {
        return false;
    }
    //已经匹配到了最后一个字母，返回 true
    if (index == word.length() - 1) {
        return true;
    }
    //将当前位置标记位已访问
    visited[row][col] = true;
    //对四个位置分别进行尝试
    boolean up = existRecursive(board, row - 1, col, word, index + 1, visited);
    if (up) {
        return true;
    }
    boolean down = existRecursive(board, row + 1, col, word, index + 1, visited);
    if (down) {
        return true;
    }
    boolean left = existRecursive(board, row, col - 1, word, index + 1, visited);
    if (left) {
        return true;
```

```
    }
    boolean right = existRecursive(board, row, col + 1, word, index + 1,
visited);
    if (right) {
        return true;
    }
    //当前位置没有选进来，恢复标记为 false
    visited[row][col] = false;
    return false;
}
```

我们可以优化一下空间复杂度，我们之前是用了一个等大的二维数组来标记是否访问过。其实我们完全可以用之前的 board，我们把当前访问的元素置为 "$"，也就是一个在 board 中不会出现的字符。然后当上下左右全部尝试完之后，我们再把当前元素还原就可以了。

```
public boolean exist(char[][] board, String word) {
    int rows = board.length;
    if (rows == 0) {
        return false;
    }
    int cols = board[0].length;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (existRecursive(board, i, j, word, 0)) {
                return true;
            }
        }
    }
    return false;
}

private boolean existRecursive(char[][] board, int row, int col, String word, int
index) {
    if (row < 0 || row >= board.length || col < 0 || col >= board[0].length) {
        return false;
    }
    if (board[row][col] != word.charAt(index)) {
        return false;
    }
    if (index == word.length() - 1) {
        return true;
    }
    /*********************改变的地方*****************************************/
    char temp = board[row][col];
    board[row][col] = '$';
```

```
    /*********************************************************************/
    boolean up = existRecursive(board, row - 1, col, word, index + 1);
    if (up) {
        return true;
    }
    boolean down = existRecursive(board, row + 1, col, word, index + 1);
    if (down) {
        return true;
    }
    boolean left = existRecursive(board, row, col - 1, word, index + 1);
    if (left) {
        return true;
    }
    boolean right = existRecursive(board, row, col + 1, word, index + 1);
    if (right) {
        return true;
    }
    /*********************改变的地方*****************************************/
    board[row][col] = temp;
    /*********************************************************************/
    return false;
}
```

在这里，看到另外一种标记和还原的方法。异或。

```
/*********************之前的做法*****************************************/
char temp = board[row][col];
board[row][col] = '$';
/*********************************************************************/

/*********************利用异或*******************************************/
board[row][col] ^= 128;
/*********************************************************************/

//还原
/*********************之前的做法*****************************************/
board[row][col] = temp;
/*********************************************************************/

/*********************利用异或*******************************************/
board[row][col] ^= 128;
/*********************************************************************/
```

至于原理，因为 ASCII 码值的范围是 0 - 127，二进制的话就是 0000 0000 - 0111 1111，我们把它和 128 做异或，也就是和 1000 0000 。这样，如果想还原原来的数字只需要再异或 128 就可以了。

其实原理是一样的，都是把之前的数字变成当前 board 不存在的字符，然后再变回来。只不过这里考虑它的二进制编码，在保留原有信息的基础上做改变，不再需要 temp 变量。

总

关键是对题目的理解，抽象到 DFS，题目就迎刃而解了。异或的应用很强。

80、题目描述（中等难度）

80. Remove Duplicates from Sorted Array II

Medium 649 533 Favorite Share

Given a sorted array *nums*, remove the duplicates **in-place** such that duplicates appeared at most *twice* and return the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array in-place** with O(1) extra memory.

Example 1:

```
Given nums = [1,1,1,2,2,3],

Your function should return length = 5, with the first five elements of nums being 1, 1, 2,
2 and 3 respectively.

It doesn't matter what you leave beyond the returned length.
```

Example 2:

```
Given nums = [0,0,1,1,1,1,2,3,3],

Your function should return length = 7, with the first seven elements of nums being
modified to 0, 0, 1, 1, 2, 3 and 3 respectively.

It doesn't matter what values are set beyond the returned length.
```

26 题的升级版，给定一个数组，每个数字只允许出现 2 次，将满足条件的数字全部移到前边，并且返回有多少数字。例如 [1, 1, 1, 2, 2, 3, 4, 4, 4, 4]，要变为 [1, 1, 2, 2, 3, 4, 4 ...] 剩余部分的数字不要求。

解法一 快慢指针

利用26 题的思想，慢指针指向满足条件的数字的末尾，快指针遍历原数组。并且用一个变量记录当前末尾数字出现了几次，防止超过两次。

```
public int removeDuplicates(int[] nums) {
    int slow = 0;
    int fast = 1;
    int count = 1;
    int max = 2;
    for (; fast < nums.length; fast++) {
        //当前遍历的数字和慢指针末尾数字不同，就加到慢指针的末尾
        if (nums[fast] != nums[slow]) {
            slow++;
            nums[slow] = nums[fast];
            count = 1; //当前数字置为 1 个
        //和末尾数字相同，考虑当前数字的个数，小于 2 的话，就加到慢指针的末尾
        } else {
            if (count < max) {
                slow++;
                nums[slow] = nums[fast];
                count++; //当前数字加 1
            }
        }
    }
    return slow + 1;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

解法二

参考这里，解法一中，我们用一个变量 count 记录了末尾数字出现了几次。而由于给定的数组是有序的，我们可以更直接。将当前快指针遍历的数字和慢指针指向的数字的前一个数字比较（也就是满足条件的倒数第 2 个数），如果相等，因为有序，所有倒数第 1 个数字和倒数第 2 个数字都等于当前数字，再添加就超过 2 个了，所有不添加，如果不相等，那么就添加。s 代表 slow，f 代表 fast。

```
//相等的情况
1 1 1 1 1 2 2 3
  ^   ^
  s   f
//不相等的情况
1 1 1 1 1 2 2 3
  ^       ^
  s       f
```

```java
public int removeDuplicates2(int[] nums) {
    int slow = 1;
    int fast = 2;
    int max = 2;
    for (; fast < nums.length; fast++) {
        //不相等的话就添加
        if (nums[fast] != nums[slow - max + 1]) {
            slow++;
            nums[slow] = nums[fast];
        }
    }
    return slow + 1;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

总

快慢指针是个好东西，解法二直接利用有序，和倒数第 n 个比，从而保证末尾的数字不超过 n 个，太强了。

81、题目描述（中等难度）

81. Search in Rotated Sorted Array II

Medium 658 309 Favorite Share

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., `[0, 0, 1, 2, 2, 5, 6]` might become `[2, 5, 6, 0, 0, 1, 2]`).

You are given a target value to search. If found in the array return `true`, otherwise return `false`.

Example 1:

```
Input: nums = [2,5,6,0,0,1,2], target = 0
Output: true
```

Example 2:

```
Input: nums = [2,5,6,0,0,1,2], target = 3
Output: false
```

33题的升级版，数组的操作没有变，所谓的旋转数组，就是把有序数组前边若干个数字移动到末尾。区别在于这道题出现了重复的数字，同样是找 target。

解法一

把数组遍历一遍，然后依次判断数字是否相等的解法，当然就不用说了。这里直接在33 题解法三的基础上去修改。33 题算法基于一个事实，数组从任意位置劈开后，至少有一半是有序的，什么意思呢?

比如 [4 5 6 7 1 2 3] ，从 7 劈开，左边是 [4 5 6 7] 右边是 [7 1 2 3]，左边是有序的。

基于这个事实。

我们可以先找到哪一段是有序的 (只要判断端点即可)，知道了哪一段有序，我们只需要用正常的二分法就够了，只需要看 target 在不在这一段里，如果在，那么就把另一半丢弃。如果不在，那么就把这一段丢弃。

```
public int search(int[] nums, int target) {
    int start = 0;
    int end = nums.length - 1;
    while (start <= end) {
        int mid = (start + end) / 2;
        if (target == nums[mid]) {
            return mid;
        }
        //左半段是有序的
        if (nums[start] <= nums[mid]) {
            //target 在这段里
            if (target >= nums[start] && target < nums[mid]) {
```

```
                end = mid - 1;
            //target 在另一段里
            } else {
                start = mid + 1;
            }
        //右半段是有序的
        } else {
            if (target > nums[mid] && target <= nums[end]) {
                start = mid + 1;
            } else {
                end = mid - 1;
            }
        }

    }
    return -1;
}
```

如果不加修改，直接放到 leetcode 上跑，发现 nums = [1, 3, 1, 1, 1]，target = 3，返回了 false，当然是不对的了。原因就出现在了，我们在判断哪段有序的时候，当 nums [start] <= nums [mid] 是认为左半段有序。而由于这道题出现了重复数字，此时的 nums [start] = 1, nums [mid] = 1，但此时左半段 [1, 3, 1] 并不是有序的，所以造成我们的算法错误。

所以 nums[start] == nums[mid] 需要我们单独考虑了。操作也很简单，参考这里，当相等的时候，我们只需要让 start++ 就够了。

```
public boolean search(int[] nums, int target) {
    int start = 0;
    int end = nums.length - 1;
    while (start <= end) {
        int mid = (start + end) / 2;
        if (target == nums[mid]) {
            return true;
        }
        //左半段有序
        if (nums[start] < nums[mid]) {
            if (target >= nums[start] && target < nums[mid]) {
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        } else if(nums[start] == nums[mid]){
            start++;
        //右半段有序
        }else{
            if (target > nums[mid] && target <= nums[end]) {
```

```
                start = mid + 1;
            } else {
                end = mid - 1;
            }
        }
    }
    return false;
}
```

时间复杂度：最好的情况，如果没有遇到 nums [start] == nums [mid]，还是 O（log（n））。当然最差的情况，如果是类似于这种 [1, 1, 1, 1, 2, 1]，target = 2，就是 O (n) 了。

空间复杂度：O (1)。

总

基于之前的算法，找出问题所在，然后思考解决方案。开始自己一直纠结于怎么保持时间复杂度还是 log (n)，也没想出解决方案，看了 discuss，发现似乎只能权衡一下。另外 33题 的另外两种解法，好像对于这道题完全失效了，如果大家发现怎么修改，欢迎和我交流。

82、题目描述（中等难度）

82. Remove Duplicates from Sorted List II

Medium 823 72 Favorite Share

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.

Example 1:

```
Input: 1->2->3->3->4->4->5
Output: 1->2->5
```

Example 2:

```
Input: 1->1->1->2->3
Output: 2->3
```

给一个链表，如果一个数属于重复数字，就把这个数删除，一个都不留。

解法一 迭代

只需要两个指针，一个指针 pre 代表重复数字的前边的一个指针，另一个指针 cur 用来遍历链表。d 代表哨兵节点，用来简化边界条件，初始化为 head 指针的前一个节点。p 代表 pre，c 代表 cur。

```
d 1 2 3 3 3 4 cur 和 cur.next 不相等，pre 移到 cur 的地方，cur后移
^ ^
p c

d 1 2 3 3 3 4 cur 和 cur.next 不相等，pre 移到 cur 的地方，cur后移
  ^ ^
  p c

d 1 2 3 3 3 4 5 cur 和 cur.next 相等，pre 保持不变，cur 后移
    ^ ^
    p c

d 1 2 3 3 3 4 5 cur 和 cur.next 相等，pre 保持不变，cur 后移
    ^   ^
    p   c

d 1 2 3 3 3 4 5 cur 和 cur.next 不相等，pre.next 直接指向 cur.next，  删除所有 3，cur
后移
    ^     ^
    p     c

d 1 2 4 5 cur 和 cur.next 不相等，pre 移到 cur 的地方，cur后移
    ^ ^
    p c

d 1 2 4 5 遍历结束
      ^ ^
      p c
```

```java
public ListNode deleteDuplicates(ListNode head) {
    ListNode pre = new ListNode(0);
    ListNode dummy = pre;
    pre.next = head;
    ListNode cur = head;
    while(cur!=null && cur.next!=null){
        boolean equal = false;
        //cur 和 cur.next 相等，cur 不停后移
        while(cur.next!=null && cur.val == cur.next.val){
            cur = cur.next;
            equal = true;
```

```
        }
        //发生了相等的情况
        // pre.next 直接指向 cur.next 删除所有重复数字
        if(equal){
            pre.next = cur.next;
            equal = false;
        //没有发生相等的情况
        //pre 移到 cur 的地方
        }else{
            pre = cur;
        }
        //cur 后移
        cur = cur.next;
    }
    return dummy.next;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

解法二 递归

这里看到了一种递归的解法，分享一下。 主要分了两种情况，头结点和后边的节点相等，头结点和后边的节点不相等。

```
public ListNode deleteDuplicates(ListNode head) {
    if (head == null) return null;
  //如果头结点和后边的节点相等
    if (head.next != null && head.val == head.next.val) {
        //跳过所有重复数字
        while (head.next != null && head.val == head.next.val) {
            head = head.next;
        }
        //将所有重复数字去掉后，进入递归
        return deleteDuplicates(head.next);
    //头结点和后边的节点不相等
    } else {
        //保留头结点，后边的所有节点进入递归
        head.next = deleteDuplicates(head.next);
    }
    //返回头结点
    return head;
}
```

总

主要还是对链表的理解，然后就是指来指去就好了。

83、题目描述（简单难度）

83. Remove Duplicates from Sorted List

Easy 756 78 Favorite Share

Given a sorted linked list, delete all duplicates such that each element appear only *once*.

Example 1:

```
Input: 1->1->2
Output: 1->2
```

Example 2:

```
Input: 1->1->2->3->3
Output: 1->2->3
```

给定一个链表，去重，每个数字只保留一个。

解法一 修改

按偷懒的方法，直接在 82 题的基础上改，如果没做过可以先去看一下。之前是重复的数字一个都不保留，这道题的话要留一个，所以代码也很好改。

迭代法

```
public ListNode deleteDuplicates(ListNode head) {
    ListNode pre = new ListNode(0);
    ListNode dummy = pre;
    pre.next = head;
    ListNode cur = head;
    while(cur!=null && cur.next!=null){
        boolean equal = false;
        while(cur.next!=null && cur.val == cur.next.val){
            cur = cur.next;
            equal = true;
        }

        if(equal){
```

```
            /*************修改的地方*****************/
            //pre.next 指向 cur，不再跳过当前数字
                pre.next = cur;
                pre = cur;
            /************************************/
                equal = false;
            }else{
                pre = cur;
            }
            cur = cur.next;
        }
        return dummy.next;
    }
```

递归

```
public ListNode deleteDuplicates(ListNode head) {
    if (head == null) return null;
    //如果头结点和后边的节点相等
    if (head.next != null && head.val == head.next.val) {
        //跳过所有重复数字
        while (head.next != null && head.val == head.next.val) {
            head = head.next;
        }
        /*************修改的地方*****************/
        //将 head 也包含，进入递归
        return deleteDuplicates(head);
        /************************************/
        //头结点和后边的节点不相等
    } else {
        //保留头结点，后边的所有节点进入递归
        head.next = deleteDuplicates(head.next);
    }
    //返回头结点
    return head;
}
```

解法二 迭代

82 题由于我们要把所有重复的数字都要删除，所有要有一个 pre 指针，指向所有重复数字的最前边。而这道题，我们最终要保留一个数字，所以完全不需要 pre 指针。还有就是，我们不用一次性找到所有重复的数字，我们只需要找到一个，删除一个就够了。所以代码看起来更加简单了。

```
public ListNode deleteDuplicates(ListNode head) {
    ListNode cur = head;
    while(cur!=null && cur.next!=null){
        //相等的话就删除下一个节点
        if(cur.val == cur.next.val){
            cur.next = cur.next.next;
        //不相等就后移
        }else{
            cur = cur.next;
        }
    }
    return head;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

解法三 递归

同样的，递归也会更简单些。

```
public ListNode deleteDuplicates(ListNode head) {
    if(head == null || head.next == null){
        return head;
    }
    //头结点和后一个时候相等
    if(head.val == head.next.val){
        //去掉头结点
        return deleteDuplicates(head.next);
    }else{
        //加上头结点
        head.next = deleteDuplicates(head.next);
        return head;
    }
}
```

总

如果 82 题会做的话，这道题就水到渠成了。

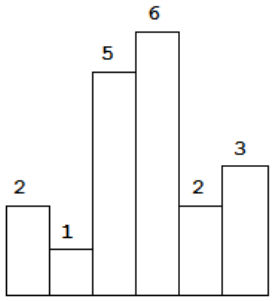
84、题目描述（困难难度）

84. Largest Rectangle in Histogram

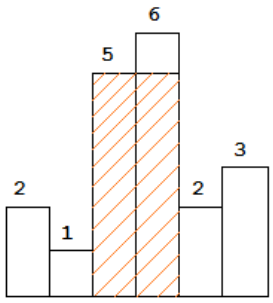
Description | Hints | Submissions | Discuss | Solution

Pick One

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.



The largest rectangle is shown in the shaded area, which has area = `10` unit.

Example:

```
Input: [2,1,5,6,2,3]
Output: 10
```

给一个柱状图，输出一个矩形区域的最大面积。

解法一 暴力破解

以题目给出的例子为例，柱形图高度有 1，2，3，5，6，我们只需要找出每一个高度对应的最大的面积，选出最大的即可。如果求高度为 3 的面积最大的，我们只需要遍历每一个高度，然后看连续的大于等于 3 的柱形有几个，如果是 n 个，那么此时的面积就是 3 * n。所以高度确定的话，我们只需要找连续的大于等于 3 的柱形个数，也就是高度。

```java
public int largestRectangleArea(int[] heights) {
    HashSet<Integer> heightsSet = new HashSet<Integer>();
    //得到所有的高度，也就是去重。
    for (int i = 0; i < heights.length; i++) {
        heightsSet.add(heights[i]);
    }
    int maxArea = 0;
    //遍历每一个高度
    for (int h : heightsSet) {
        int width = 0;
        int maxWidth = 1;
        //找出连续的大于等于当前高度的柱形个数的最大值
        for (int i = 0; i < heights.length; i++) {
            if (heights[i] >= h) {
                width++;
            //出现小于当前高度的就归零，并且更新最大宽度
            } else {
                maxWidth = Math.max(width, maxWidth);
                width = 0;
            }
        }
        maxWidth = Math.max(width, maxWidth);
        //更新最大区域的面积
        maxArea = Math.max(maxArea, h * maxWidth);
    }
    return maxArea;
}
```

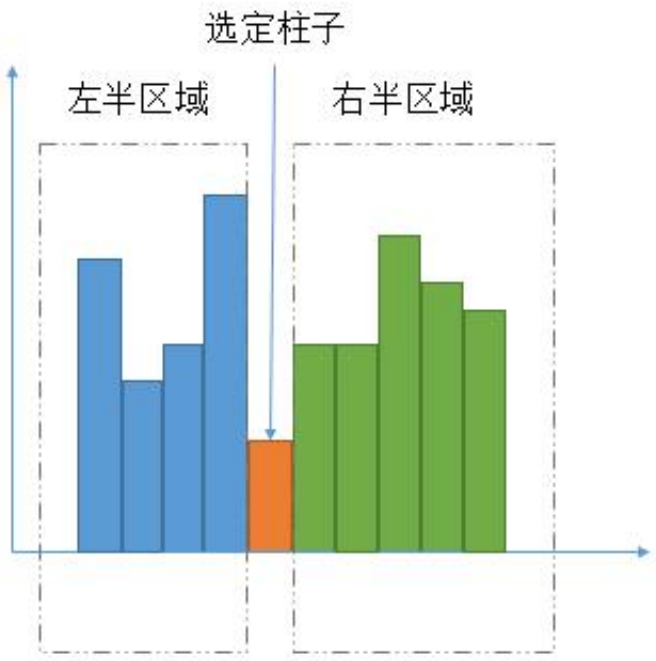
时间复杂度：O（n^2）。

空间复杂度：O(n)。存所有高度。

解法二

参考这里。有一些快排的影子，大家不妨先去回顾一下快排。快排中，我们找了一个基准点，把数组分成了小于基准点的数，和大于基准点的数。然后递归的完成了排序。

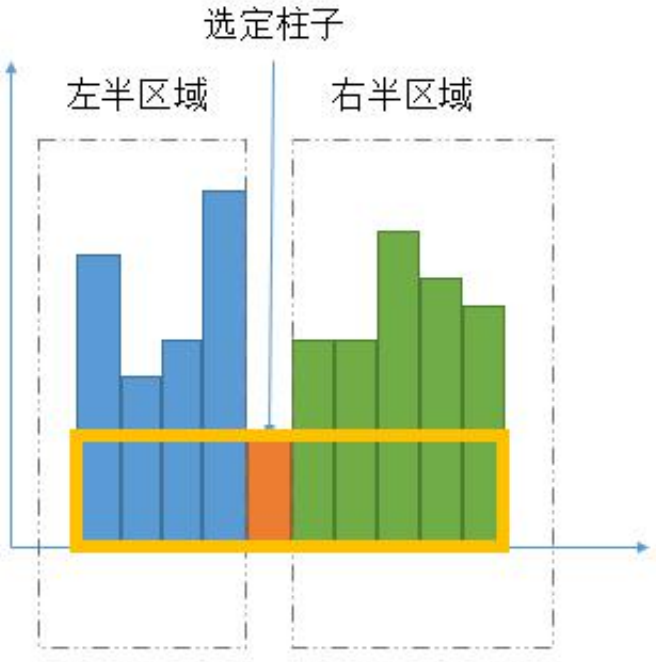
类似的，这里我们也可以找一个柱子。然后把所有柱子分成左半区域和右半区域。



这样要找的最大矩形区域就是三种情况了。

1. 最大矩形区域在不包含选定柱子的左半区域当中。
2. 最大矩形区域在不包含选定柱子的右半区域当中。
3. 最大矩形区域包含选定柱子的区域。

对于 1、2 两种情况，我们只需要递归的去求就行了。而对于第 3 种情况，我们找一个特殊的柱子作为分界点以方便计算，哪一个柱子呢？最矮的那个！有什么好处呢？这样包含该柱子的最大区域，一定是涵盖了当前所有柱子。



所以面积当然是当前选定柱子的高度乘以当前的最大宽度了。

对于当前的时间复杂度，如果每次选定的柱子都可以把区域一分为二，递推式就是

T（n） = 2 * T（n / 2) + n。

上边多加的 n 是找最小柱子耗费，因为需要遍历一遍柱子。然后和快排一样，这样递归下去，时间复杂度就是 O（n log（n））。当然和快排一样的问题，最坏的情况，如果最小柱子每次都出现在末尾，这样会使得只有左半区域，右半区域是 0。递推式就变成了

T（n） = T（n - 1) + n。

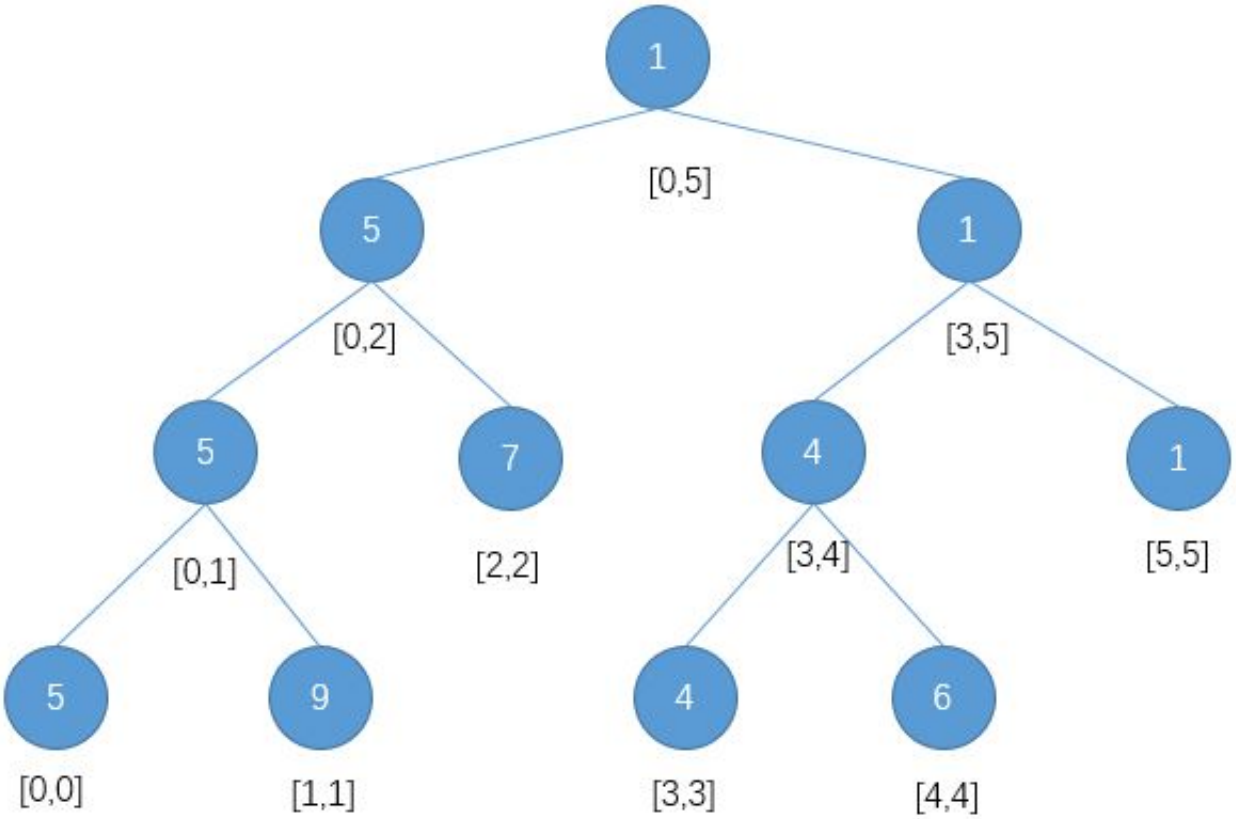
时间复杂度就变成 O（n^2）了，怎么优化呢?

重点就在上边找最小柱子多加的 n 上了，如果我们找最小柱子时间复杂度优化成 log（n）。那么在最坏情况下，递推式变成

T（n） = T（n - 1) + log（n）。

最坏的情况，递推式代入，依旧是 O（n log（n））。而找最小柱子，就可以抽象成，在一个数组区间内找最小值问题，而这个问题前人已经提出了一个数据结构，可以使得时间复杂度是 log（n），完美！名字叫做线段树，可以参考这里和线段树空间复杂度，我就不重复讲了。主要思想就是利用二叉树，使得查找时间复杂度变成了 O（log（n））。

我们以序列 { 5, 9, 7, 4 ,6, 1} 为例，线段树长下边的样子。节点的值代表当前区间内的最小值。



```
class Node// 节点
{
    int start;// 区间的左端点
    int end;// 区间的右端点
    int data;// 该区间的最小值
    int index; // 该节点最小值对应数组的下标

    public Node(int start, int end)// 构造方法中传入左端点和右端点
    {
        this.start = start;
        this.end = end;
    }

}

class SegmentTree {
    private int[] base;;// 给定数组
    Node[] nodes;// 存储线段树的数组

    public SegmentTree(int[] nums) {
        base = new int[nums.length];
        for (int i = 0; i < nums.length; i++) {
            base[i] = nums[i];
        }
        //需要 4n 的空间，上边链接给出了原因
        nodes = new Node[base.length * 4];
    }

    public void build(int index)// 构造一颗线段树，传入下标
    {
        Node node = nodes[index];// 取出该下标下的节点
        if (node == null) {// 根节点需要手动创建
            nodes[index] = new Node(0, this.base.length - 1);
            node = nodes[index];
        }
        if (node.start == node.end) {// 如果这个线段的左端点等于右端点则这个点是叶子节点
            node.data = base[node.start];
            node.index = node.start;
        } else { // 否则递归构造左右子树
            int mid = (node.start + node.end) >> 1;// 现在这个线段的中点
            nodes[(index << 1) + 1] = new Node(node.start, mid);// 左孩子线段
            nodes[(index << 1) + 2] = new Node(mid + 1, node.end);// 右孩子线段
            build((index << 1) + 1);// 构造左孩子
            build((index << 1) + 2);// 构造右孩子
            if (nodes[(index << 1) + 1].data <= nodes[(index << 1) + 2].data) {
                node.data = nodes[(index << 1) + 1].data;
```

```
                node.index = nodes[(index << 1) + 1].index;
            } else {
                node.data = nodes[(index << 1) + 2].data;
                node.index = nodes[(index << 1) + 2].index;
            }
        }
    }

    public Node query(int index, int start, int end) {
        Node node = nodes[index];
        if (node.start > end || node.end < start)
            return null;// 当前区间和待查询区间没有交集，那么返回一个极大值
        if (node.start >= start && node.end <= end)
            return node;// 如果当前的区间被包含在待查询的区间内则返回当前区间的最小值
        Node left = query((index << 1) + 1, start, end);
        int dataLeft = left == null ? Integer.MAX_VALUE : left.data;
        Node right = query((index << 1) + 2, start, end);
        int dataRight = right == null ? Integer.MAX_VALUE : right.data;
        return dataLeft <= dataRight ? left : right;

    }
}
class Solution {
    public int largestRectangleArea(int[] heights) {
        if (heights.length == 0) {
            return 0;
        }
        //构造线段树
        SegmentTree tree = new SegmentTree(heights);
        tree.build(0);
        return getMaxArea(tree, 0, heights.length - 1, heights);
    }
  /**
     *  查询某个区间的最小值
     * @param tree 构造好的线段树
     * @param start 待查询的区间的左端点
     * @param end 待查询的区间的右端点
     * @param heights 给定的数组
     * @return 返回当前区间的矩形区域的最大值
     */
    private int getMaxArea(SegmentTree tree, int start, int end, int[] heights) {
        if (start == end) {
            return heights[start];
        }
        //非法情况，返回一个最小值，防止影响正确的最大值
        if (start > end) {
```

```
            return Integer.MIN_VALUE;
        }
        //找出最小的柱子的下标
        int min = tree.query(0, start, end).index;
        //最大矩形区域包含选定柱子的区域。
        int area1 = heights[min] * (end - start + 1);
        //最大矩形区域在不包含选定柱子的左半区域。
        int area2 = getMaxArea(tree, start, min - 1, heights);
        //最大矩形区域在不包含选定柱子的右半区域。
        int area3 = getMaxArea(tree, min + 1, end, heights);
        //返回最大的情况
        return Math.max(Math.max(area1, area2), area3);
    }
}
```

时间复杂度：O（n log（n））。

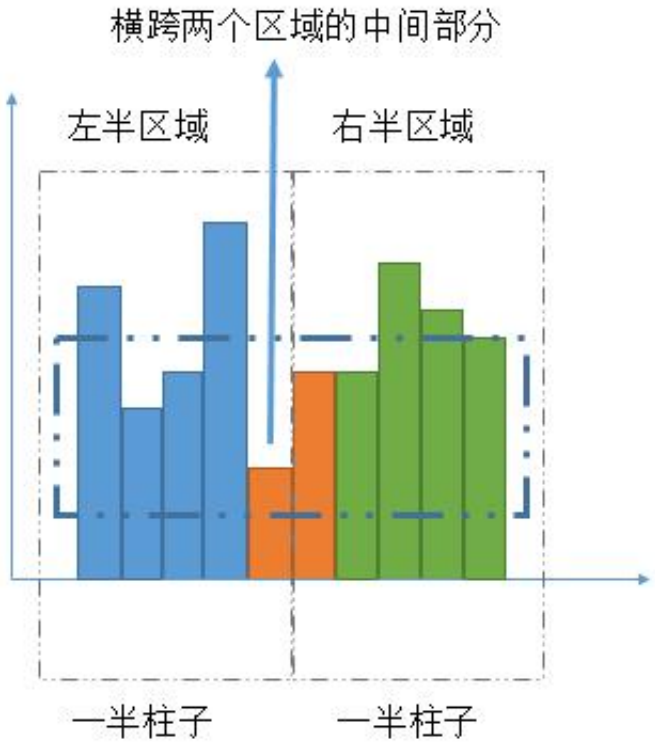
空间复杂度：O（n），用来存储线段树。

解法三

参考这里，思考下解法二中遇到的问题，利用了类似快排的思想，最好情况的递推式是

T（n） = 2 * T（n / 2) + n。

就是 n log（n），但是由于分界点的柱子的选择，并不能总保证两部分的柱子数量均分。所以如果这个问题解决，那么我们就可以保证时间复杂度是 n log（n）了。如何让它均分呢？我们强行把它分成 3 部分呗。



1. 左半区域，含有一半柱子，当然如果总数是奇数个，这里会多一个。
2. 右半区域，含有一半柱子，当然如果总数是奇数个，这里会少一个。
3. 包含最中间柱子的部分，最大区域一定会包含橙色部分，这样才会横跨两个区域。

情况 1、2 的最大区域面积可以用递归来解决，情况 3 的话，我们只要保证是 O（n），就满足了我们的递推式，从而保证时间复杂度是O（n log（n））。怎么做呢?

贪婪的思想，每次选两边较高的柱子扩展柱子。然后其实就是求出了 2 个柱子，3 个柱子，4 个柱子，5 个柱子...每种情况的最大值，然后选最大的就可以了。

1. 初始的时候是两个柱子，记录此时的面积。
2. 然后加 1 个柱子，选取两边较高的柱子，然后计算此时的面积，更新最大区域面积。
3. 不停的重复过程 2 ，直到所有柱子遍历完

```
public int largestRectangleArea(int[] heights) {
    if (heights.length == 0) {
        return 0;
    }
    return getMaxArea(heights, 0, heights.length - 1);
}

private int getMaxArea(int[] heights, int left, int right) {
    if (left == right) {
        return heights[left];
    }
    int mid = left + (right - left) / 2;
```

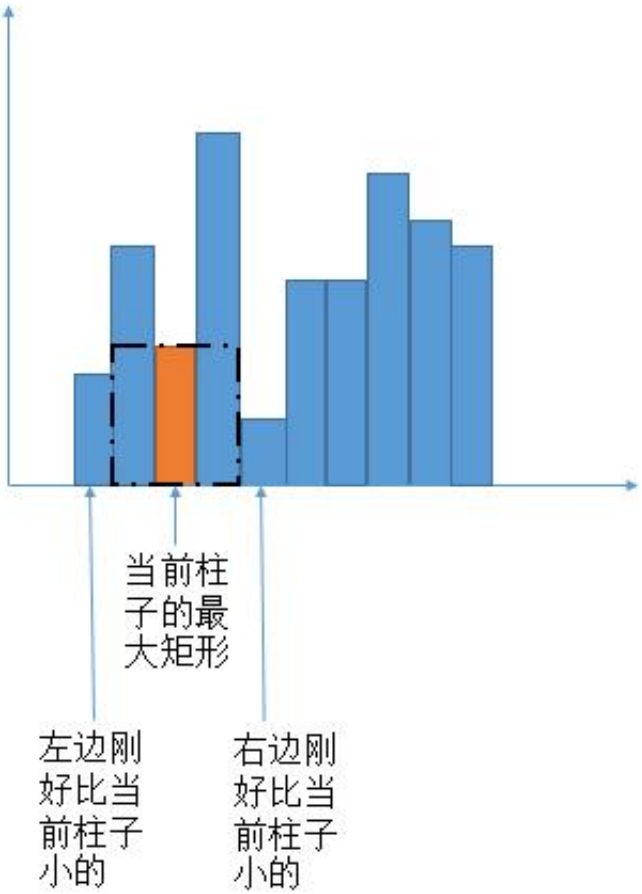
```
    //左半部分
    int area1 = getMaxArea(heights, left, mid);
    //右半部分
    int area2 = getMaxArea(heights, mid + 1, right);
    //中间区域
    int area3 = getMidArea(heights, left, mid, right);
    //选择最大的
    return Math.max(Math.max(area1, area2), area3);
}

private int getMidArea(int[] heights, int left, int mid, int right) {
    int i = mid;
    int j = mid + 1;
    int minH = Math.min(heights[i], heights[j]);
    int area = minH * 2;
    //向两端扩展
    while (i >= left && j <= right) {
        minH = Math.min(minH, Math.min(heights[i], heights[j]));
        //更新最大面积
        area = Math.max(area, minH * (j - i + 1));
        if (i == left) {
            j++;
        } else if (j == right) {
            i--;
            //选择较高的柱子
        } else if (heights[i - 1] >= heights[j + 1]) {
            i--;
        } else {
            j++;

        }
    }
    return area;
}
```

时间复杂度：O（n log（n））。

空间复杂度：O（log（n）），压栈的空间。

解法四

参考这里。解法一暴力破解中，我们把所有矩形区域按高度依次求出来，选出了最大的。这里我们想另外一个分类方法。分别求出包含每个柱子的矩形区域的最大面积，然后选最大的。要包含这个柱子，也就是这个柱子是当前矩形区域的高度。也就是，这个柱子是当前矩形区域中柱子最高的。如下图中包含橙色柱子的矩形区域的最大面积。



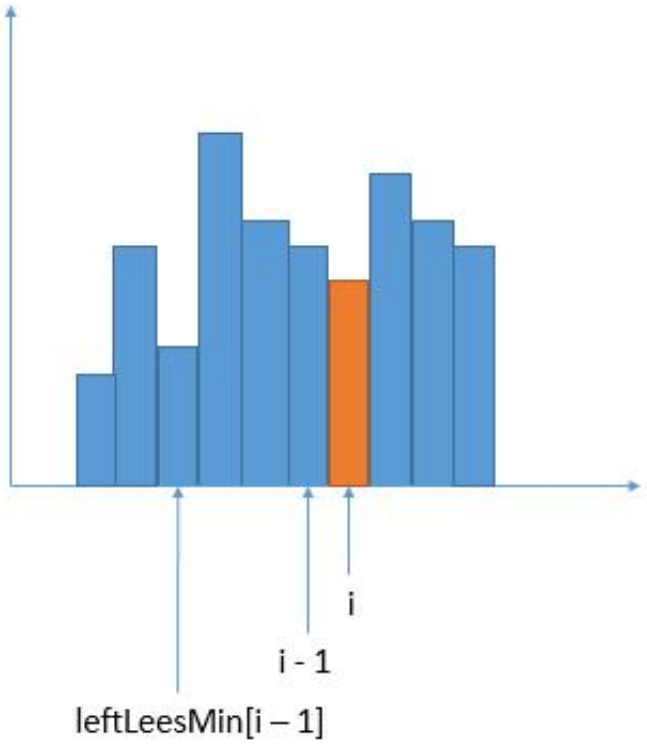
求当前的矩形区域，我们只需要知道比当前柱子到左边第一个小的 leftLessMin 和到右边第一个小的 rightLessMin 两个柱子下标，就可以求出矩形的面积为 (rightLessMin - leftLessMin - 1) * 当前柱子高度。然后遍历每个柱子，按同样的方法求出矩形面积，选最大的就可以了。

现在的问题就是，怎么知道 rightLessMin 和 leftLessMin 。

我们可以用一个数组，leftLessMin[ ] 保存各自的左边第一个小的柱子。

```
leftLessMin[0] = -1;//第一个柱子前边没有柱子，所以赋值为 -1，便于计算面积
for (int i = 1; i < heights.length; i++) {
    int p = i - 1;
    //p 一直减少，找到第一个比当前高度小的柱子就停止
    while (p >= 0 && height[p] >= height[i]) {
        p--;
    }
    leftLessMin[i] = p;
}
```

上边的时间复杂度是 O（n^2），我们可以进行优化。参考下边的图，当前柱子 i 比上一个柱子小的时候，因为我们是找比当前柱子矮的，之前我们进行减 1，判断上上个。但是我们之前已经求出了上一个柱子的 leftLessMin[i - 1]，也就是第一个比上个柱子小的柱子，所以其实我们可以直接跳到 leftLessMin[i - 1] 比较。因为从 leftLessMin[i - 1] + 1到 i - 1 的柱子一定是比当前柱子 i 高的，所以可以跳过。



这样的话时间复杂度达到了O（n）。开始自己不理解，问了一下同学。其实证明的话，可以结合解法五，我们寻找 leftLessMin 其实可以看做压栈出栈的过程，每个元素只会被访问两次。

```
int[] leftLessMin = new int[heights.length];
leftLessMin[0] = -1;
for (int i = 1; i < heights.length; i++) {
    int l = i - 1;
    //当前柱子更小一些，进行左移
    while (l >= 0 && heights[l] >= heights[i]) {
        l = leftLessMin[l];
    }
    leftLessMin[i] = l;
}
```

求到右边第一个小的柱子同理，下边是完整的代码。

```
public int largestRectangleArea(int[] heights) {
    if (heights.length == 0) {
        return 0;
    }
    //求每个柱子的左边第一个小的柱子的下标
    int[] leftLessMin = new int[heights.length];
    leftLessMin[0] = -1;
    for (int i = 1; i < heights.length; i++) {
```

```
        int l = i - 1;
        while (l >= 0 && heights[l] >= heights[i]) {
            l = leftLessMin[l];
        }
        leftLessMin[i] = l;
    }

    //求每个柱子的右边第一个小的柱子的下标
    int[] rightLessMin = new int[heights.length];
    rightLessMin[heights.length - 1] = heights.length;
    for (int i = heights.length - 2; i >= 0; i--) {
        int r = i + 1;
        while (r <= heights.length - 1 && heights[r] >= heights[i]) {
            r = rightLessMin[r];
        }
        rightLessMin[i] = r;
    }

    //求包含每个柱子的矩形区域的最大面积，选出最大的
    int maxArea = 0;
    for (int i = 0; i < heights.length; i++) {
        int area = (rightLessMin[i] - leftLessMin[i] - 1) * heights[i];
        maxArea = Math.max(area, maxArea);
    }
    return maxArea;
}
```

时间复杂度：O（n），取决于找 leftLessMin [i] 的复杂度。

空间复杂度：O（n），保存每个柱子左边右边第一个小的柱子下标。

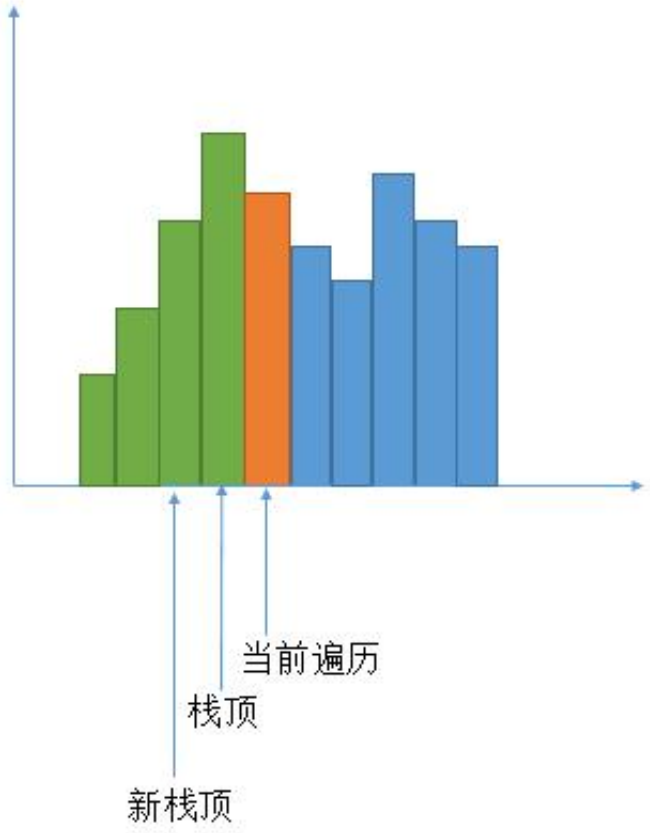
解法五 栈

参考这里。之前也遇到利用栈巧妙解题的，例如42题的解法五，和这道题的共同点就是配对问题。思路的话，本质上和解法四是一样的，可以先看下解法四，左边第一个小于当前柱子和右边第一个小于当前柱子是一对。通过它俩可以求出当前柱子的最大矩形区域。那么具体怎么操作呢?

遍历每个柱子，然后分下边几种情况。

- 如果当前栈空，或者当前柱子大于栈顶柱子的高度，就将当前柱子的下标入栈
- 当前柱子的高度小于栈顶柱子的高度。那么就把栈顶柱子出栈，当做解法四中的当前要求面积的柱子。而右边第一个小于当前柱子的下标就是当前在遍历的柱子，左边第一个小于当前柱子的下标就是当前新的栈顶。

遍历完成后，如果栈没有空。就依次出栈，出栈元素当做解法四中的要求面积的柱子，然后依次计算矩形区域面积。



结合图可以看一下，从头开始遍历，遍历柱子开始的时候都大于前一个柱子高度，所以依次入栈。直到遍历到橙色部分，栈顶元素出栈，计算包含栈顶柱子的矩形区域。而左边第一个小于要求柱子的就是新栈顶，右边第一个小于要求柱子的就是当前遍历柱子。

```
public int largestRectangleArea(int[] heights) {
    int maxArea = 0;
    Stack<Integer> stack = new Stack<>();
    int p = 0;
    while (p < heights.length) {
        //栈空入栈
        if (stack.isEmpty()) {
            stack.push(p);
            p++;
        } else {
            int top = stack.peek();
            //当前高度大于栈顶，入栈
            if (heights[p] >= heights[top]) {
                stack.push(p);
                p++;
            } else {
                //保存栈顶高度
                int height = heights[stack.pop()];
```

```
                //左边第一个小于当前柱子的下标
                int leftLessMin = stack.isEmpty() ? -1 : stack.peek();
                //右边第一个小于当前柱子的下标
                int RightLessMin = p;
                //计算面积
                int area = (RightLessMin - leftLessMin - 1) * height;
                maxArea = Math.max(area, maxArea);
            }
        }
    }
    while (!stack.isEmpty()) {
        //保存栈顶高度
        int height = heights[stack.pop()];
        //左边第一个小于当前柱子的下标
        int leftLessMin = stack.isEmpty() ? -1 : stack.peek();
        //右边没有小于当前高度的柱子，所以赋值为数组的长度便于计算
        int RightLessMin = heights.length;
        int area = (RightLessMin - leftLessMin - 1) * height;
        maxArea = Math.max(area, maxArea);
    }
    return maxArea;
}
```

时间复杂度：O（n），因为对于每个柱子只会经历入栈出栈，所以最多 2n 次。

空间复杂度：O（n），栈的大小。

总

这道题经典呀，第一次用快排的思想去解决问题，太优雅了。另外通过对问题的挖掘，时间复杂度优化到 O（n），也是惊叹。

85、题目描述（困难难度）

85. Maximal Rectangle

Hard 1452 51 Favorite Share

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

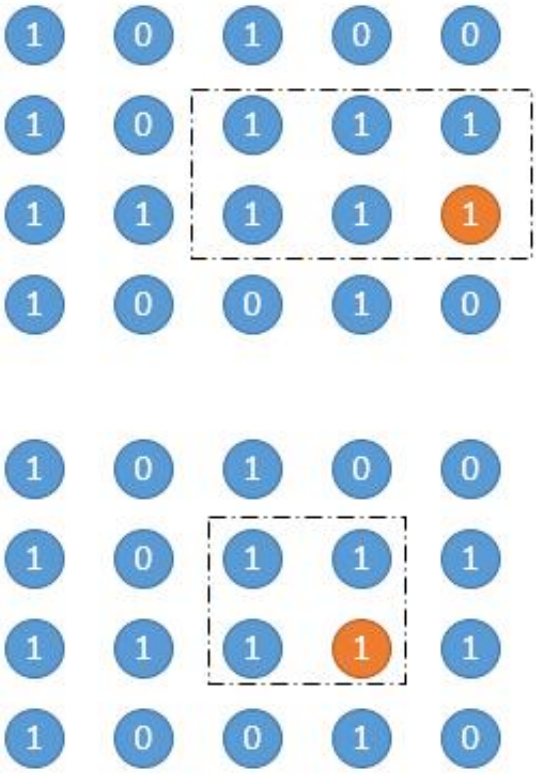
Example:

```
Input:
[
  ["1","0","1","0","0"],
  ["1","0","1","1","1"],
  ["1","1","1","1","1"],
  ["1","0","0","1","0"]
]
Output: 6
```

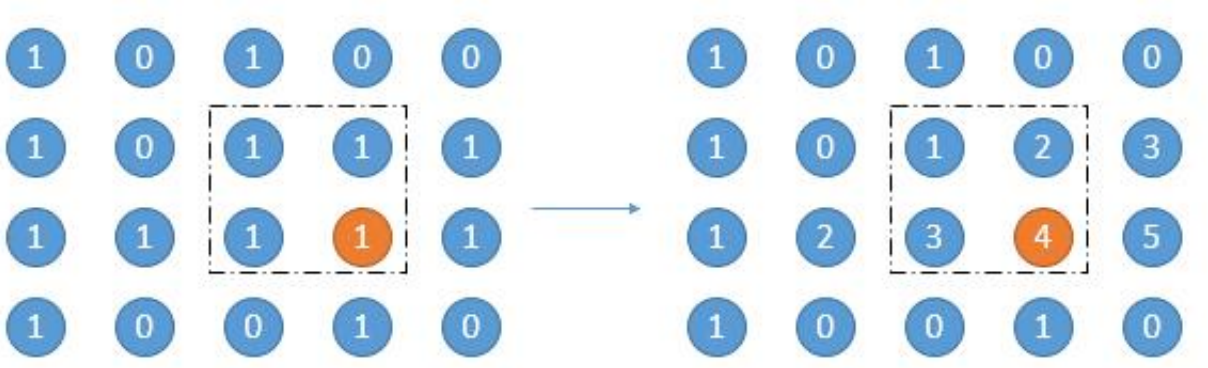
给一个只有 0 和 1 的矩阵，输出一个最大的矩形的面积，这个矩形里边只含有 1。

解法一 暴力破解

参考这里，遍历每个点，求以这个点为矩阵右下角的所有矩阵面积。如下图的两个例子，橙色是当前遍历的点，然后虚线框圈出的矩阵是其中一个矩阵。



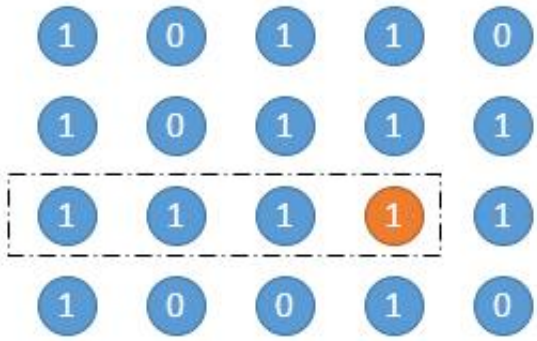
怎么找出这样的矩阵呢？如下图，如果我们知道了以这个点结尾的连续 1 的个数的话，问题就变得简单了。



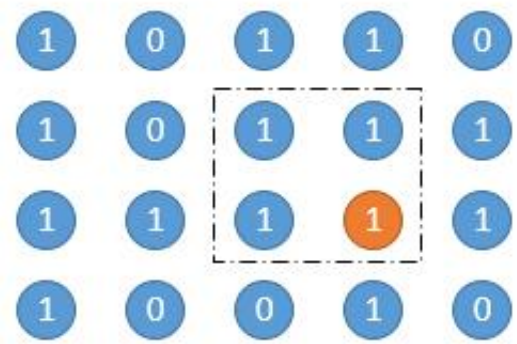
1. 首先求出高度是 1 的矩形面积，也就是它自身的数，也就是上图以橙色的 4 结尾的 「1234」的那个矩形，面积就是 4。
2. 然后向上扩展一行，高度增加一，选出当前列最小的数字，作为矩阵的宽，如上图，当前列中有 `2` 和 `4` ，那么，就将 `2` 作为矩形的宽，求出面积，对应上图的矩形圈出的部分。
3. 然后继续向上扩展，重复步骤 2。

按照上边的方法，遍历所有的点，以当前点为矩阵的右下角，求出所有的矩阵就可以了。下图是某一个点的过程。

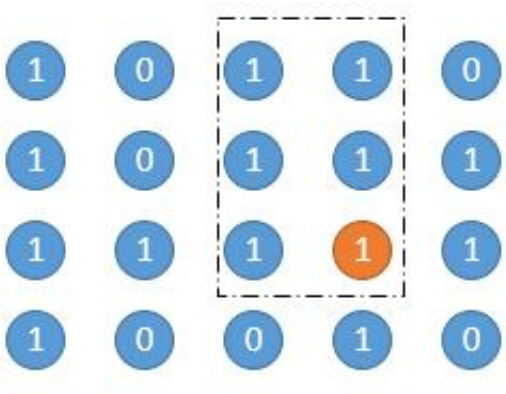
以橙色的点为右下角，高度为 1。



高度为 2。



高度为 3。



代码的话，把求每个点累计的连续 `1` 的个数用 `width` 保存，同时把求最大矩形的面积和求 `width` 融合到同一个循环中。

```
public int maximalRectangle(char[][] matrix) {
    if (matrix.length == 0) {
        return 0;
    }
    //保存以当前数字结尾的连续 1 的个数
    int[][] width = new int[matrix.length][matrix[0].length];
    int maxArea = 0;
    //遍历每一行
    for (int row = 0; row < matrix.length; row++) {
        for (int col = 0; col < matrix[0].length; col++) {
            //更新 width
            if (matrix[row][col] == '1') {
                if (col == 0) {
                    width[row][col] = 1;
                } else {
                    width[row][col] = width[row][col - 1] + 1;
                }
```

```
            } else {
                width[row][col] = 0;
            }
            //记录所有行中最小的数
            int minWidth = width[row][col];
            //向上扩展行
            for (int up_row = row; up_row >= 0; up_row--) {
                int height = row - up_row + 1;
                //找最小的数作为矩阵的宽
                minWidth = Math.min(minWidth, width[up_row][col]);
                //更新面积
                maxArea = Math.max(maxArea, height * minWidth);
            }
        }
    }
    return maxArea;
}
```

时间复杂度：O（m^2n）。

空间复杂度：O（mn）。

解法二

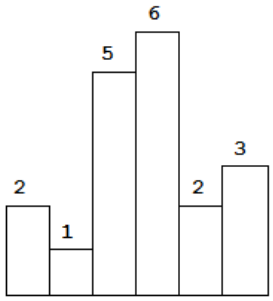
参考这里，接下来的解法，会让这道题变得异常简单。还记得 84 题吗？求一个直方图矩形的最大面积。

84. Largest Rectangle in Histogram

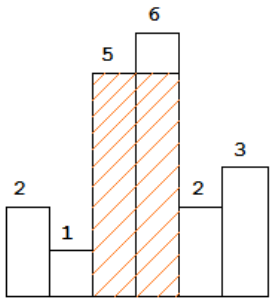
Description | Hints | Submissions | Discuss | Solution

Pick One

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.



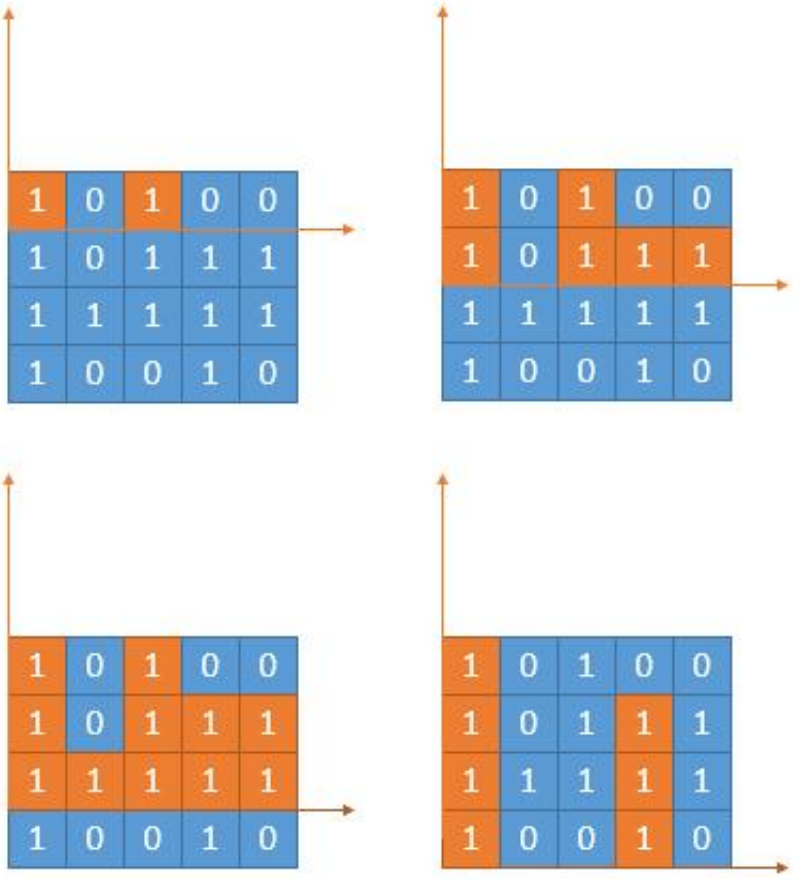
The largest rectangle is shown in the shaded area, which has area = `10` unit.

Example:

```
Input: [2,1,5,6,2,3]
Output: 10
```

大家可以先做 84 题，然后回来考虑这道题。

再想一下这个题，看下边的橙色的部分，这完全就是上一道题呀！



算法有了，就是求出每一层的 heights[] 然后传给上一题的函数就可以了。

利用上一题的栈解法。

```
public int maximalRectangle(char[][] matrix) {
    if (matrix.length == 0) {
        return 0;
    }
    int[] heights = new int[matrix[0].length];
    int maxArea = 0;
    for (int row = 0; row < matrix.length; row++) {
        //遍历每一列，更新高度
        for (int col = 0; col < matrix[0].length; col++) {
            if (matrix[row][col] == '1') {
                heights[col] += 1;
            } else {
                heights[col] = 0;
            }
        }
        //调用上一题的解法，更新函数
        maxArea = Math.max(maxArea, largestRectangleArea(heights));
    }
```

```
    return maxArea;
}

public int largestRectangleArea(int[] heights) {
    int maxArea = 0;
    Stack<Integer> stack = new Stack<>();
    int p = 0;
    while (p < heights.length) {
        //栈空入栈
        if (stack.isEmpty()) {
            stack.push(p);
            p++;
        } else {
            int top = stack.peek();
            //当前高度大于栈顶，入栈
            if (heights[p] >= heights[top]) {
                stack.push(p);
                p++;
            } else {
                //保存栈顶高度
                int height = heights[stack.pop()];
                //左边第一个小于当前柱子的下标
                int leftLessMin = stack.isEmpty() ? -1 : stack.peek();
                //右边第一个小于当前柱子的下标
                int RightLessMin = p;
                //计算面积
                int area = (RightLessMin - leftLessMin - 1) * height;
                maxArea = Math.max(area, maxArea);
            }
        }
    }
    while (!stack.isEmpty()) {
        //保存栈顶高度
        int height = heights[stack.pop()];
        //左边第一个小于当前柱子的下标
        int leftLessMin = stack.isEmpty() ? -1 : stack.peek();
        //右边没有小于当前高度的柱子，所以赋值为数组的长度便于计算
        int RightLessMin = heights.length;
        int area = (RightLessMin - leftLessMin - 1) * height;
        maxArea = Math.max(area, maxArea);
    }
    return maxArea;
}
```

时间复杂度：O（mn）。

空间复杂度：O（n）。

利用上一题的解法四。

```
public int maximalRectangle(char[][] matrix) {
    if (matrix.length == 0) {
        return 0;
    }
    int[] heights = new int[matrix[0].length];
    int maxArea = 0;
    for (int row = 0; row < matrix.length; row++) {
        //遍历每一列，更新高度
        for (int col = 0; col < matrix[0].length; col++) {
            if (matrix[row][col] == '1') {
                heights[col] += 1;
            } else {
                heights[col] = 0;
            }
        }
        //调用上一题的解法，更新函数
        maxArea = Math.max(maxArea, largestRectangleArea(heights));
    }
    return maxArea;
}

public int largestRectangleArea(int[] heights) {
    if (heights.length == 0) {
        return 0;
    }
    int[] leftLessMin = new int[heights.length];
    leftLessMin[0] = -1;
    for (int i = 1; i < heights.length; i++) {
        int l = i - 1;
        while (l >= 0 && heights[l] >= heights[i]) {
            l = leftLessMin[l];
        }
        leftLessMin[i] = l;
    }

    int[] rightLessMin = new int[heights.length];
    rightLessMin[heights.length - 1] = heights.length;
    for (int i = heights.length - 2; i >= 0; i--) {
        int r = i + 1;
        while (r <= heights.length - 1 && heights[r] >= heights[i]) {
            r = rightLessMin[r];
        }
```

```
            rightLessMin[i] = r;
        }
        int maxArea = 0;
        for (int i = 0; i < heights.length; i++) {
            int area = (rightLessMin[i] - leftLessMin[i] - 1) * heights[i];
            maxArea = Math.max(area, maxArea);
        }
        return maxArea;
    }
```

时间复杂度：O（mn）。

空间复杂度：O（n）。

解法三

解法二中套用的栈的解法，我们其实可以不用调用函数，而是把栈糅合到原来求 heights 中。因为栈的话并不是一次性需要所有的高度，所以可以求出一个高度，然后就操作栈。

```
public int maximalRectangle(char[][] matrix) {
    if (matrix.length == 0) {
        return 0;
    }
    int[] heights = new int[matrix[0].length + 1]; //小技巧后边讲
    int maxArea = 0;
    for (int row = 0; row < matrix.length; row++) {
        Stack<Integer> stack = new Stack<Integer>();
        heights[matrix[0].length] = 0;
        //每求一个高度就进行栈的操作
        for (int col = 0; col <= matrix[0].length; col++) {
            if (col < matrix[0].length) { //多申请了 1 个元素，所以要判断
                if (matrix[row][col] == '1') {
                    heights[col] += 1;
                } else {
                    heights[col] = 0;
                }
            }
            if (stack.isEmpty() || heights[col] >= heights[stack.peek()]) {
                stack.push(col);
            } else {
                //每次要判断新的栈顶是否高于当前元素
                while (!stack.isEmpty() && heights[col] < heights[stack.peek()])
{
                    int height = heights[stack.pop()];
                    int leftLessMin = stack.isEmpty() ? -1 : stack.peek();
                    int RightLessMin = col;
```

```
                    int area = (RightLessMin - leftLessMin - 1) * height;
                    maxArea = Math.max(area, maxArea);
                }
                stack.push(col);
            }
        }

    }
    return maxArea;
}
```

时间复杂度：O（mn）。

空间复杂度：O（n）。

里边有一个小技巧，84 题 的栈解法中，我们用了两个 while 循环，第二个 while 循环用来解决遍历完元素栈不空的情况。其实，我们注意到两个 while 循环的逻辑完全一样的。所以我们可以通过一些操作，使得遍历结束后，依旧进第一个 while 循环，从而剩下了第 2 个 while 循环，代码看起来会更简洁。

那就是 heights 多申请一个元素，赋值为 0。这样最后一次遍历的时候，栈顶肯定会大于当前元素，所以就进入了第一个 while 循环。

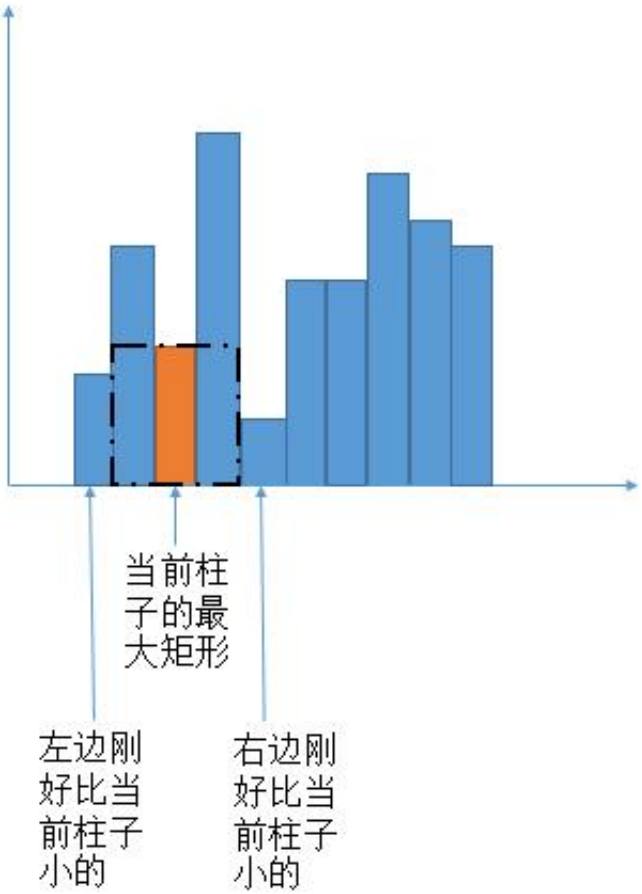
解法四 动态规划

参考这里，这是 leetcode Solution 中投票最高的，但比较难理解，但如果结合 84 题去想的话就很容易了。

解法二中，用了 84 题的两个解法，解法三中我们把栈糅合进了原算法，那么另一种可以一样的思路吗？不行！因为栈不要求所有的高度，可以边更新，边处理。而另一种，是利用两个数组， leftLessMin [ ] 和 rightLessMin [ ]。而这两个数组的更新，是需要所有高度的。

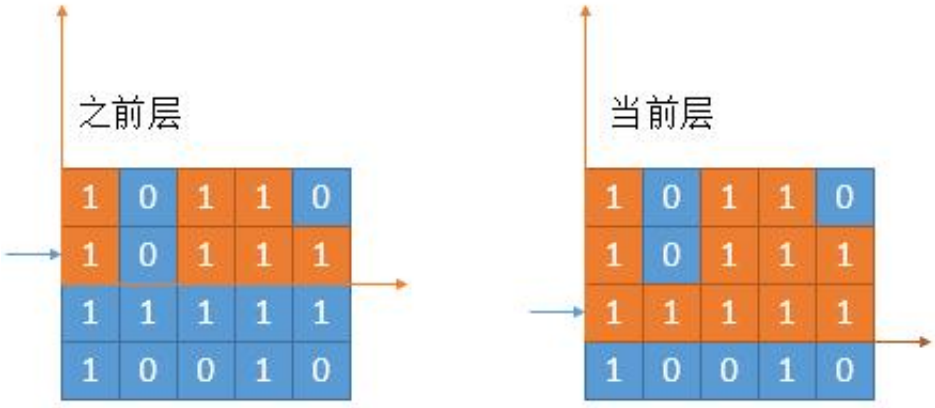
解法二中，我们更新一次 heights，就利用之前的算法，求一遍 leftLessMin [ ] 和 rightLessMin [ ]，然后更新面积。而其实，我们求 leftLessMin [ ] 和 rightLessMin [ ] 可以利用之前的 leftLessMin [ ] 和 rightLessMin [ ] 来更新本次的。

我们回想一下 leftLessMin [ ] 和 rightLessMin [ ] 的含义， leftLessMin [i] 代表左边第一个比当前柱子矮的下标，如下图橙色柱子时当前遍历的柱子。rightLessMin [ ] 时右边第一个。

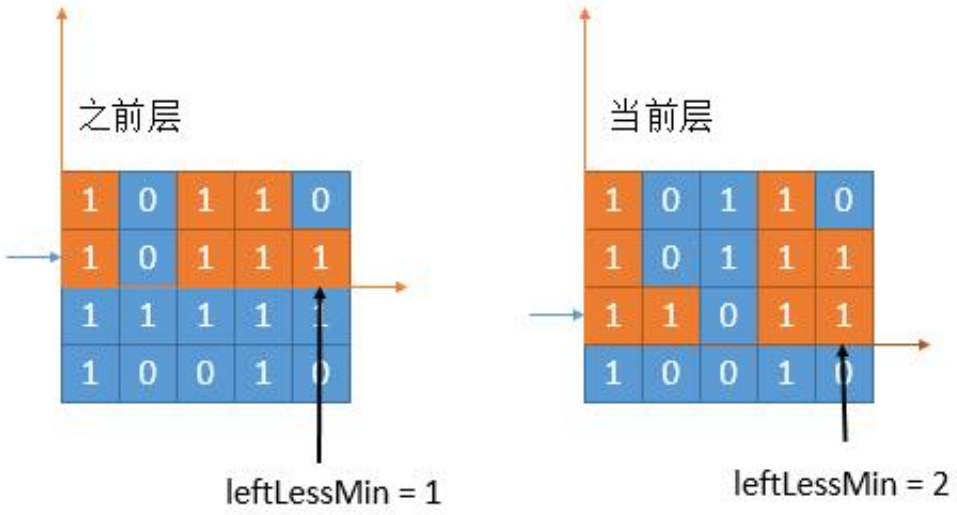


left 和 right 是对称关系，下边只考虑 left 的求法。

如下图，如果当前新增的层全部是 1，当然这时最完美的情况，那么 leftLessMin [ ] 根本不需要改变。



然而事实是残酷的，一定会有 0 的出现。



我们考虑最后一个柱子的更新。上一层的 leftLessMin = 1，也就是蓝色 0 的位置是第一个比它低的柱子。但是在当前层，由于中间出现了 0。所以不再是之前的 leftLessMin ，而是和上次出现 0 的位置进行比较（因为 0 一定比当前柱子小），谁的下标大，更接近当前柱子，就选择谁。上图中出现 0 的位置是 2，之前的 leftLessMin 是 1，选一个较大的，那就是 2 了。

```
public int maximalRectangle4(char[][] matrix) {
    if (matrix.length == 0) {
        return 0;
    }
    int maxArea = 0;
    int cols = matrix[0].length;
    int[] leftLessMin = new int[cols];
    int[] rightLessMin = new int[cols];
    Arrays.fill(leftLessMin, -1); //初始化为 -1，也就是最左边
    Arrays.fill(rightLessMin, cols); //初始化为 cols，也就是最右边
    int[] heights = new int[cols];
    for (int row = 0; row < matrix.length; row++) {
        //更新所有高度
        for (int col = 0; col < cols; col++) {
            if (matrix[row][col] == '1') {
                heights[col] += 1;
            } else {
                heights[col] = 0;
            }
        }
    //更新所有leftLessMin
        int boundary = -1; //记录上次出现 0 的位置
        for (int col = 0; col < cols; col++) {
            if (matrix[row][col] == '1') {
                //和上次出现 0 的位置比较
```

```
                leftLessMin[col] = Math.max(leftLessMin[col], boundary);
            } else {
                //当前是 0 代表当前高度是 0，所以初始化为 -1，防止对下次循环的影响
                leftLessMin[col] = -1;
                //更新 0 的位置
                boundary = col;
            }
        }
        //右边同理
        boundary = cols;
        for (int col = cols - 1; col >= 0; col--) {
            if (matrix[row][col] == '1') {
                rightLessMin[col] = Math.min(rightLessMin[col], boundary);
            } else {
                rightLessMin[col] = cols;
                boundary = col;
            }
        }

        //更新所有面积
        for (int col = cols - 1; col >= 0; col--) {
            int area = (rightLessMin[col] - leftLessMin[col] - 1) * heights[col];
            maxArea = Math.max(area, maxArea);
        }

    }
    return maxArea;

}
```

时间复杂度：O（mn）。

空间复杂度：O（n）。

总

有时候，如果可以把题抽象到已解决的问题当中去，可以大大的简化问题，很酷！

86、题目描述（中等难度）

86. Partition List

Medium 670 184 Favorite Share

Given a linked list and a value *x*, partition it such that all nodes less than *x* come before nodes greater than or equal to *x*.

You should preserve the original relative order of the nodes in each of the two partitions.

Example:

```
Input: head = 1->4->3->2->5->2, x = 3
Output: 1->2->2->4->3->5
```

题目描述的很难理解，其实回想一下快排就很好理解了。就是快排的分区，将链表分成了两部分，一部分的数字全部小于分区点 x，另一部分全部大于等于分区点 x。最后就是 1 2 2 和 4 3 5 两部分。

解法一

回顾下快排的解法，快排中我们分区用了两个指针，一个指针表示该指针前边的数都小于分区点。另一个指针遍历数组。

```
1 4 3 2 5 2  x = 3
  ^   ^
  i   j
i 表示前边的数都小于分区点 3, j 表示当前遍历正在遍历的点
如果 j 当前指向的数小于分区点，就和 i 指向的点交换位置，i 后移

1 2 3 4 5 2  x = 3
    ^   ^
    i   j

然后继续遍历就可以了。
```

这道题无非是换成了链表，而且题目要求不能改变数字的相对位置。所以我们肯定不能用交换的策略了，更何况链表交换也比较麻烦，其实我们直接用插入就可以了。

同样的，用一个指针记录当前小于分区点的链表的末尾，用另一个指针遍历链表，每次遇到小于分区点的数，就把它插入到记录的链表末尾，并且更新末尾指针。dummy 哨兵节点，减少边界情况的判断。

```java
public ListNode partition(ListNode head, int x) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode tail = null;
    head = dummy;
    //找到第一个大于等于分区点的节点，tail 指向它的前边
```

```
    while (head.next != null) {
        if (head.next.val >= x) {
            tail = head;
            head = head.next;
            break;
        }else {
            head = head.next;
        }
    }
    while (head.next != null) {
        //如果当前节点小于分区点，就把它插入到 tail 的后边
        if (head.next.val < x) {
            //拿出要插入的节点
            ListNode move = head.next;
            //将要插入的结点移除
            head.next = move.next;
            //将 move 插入到 tail 后边
            move.next = tail.next;
            tail.next = move;
            //更新 tail
            tail = move;
        }else{
            head = head.next;
        }

    }
    return dummy.next;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

解法二

官方给出的 solution 也许更好理解一些。

我们知道，快排中之所以用相对不好理解的双指针，就是为了减少空间复杂度，让我们想一下最直接的方法。new 两个数组，一个数组保存小于分区点的数，另一个数组保存大于等于分区点的数，然后把两个数组结合在一起就可以了。

```
1 4 3 2 5 2  x = 3
min = {1 2 2}
max = {4 3 5}
接在一起
ans = {1 2 2 4 3 5}
```

数组由于需要多浪费空间，而没有采取这种思路，但是链表就不一样了呀，它并不需要开辟新的空间，而只改变指针就可以了。

```
public ListNode partition(ListNode head, int x) {
    //小于分区点的链表
    ListNode min_head = new ListNode(0);
    ListNode min = min_head;
    //大于等于分区点的链表
    ListNode max_head = new ListNode(0);
    ListNode max = max_head;

    //遍历整个链表
    while (head != null) {
        if (head.val < x) {
            min.next = head;
            min = min.next;
        } else {
            max.next = head;
            max = max.next;
        }

        head = head.next;
    }
    max.next = null;  //这步不要忘记，不然链表就出现环了
    //两个链表接起来
    min.next = max_head.next;

    return min_head.next;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

总

做完了 84、85 连续两个 hard 后，终于可以做个链表压压惊了。本质上就是快排的分区，而在当时被抛弃的用两个数组分别保存，最 naive 的方法，用到链表这里却显示出了它的简洁。

87、题目描述（困难难度）

87. Scramble String

Description | Hints | Submissions | Discuss | Solution

Pick One

Given a string *s1*, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of *s1* = `"great"`:

```
    great
   /    \
  gr    eat
 / \    /  \
g   r  e   at
           / \
          a   t
```

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node `"gr"` and swap its two children, it produces a scrambled string `"rgeat"`.

```
    rgeat
   /    \
  rg    eat
 / \    /  \
r   g  e   at
           / \
          a   t
```

We say that `"rgeat"` is a scrambled string of `"great"`.

Similarly, if we continue to swap the children of nodes `"eat"` and `"at"`, it produces a scrambled string `"rgtae"`.

```
    rgtae
   /    \
  rg    tae
 / \    /  \
r   g  ta  e
       / \
      t   a
```

We say that `"rgtae"` is a scrambled string of `"great"`.

Given two strings *s1* and *s2* of the same length, determine if *s2* is a scrambled string of *s1*.

Example 1:

```
Input: s1 = "great", s2 = "rgeat"
Output: true
```

Example 2:

```
Input: s1 = "abcde", s2 = "caebd"
Output: false
```

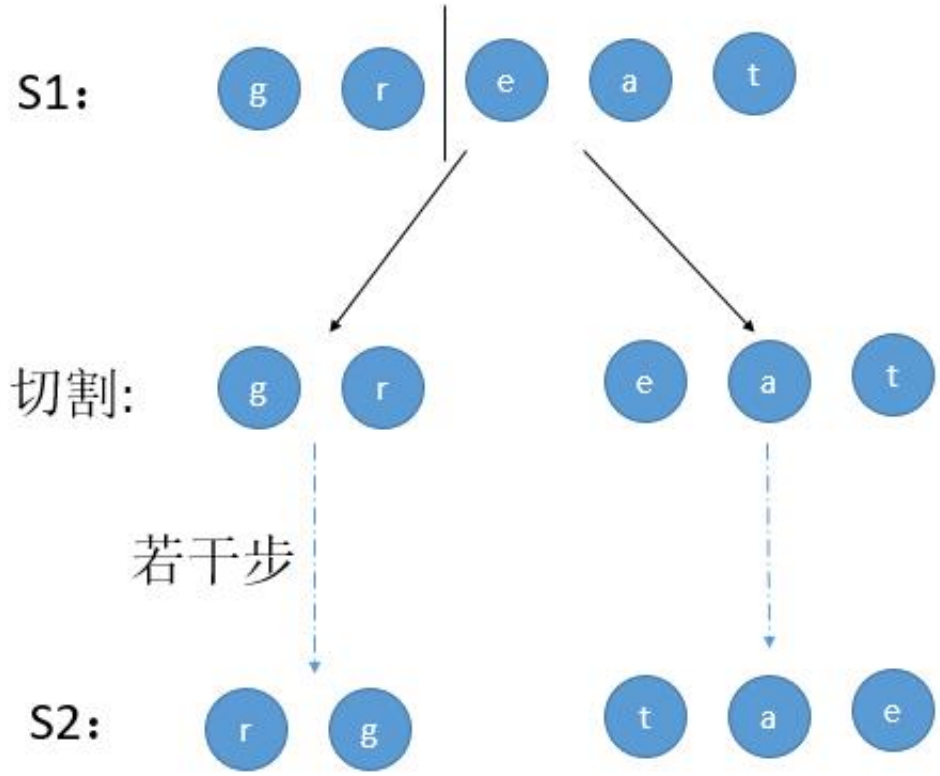
把一个字符串按照树的形状，分成两部分，分成两部分...直到达到叶子节点。并且可以多次交换非叶子节点的两个子树，最后从左到右读取叶子节点，记为生成的字符串。题目是给两个字符串 S1 和 S2，然后问 S2 是否是 S1 经过上述方式生成的。

解法一 递归

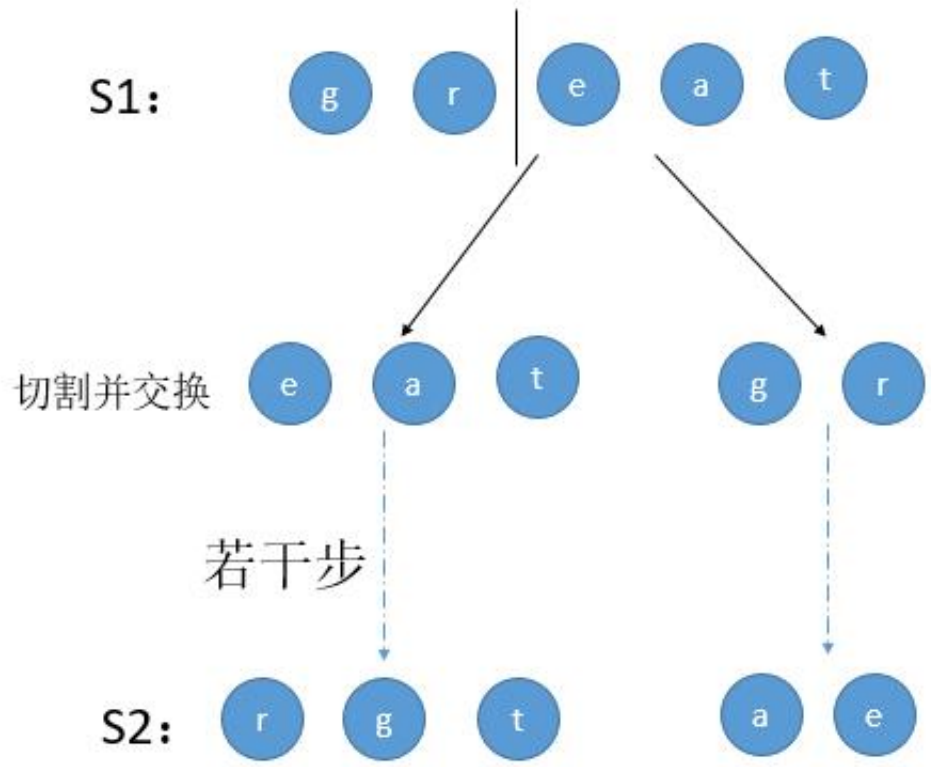
开始的时候，由于给出的图示很巧都是平均分的，我以为只能平均分字符串，看了这里，明白其实可以任意位置把字符串分成两部分，这里需要注意一下。

这道题很容易想到用递归的思想去解，假如两个字符串 great 和 rgeat。考虑其中的一种切割方式。

第 1 种情况：S1 切割为两部分，然后进行若干步切割交换，最后判断两个子树分别是否能变成 S2 的两部分。



第 2 种情况：S1 切割并且交换为两部分，然后进行若干步切割交换，最后判断两个子树是否能变成 S2 的两部分。



上边是一种切割方式，我们只需要遍历所有的切割点即可。

```
public boolean isScramble(String s1, String s2) {
    if (s1.length() != s2.length()) {
        return false;
    }
    if (s1.equals(s2)) {
        return true;
    }

    //判断两个字符串每个字母出现的次数是否一致
    int[] letters = new int[26];
    for (int i = 0; i < s1.length(); i++) {
        letters[s1.charAt(i) - 'a']++;
        letters[s2.charAt(i) - 'a']--;
    }
    //如果两个字符串的字母出现不一致直接返回 false
    for (int i = 0; i < 26; i++) {
        if (letters[i] != 0) {
            return false;
        }
    }

    //遍历每个切割位置
```

```
    for (int i = 1; i < s1.length(); i++) {
        //对应情况 1 ，判断 S1 的子树能否变为 S2 相应部分
        if (isScramble(s1.substring(0, i), s2.substring(0, i)) &&
isScramble(s1.substring(i), s2.substring(i))) {
            return true;
        }
        //对应情况 2 ，S1 两个子树先进行了交换，然后判断 S1 的子树能否变为 S2 相应部分
        if (isScramble(s1.substring(i), s2.substring(0, s2.length() - i)) &&
           isScramble(s1.substring(0, i), s2.substring(s2.length() - i)) ) {
            return true;
        }
    }
    return false;
}
```

时间复杂度：

空间复杂度：

当然，我们可以用 memoization 技术，把递归过程中的结果存储起来，如果第二次递归过来，直接返回结果即可，无需重复递归。

```
public boolean isScramble(String s1, String s2) {
    HashMap<String, Integer> memoization = new HashMap<>();
    return isScrambleRecursion(s1, s2, memoization);
}

public boolean isScrambleRecursion(String s1, String s2, HashMap<String, Integer>
memoization) {
      //判断之前是否已经有了结果
    int ret = memoization.getOrDefault(s1 + "#" + s2, -1);
    if (ret == 1) {
      return true;
    } else if (ret == 0) {
      return false;
    }
    if (s1.length() != s2.length()) {
      memoization.put(s1 + "#" + s2, 0);
      return false;
    }
    if (s1.equals(s2)) {
      memoization.put(s1 + "#" + s2, 1);
      return true;
    }

    int[] letters = new int[26];
```

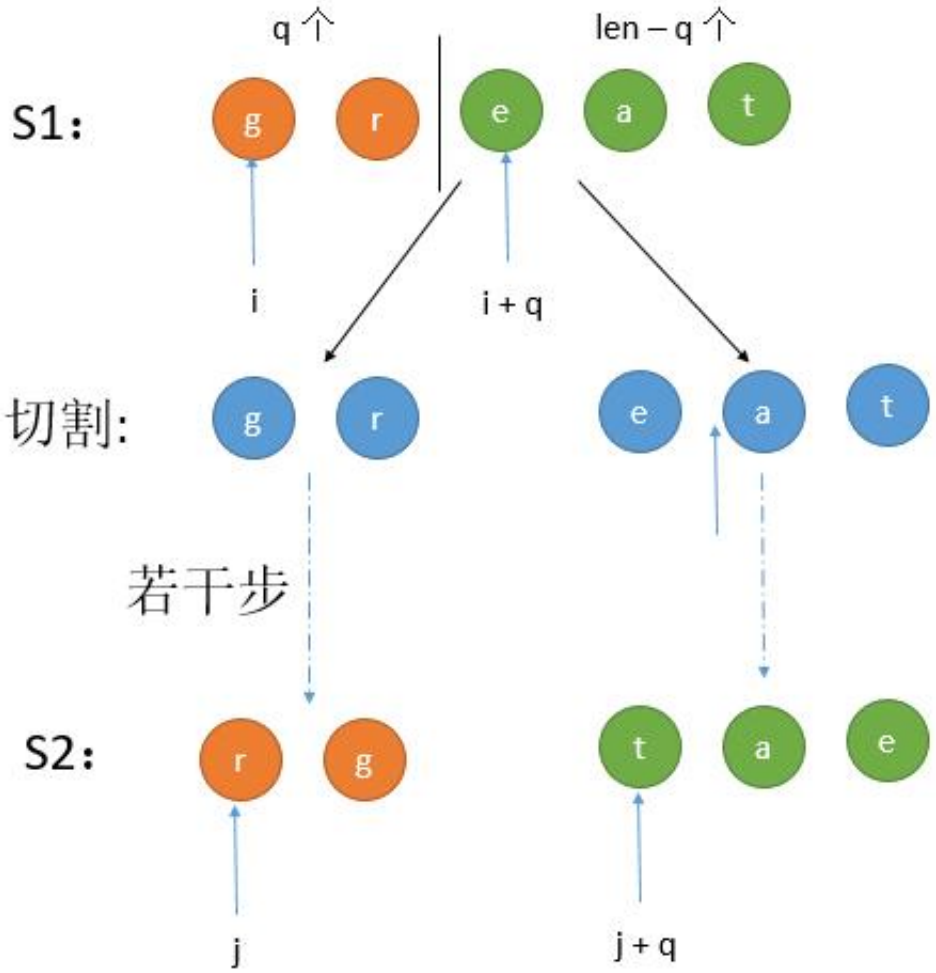
```
    for (int i = 0; i < s1.length(); i++) {
      letters[s1.charAt(i) - 'a']++;
      letters[s2.charAt(i) - 'a']--;
    }
    for (int i = 0; i < 26; i++)
      if (letters[i] != 0) {
        memoization.put(s1 + "#" + s2, 0);
        return false;
      }

    for (int i = 1; i < s1.length(); i++) {
      if (isScramble(s1.substring(0, i), s2.substring(0, i)) &&
isScramble(s1.substring(i), s2.substring(i))) {
        memoization.put(s1 + "#" + s2, 1);
        return true;
      }
      if (isScramble(s1.substring(0, i), s2.substring(s2.length() - i))
          && isScramble(s1.substring(i), s2.substring(0, s2.length() - i))) {
        memoization.put(s1 + "#" + s2, 1);
        return true;
      }
    }
    memoization.put(s1 + "#" + s2, 0);
    return false;
  }
```

解法二 动态规划

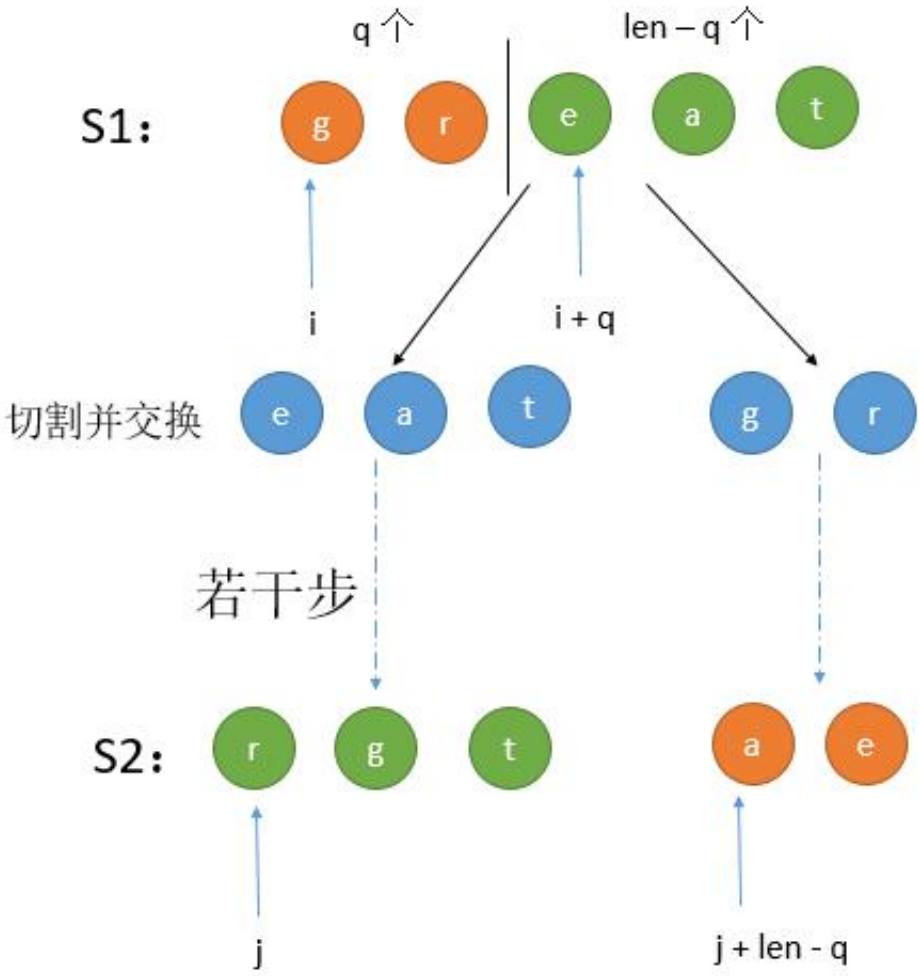
既然是递归，压栈压栈压栈，出栈出栈出栈，我们可以利用动态规划的思想，省略压栈的过程，直接从底部往上走。

我们用 dp [len][i] [j] 来表示 s1[i, i + len) 和 s2 [j, j + len) 两个字符串是否满足条件。换句话说，就是 s1 从 i 开始的 len 个字符是否能转换为 S2 从 j 开始的 len 个字符。那么解法一的两种情况，递归式可以写作。

第 1 种情况，参考下图： 假设左半部分长度是 q，dp [len][i] [j] = dp [q][i] [j] && dp [len - q][i + q] [j + q] 。也就是 S1 的左半部分和 S2 的左半部分以及 S1 的右半部分和 S2 的右半部分。



第 2 种情况，参考下图： 假设左半部分长度是 q，那么 dp [len][i] [j] = dp [q][i] [j + len - q] && dp [len - q][i + q] [j] 。也就是 S1 的右半部分和 S2 的左半部分以及 S1 的左半部分和 S2 的右半部分。




```java
public boolean isScramble4(String s1, String s2) {
    if (s1.length() != s2.length()) {
        return false;
    }
    if (s1.equals(s2)) {
        return true;
    }

    int[] letters = new int[26];
    for (int i = 0; i < s1.length(); i++) {
        letters[s1.charAt(i) - 'a']++;
        letters[s2.charAt(i) - 'a']--;
    }
    for (int i = 0; i < 26; i++) {
        if (letters[i] != 0) {
            return false;
        }
    }

    int length = s1.length();
```

```
    boolean[][][] dp = new boolean[length + 1][length][length];
  //遍历所有的字符串长度
    for (int len = 1; len <= length; len++) {
        //S1 开始的地方
        for (int i = 0; i + len <= length; i++) {
            //S2 开始的地方
            for (int j = 0; j + len <= length; j++) {
                //长度是 1 无需切割
                if (len == 1) {
                    dp[len][i][j] = s1.charAt(i) == s2.charAt(j);
                } else {
                    //遍历切割后的左半部分长度
                    for (int q = 1; q < len; q++) {
                        dp[len][i][j] = dp[q][i][j] && dp[len - q][i + q][j + q]
                            || dp[q][i][j + len - q] && dp[len - q][i + q][j];
                        //如果当前是 true 就 break，防止被覆盖为 false
                        if (dp[len][i][j]) {
                            break;
                        }
                    }
                }
            }
        }
    }
    return dp[length][0][0];
}
```

时间复杂度：$$O(n^4)$$。

空间复杂度：$$O(n^3)$$。

总

有时候太惯性思维了，二分查找做多了，看见树就想二分，这一点需要注意。这里还遇到一个问题时，解法一的 memoization 和解法二的动态规划理论上都会比解法一原始解法快一些，但是在 leetcode 上反而最开始的解法是最快的，这里有些想不通，大家有什么想法可以和我交流下。

88、题目描述（简单难度）

88. Merge Sorted Array

Easy 1102 2800 Favorite Share

Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.

Note:

- The number of elements initialized in *nums1* and *nums2* are *m* and *n* respectively.
- You may assume that *nums1* has enough space (size that is greater or equal to *m* + *n*) to hold additional elements from *nums2*.

Example:

```
Input:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3

Output: [1,2,2,3,5,6]
```

给两个有序数组，把第二个数组合并到第一个数组中，保持有序。可以注意到第一个数组已经为我们多开辟了第二个数组所需要的空间。

解法一 直接法

简单粗暴，nums1 作为被插入的数组，然后遍历 nums2。用两个指针 i 和 j ，i 指向 nums1 当前判断的数字，j 指向 num2 当前遍历的数字。如果 j 指向的数字小于 i 指向的数字，那么就做插入操作。否则的话后移 i ，找到需要插入的位置 。

```
1 2 3 0 0 0  | 2 5 6  //当前 j 指向的数字不小于 i 指向的数字，无需插入，后移 i
^              ^
i              j

1 2 3 0 0 0  | 2 5 6  //当前 j 指向的数字不小于 i 指向的数字，无需插入后移 i
  ^            ^
  i            j

1 2 3 0 0 0  | 2 5 6  //当前 j 指向的数字小于 i 指向的数字，将当前数字插入到 nums1 中
    ^          ^
    i          j

1 2 2 3 0 0  | 2 5 6  //插入完成后，j 进行后移，同时由于在 i 前边插入了数字，i 后移回到原来的数字
    ^          ^
    i          j
```

```
1 2 2 3 0 0  | 2 5 6  //当前 j 指向的数字不小于 i 指向的数字，无需插入后移 i
      ^           ^
      i           j

1 2 2 3 0 0  | 2 5 6  //由于 nums1 完成遍历，将剩余的 nums2 直接插入
        ^         ^
        i         j

1 2 2 3 5 6  | 2 5 6
          ^         ^
          i         j
```

```java
public void merge(int[] nums1, int m, int[] nums2, int n) {
    int i = 0, j = 0;
    //遍历 nums2
    while (j < n) {
        //判断 nums1 是否遍历完
        //（nums1 原有的数和当前已经插入的数相加）和 i 进行比较
        if (i == m + j) {
            //将剩余的 nums2 插入
            while (j < n) {
                nums1[m + j] = nums2[j];
                j++;
            }
            return;
        }
        //判断当前 nums2 是否小于 nums1
        if (nums2[j] < nums1[i]) {
            //nums1 后移数组，空出位置以便插入
            for (int k = m + j; k > i; k--) {
                nums1[k] = nums1[k - 1];
            }
            nums1[i] = nums2[j];
            //i 和 j 后移
            j++;
            i++;
        //当前 nums2 不小于 nums1， i 后移
        }else{
            i++;
        }
    }
}
```

时间复杂度：极端情况下，如果每次都需要插入，那么是 O（n^2）。

空间复杂度：O（1）。

解法二

两个有序数组的合并，其实就是归并排序中的一个步骤。回想下，我们当时怎么做的。

我们当时是新开辟一个和 nums1 + nums2 等大的空间，然后用两个指针遍历 nums1 和 nums2，依次选择小的把它放到新的数组中。

这道题中，nums1 其实就是我们最后合并好的大数组，但是如果 nums1 当作上述新开辟的空间，那原来的 nums1 的数字放到哪里呢？放到 nums1 的末尾。这样我们就可以完全按照归并排序中的思路了，用三个指针就可以了。

```
1 2 3 0 0 0 0 | 2 5 6 7 //将 nums1 的数据放到 nums1 的末尾

1 2 3 0 1 2 3 | 2 5 6 7 //i 和 j 分别指向两组数据的开头，k 指向代插入位置
^       ^       ^
k       i       j

1 2 3 0 1 2 3 | 2 5 6 7 //此时 i 指向的数据小，把它插入，然后 i 后移，k 后移
^       ^       ^
k       i       j

1 2 3 0 1 2 3 | 2 5 6 7 //此时 i 指向的数据小，把它插入，然后 i 后移，k 后移
  ^       ^     ^
  k       i     j

1 2 3 0 1 2 3 | 2 5 6 7 //此时 j 指向的数据小，把它插入，然后 j 后移，k 后移
    ^       ^   ^
    k       i   j

1 2 2 0 1 2 3 | 2 5 6 7 //此时 i 指向的数据小，把它插入，然后 i 后移，k 后移
      ^     ^     ^
      k     i     j

1 2 2 3 1 2 3   | 2 5 6 7 //此时 i 遍历完，把 nums2 全部加入
        ^     ^     ^
        k     i     j

1 2 2 3 5 6 7   | 2 5 6 7
```

```
public void merge(int[] nums1, int m, int[] nums2, int n) {
    //将 nums1 的数字全部移动到末尾
```

```
    for (int count = 1; count <= m; count++) {
        nums1[m + n - count] = nums1[m - count];
    }
    int i = n; //i 从 n 开始
    int j = 0;
    int k = 0;
    //遍历 nums2
    while (j < n) {
        //如果 nums1 遍历结束，将 nums2 直接加入
        if (i == m + n) {
            while (j < n) {
                nums1[k++] = nums2[j++];
            }
            return;
        }
        //哪个数小就对应的添加哪个数
        if (nums2[j] < nums1[i]) {
            nums1[k] = nums2[j++];
        } else {
            nums1[k] = nums1[i++];
        }
        k++;
    }
}
```

时间复杂度： O（n）。

空间复杂度：O（1）。

可以注意到，我们只考虑如果 nums1 遍历结束，将 nums2 直接加入。为什么不考虑如果 nums2 遍历结束，将 nums1 直接加入呢？因为我们最开始的时候已经把 nums1 全部放到了末尾，所以不需要再赋值了。

解法三

本以为自己的解法二已经很机智了，直到看到了这里，发现了一个神仙操作。

解法二中我们的思路是，把 nums1 当作合并后的大数组，依次从两个序列中选较小的数，此外，为了防止 nums1 原有的数字被覆盖，首先先把他放到了末尾。

那么，我们为什么不从 nums1 的末尾开始，依次选两个序列末尾较大的数插入呢？同样是 3 个指针，只不过变成哪个数大就对应的添加哪个数。

```
public void merge3(int[] nums1, int m, int[] nums2, int n) {
    int i = m - 1; //从末尾开始
    int j = n - 1; //从末尾开始
    int k = m + n - 1; //从末尾开始
```

```
    while (j >= 0) {
        if (i < 0) {
            while (j >= 0) {
                nums1[k--] = nums2[j--];
            }
            return;
        }
        //哪个数大就对应的添加哪个数。
        if (nums1[i] > nums2[j]) {
            nums1[k--] = nums1[i--];
        } else {
            nums1[k--] = nums2[j--];
        }
    }
}
```

时间复杂度： O（n）。

空间复杂度：O（1）。

总

这道题看起来简单，但用到的思想很经典了。解法二中充分利用已有空间的思想，以及解法三中逆转我们的惯性思维，给定的数组从小到大，然后惯性上习惯从小到大，但如果逆转过来，从大的选，简直是神仙操作了！

89、题目描述（中等难度）

89. Gray Code

Medium 384 1190 Favorite Share

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer *n* representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

Example 1:

```
Input: 2
Output: [0,1,3,2]
Explanation:
00 - 0
01 - 1
11 - 3
10 - 2

For a given n, a gray code sequence may not be uniquely defined.
For example, [0,2,3,1] is also a valid gray code sequence.

00 - 0
10 - 2
11 - 3
01 - 1
```

Example 2:

```
Input: 0
Output: [0]
Explanation: We define the gray code sequence to begin with 0.
             A gray code sequence of n has size = 2^n, which for n = 0 the size is 2^0 = 1.
             Therefore, for n = 0 the gray code sequence is [0].
```

生成 n 位格雷码，所谓格雷码，就是连续的两个数字，只有一个 bit 位不同。

解法一 动态规划

按照动态规划或者说递归的思路去想，也就是解决了小问题，怎么解决大问题。

我们假设我们有了 n = 2 的解，然后考虑怎么得到 n = 3 的解。

```
n = 2 的解
00 - 0
10 - 2
11 - 3
01 - 1
```

如果再增加一位，无非是在最高位增加 0 或者 1，考虑先增加 0。由于加的是 0，其实数值并没有变化。

```
n = 3 的解，最高位是 0
000 - 0
010 - 2
011 - 3
001 - 1
```

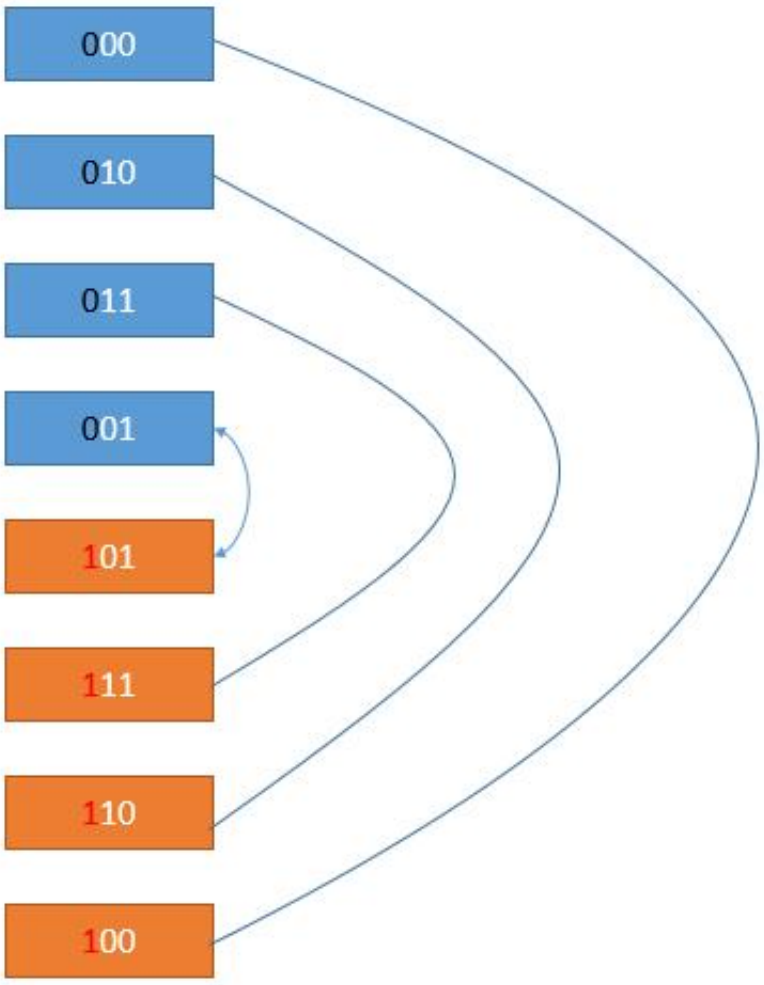
再考虑增加 1，在 n = 2 的解基础上在最高位把 1 丢过去?

```
n = 3 的解
000 - 0
010 - 2
011 - 3
001 - 1
------------- 下面的是新增的
100 - 4
110 - 6
111 - 7
101 - 5
```

似乎没这么简单哈哈，第 4 行 001 和新增的第 5 行 100，有 3 个 bit 位不同了，当然不可以了。怎么解决呢?

很简单，第 5 行新增的数据最高位由之前的第 4 行的 0 变成了 1，所以其它位就不要变化了，直接把第 4 行的其它位拉过来，也就是 101。

接下来，为了使得第 6 行和第 5 行只有一位不同，由于第 5 行拉的第 4 行的低位，而第 4 行和第 3 行只有一位不同。所以第 6 行可以把第 3 行的低位拿过来。其他行同理，如下图。



蓝色部分由于最高位加的是 0，所以它的数值和 n = 2 的所有解的情况一样。而橙色部分由于最高位加了 1，所以值的话，就是在其对应的值上加 4，也就是 $$2^2$$，即$$2^{3-1}$$，也就是 1 << (n - 1) 。所以我们的算法可以用迭代求出来了。

所以如果知道了 n = 2 的解的话，如果是 { 0, 1, 3, 2}，那么 n = 3 的解就是 { 0, 1, 3, 2, 2 + 4, 3 + 4, 1 + 4, 0 + 4 }，即 { 0 1 3 2 6 7 5 4 }。之前的解直接照搬过来，然后倒序把每个数加上 1 << (n - 1) 添加到结果中即可。

```
public List<Integer> grayCode(int n) {
    List<Integer> gray = new ArrayList<Integer>();
    gray.add(0); //初始化 n = 0 的解
    for (int i = 0; i < n; i++) {
        int add = 1 << i; //要加的数
        //倒序遍历，并且加上一个值添加到结果中
        for (int j = gray.size() - 1; j >= 0; j--) {
            gray.add(gray.get(j) + add);
        }
    }
    return gray;
}
```

时间复杂度：$$O(2^n)$$，因为有这么多的结果。

空间复杂度：O（1）。

解法二 直接推导

解法一我觉得，在不了解格雷码的情况下，还是可以想到的，下边的话，应该是之前了解过格雷码才写出来的。看下维基百科提供的一个生成格雷码的思路。

> 以二进制为 0 值的格雷码为第零项，第一项改变最右边的位元，第二项改变右起第一个为1的位元的左边位元，第三、四项方法同第一、二项，如此反复，即可排列出n个位元的格雷码。

以 n = 3 为例。

0 0 0 第零项初始化为 0。

0 0 **1** 第一项改变上一项最右边的位元

0 **1** 1 第二项改变上一项右起第一个为 1 的位元的左边位

0 1 **0** 第三项同第一项，改变上一项最右边的位元

1 1 0 第四项同第二项，改变最上一项右起第一个为 1 的位元的左边位

1 1 **1** 第五项同第一项，改变上一项最右边的位元

1 **0** 1 第六项同第二项，改变最上一项右起第一个为 1 的位元的左边位

1 0 **0** 第七项同第一项，改变上一项最右边的位元

思路有了，代码自然也就出来了。

```
public List<Integer> grayCode2(int n) {
    List<Integer> gray = new ArrayList<Integer>();
    gray.add(0); //初始化第零项
    for (int i = 1; i < 1 << n; i++) {
        //得到上一个的值
        int previous = gray.get(i - 1);
        //同第一项的情况
        if (i % 2 == 1) {
            previous ^= 1; //和 0000001 做异或，使得最右边一位取反
            gray.add(previous);
        //同第二项的情况
        } else {
            int temp = previous;
            //寻找右边起第第一个为 1 的位元
            for (int j = 0; j < n; j++) {
                if ((temp & 1) == 1) {
                    //和 00001000000 类似这样的数做异或，使得相应位取反
                    previous = previous ^ (1 << (j + 1));
```

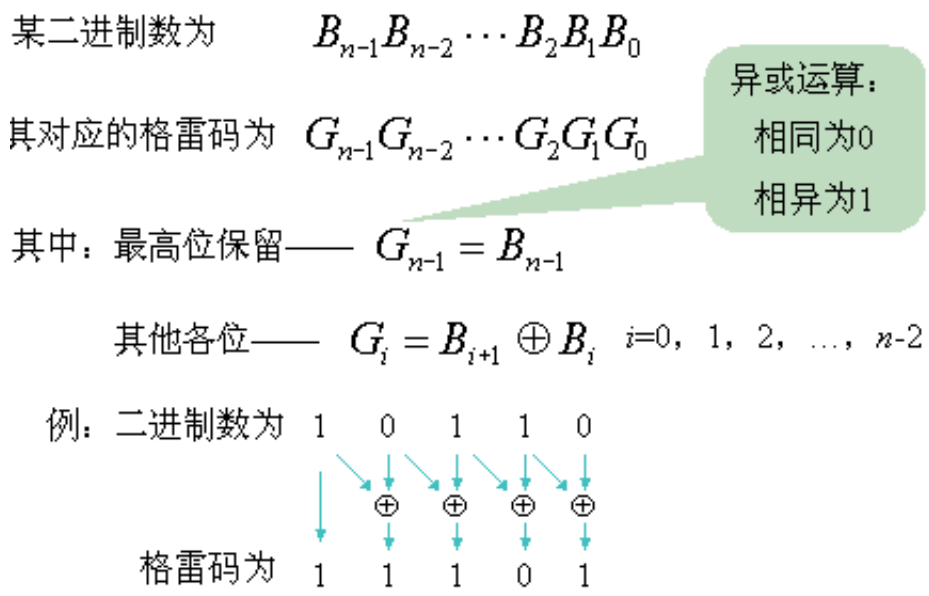
```
                    gray.add(previous);
                    break;
                }
                temp = temp >> 1;
            }
        }
    }
    return gray;
}
```

时间复杂度：由于每添加两个数需要找第一个为 1 的位元，需要 O（n），所以$$O(n2^n)$$。

空间复杂度：O（1）。

解法三 公式

二进制转成格雷码有一个公式。



所以我们遍历 0 到 $$2^n-1$$，然后利用公式转换即可。即最高位保留，其它位是当前位和它的高一位进行异或操作。

```
public List<Integer> grayCode(int n) {
    List<Integer> gray = new ArrayList<Integer>();
    for(int binary = 0;binary < 1 << n; binary++){
        gray.add(binary ^ binary >> 1);
    }
    return gray;
}
```

时间复杂度：$$O(2^n)$$，因为有这么多的结果。

空间复杂度：O（1）。

总

解法一通过利用大问题化小问题的思路，解决了问题。解法二和解法三需要对格雷码有一定的了解才可以。此外，通过格雷码还可以去解汉诺塔和九连环的问题，大家有兴趣可以搜一下。

90、题目描述（中等难度）

90. Subsets II

Medium 897 50 Favorite Share

Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets (the power set).

Note: The solution set must not contain duplicate subsets.

Example:

```
Input: [1,2,2]
Output:
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

78题升级版，大家可以先做 78 题。给定一个数组，输出所有它的子数组。区别在于，这道题给定的数组中，出现了重复的数字。

直接根据 78 题的思路去做。

解法一 回溯法

这个比较好改，我们只需要判断当前数字和上一个数字是否相同，相同的话跳过即可。当然，要把数字首先进行排序。

```
public List<List<Integer>> subsetsWithDup(int[] nums) {
    List<List<Integer>> ans = new ArrayList<>();
    Arrays.sort(nums); //排序
    getAns(nums, 0, new ArrayList<>(), ans);
    return ans;
}
```

```
private void getAns(int[] nums, int start, ArrayList<Integer> temp,
List<List<Integer>> ans) {
    ans.add(new ArrayList<>(temp));
    for (int i = start; i < nums.length; i++) {
        //和上个数字相等就跳过
        if (i > start && nums[i] == nums[i - 1]) {
            continue;
        }
        temp.add(nums[i]);
        getAns(nums, i + 1, temp, ans);
        temp.remove(temp.size() - 1);
    }
}
```

时间复杂度：

空间复杂度：

解法二 迭代法

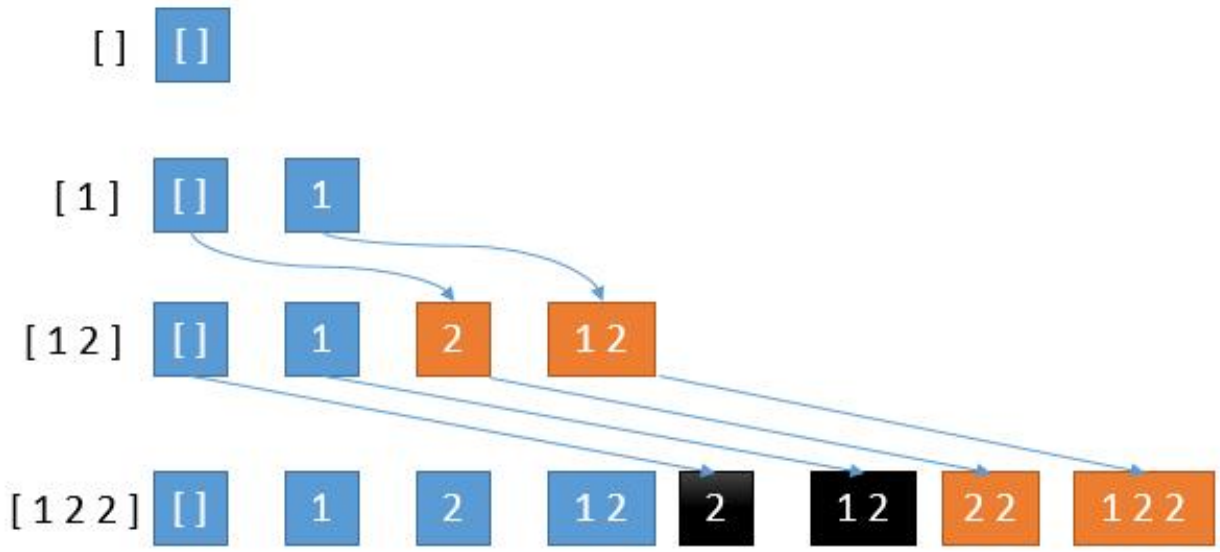
根据78题解法二修改。我们看一下如果直接按照 78 题的思路会出什么问题。之前的思路是，先考虑 0 个数字的所有子串，再考虑 1 个的所有子串，再考虑 2 个的所有子串。而求 n 个的所有子串，就是 【n - 1 的所有子串】和 【n - 1 的所有子串加上 n】。例如，

```
数组 [ 1 2 3 ]
[ ]的所有子串 [ ]
[ 1 ] 个的所有子串 [ ] [ 1 ]
[ 1 2 ] 个的所有子串 [ ] [ 1 ] [ 2 ][ 1 2 ]
[ 1 2 3 ] 个的所有子串 [ ] [ 1 ] [ 2 ] [ 1 2 ] [ 3 ] [ 1 3 ] [ 2 3 ] [ 1 2 3 ]
```

但是如果有重复的数字，会出现什么问题呢

```
数组 [ 1 2 2 ]
[ ] 的所有子串 [ ]
[ 1 ] 的所有子串 [ ] [ 1 ]
[ 1 2 ] 的所有子串 [ ] [ 1 ] [ 2 ][ 1 2 ]
[ 1 2 2 ] 的所有子串 [ ] [ 1 ] [ 2 ] [ 1 2 ] [ 2 ] [ 1 2 ] [ 2 2 ] [ 1 2 2 ]
```

我们发现出现了重复的数组，那么我们可不可以像解法一那样，遇到重复的就跳过这个数字呢？答案是否定的，如果最后一步 [1 2 2] 增加了 2，跳过后，最终答案会缺少 [2 2]、[1 2 2] 这两个解。我们仔细观察这两个解是怎么产生的。



我们看到第 4 行黑色的部分，重复了，是怎么造成的呢?

第 4 行新添加的 2 要加到第 3 行的所有解中，而第 3 行的一部分解是旧解，一部分是新解。可以看到，我们黑色部分是由第 3 行的旧解产生的，橙色部分是由新解产生的。

而第 1 行到第 2 行，已经在旧解中加入了 2 产生了第 2 行的橙色部分，所以这里如果再在旧解中加 2 产生黑色部分就造成了重复。

所以当有重复数字的时候，我们只考虑上一步的新解，算法中用一个指针保存每一步的新解开始的位置即可。

```
public List<List<Integer>> subsetsWithDup(int[] nums) {
    List<List<Integer>> ans = new ArrayList<>();
    ans.add(new ArrayList<>());// 初始化空数组
    Arrays.sort(nums);
    int start = 1; //保存新解的开始位置
    for (int i = 0; i < nums.length; i++) {
        List<List<Integer>> ans_tmp = new ArrayList<>();
        // 遍历之前的所有结果
        for (int j = 0; j < ans.size(); j++) {
            List<Integer> list = ans.get(j);
            //如果出现重复数字，就跳过所有旧解
            if (i > 0 && nums[i] == nums[i - 1] && j < start) {
                continue;
            }
            List<Integer> tmp = new ArrayList<>(list);
            tmp.add(nums[i]); // 加入新增数字
            ans_tmp.add(tmp);
        }
```

```
            start = ans.size(); //更新新解的开始位置
            ans.addAll(ans_tmp);
        }
        return ans;
    }
```

时间复杂度：

空间复杂度：O（1）。

还有一种思路，参考这里，当有重复数字出现的时候我们不再按照之前的思路走，而是单独考虑这种情况。

当有 n 个重复数字出现，其实就是在出现重复数字之前的所有解中，分别加 1 个重复数字， 2 个重复数字，3 个重复数字 ... 什么意思呢，看一个例子。

```
数组 [ 1 2 2 2 ]
[ ]的所有子串 [ ]
[ 1 ] 个的所有子串 [ ] [ 1 ]
然后出现了重复数字 2，那么我们记录重复的次数。然后遍历之前每个解即可
对于 [ ] 这个解，
加 1 个 2，变成 [ 2 ]
加 2 个 2，变成 [ 2 2 ]
加 3 个 2，变成 [ 2 2 2 ]
对于 [ 1 ] 这个解
加 1 个 2，变成 [ 1 2 ]
加 2 个 2，变成 [ 1 2 2 ]
加 3 个 2，变成 [ 1 2 2 2 ]
```

代码的话，就很好写了。

```
public List<List<Integer>> subsetsWithDup(int[] num) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    List<Integer> empty = new ArrayList<Integer>();
    result.add(empty);
    Arrays.sort(num);

    for (int i = 0; i < num.length; i++) {
        int dupCount = 0;
        //判断当前是否是重复数字，并且记录重复的次数
        while( ((i+1) < num.length) && num[i+1] == num[i]) {
            dupCount++;
            i++;
        }
        int prevNum = result.size();
        //遍历之前几个结果的每个解
```

```
            for (int j = 0; j < prevNum; j++) {
                List<Integer> element = new ArrayList<Integer>(result.get(j));
                //每次在上次的结果中多加 1 个重复数字
                for (int t = 0; t <= dupCount; t++) {
                    element.add(num[i]); //加入当前重复的数字
                    result.add(new ArrayList<Integer>(element));
                }
            }
        }
        return result;
    }
```

解法三 位操作

本以为这个思路想不出来怎么去改了，然后看到了这里。

回顾一下，这个题的思想就是每一个数字，考虑它的二进制表示。

例如，nums = [1, 2 , 3]。用 1 代表在，0 代表不在。

```
1 2 3
0 0 0 -> [      ]
0 0 1 -> [    3]
0 1 0 -> [  2  ]
0 1 1 -> [  2 3]
1 0 0 -> [1    ]
1 0 1 -> [1   3]
1 1 0 -> [1 2  ]
1 1 1 -> [1 2 3]
```

但是如果有了重复数字，很明显就行不通了。例如对于 nums = [1 2 2 2 3]。

```
1 2 2 2 3
0 1 1 0 0  -> [  2 2  ]
0 1 0 1 0  -> [  2 2  ]
0 0 1 1 0  -> [  2 2  ]
```

上边三个数产生的数组重复的了。三个中我们只取其中 1 个，取哪个呢？取从重复数字的开头连续的数字。什么意思呢？就是下边的情况是我们所保留的。

```
2 2 2 2 2
1 0 0 0 0 -> [  2          ]
1 1 0 0 0 -> [  2 2        ]
1 1 1 0 0 -> [  2 2 2      ]
1 1 1 1 0 -> [  2 2 2 2    ]
1 1 1 1 1 -> [  2 2 2 2 2  ]
```

而对于 [2 2] 来说，除了 1 1 0 0 0 可以产生，下边的几种情况，都是产生的 [2 2]

```
2 2 2 2 2
1 1 0 0 0 -> [  2 2        ]
1 0 1 0 0 -> [  2 2        ]
0 1 1 0 0 -> [  2 2        ]
0 1 0 1 0 -> [  2 2        ]
0 0 0 1 1 -> [  2 2        ]
......
```

怎么把 1 1 0 0 0 和上边的那么多种情况区分开来呢？我们来看一下出现了重复数字，并且当前是 1 的前一个的二进位。

对于 **1** 1 0 0 0，是 1。

对于 1 **0** 1 0 0，是 0。

对于 **0** 1 1 0 0，是 0。

对于 **0** 1 0 1 0，是 0。

对于 0 0 **0** 1 1，是 0。

.....

可以看到只有第一种情况对应的是 1 ，其他情况都是 0。其实除去从开头是连续的 1 的话，就是两种情况。

第一种就是，占据了开头，类似于这种 1**0**...1....

第二种就是，没有占据开头，类似于这种 **0**...1...

这两种情况，除了第一位，其他位的 1 的前边一定是 0。所以的话，我们的条件是看出现了重复数字，并且当前位是 1 的前一个的二进位。

所以可以改代码了。

```java
public List<List<Integer>> subsetsWithDup(int[] num) {
    Arrays.sort(num);
    List<List<Integer>> lists = new ArrayList<>();
    int subsetNum = 1<<num.length;
    for(int i=0;i<subsetNum;i++){
        List<Integer> list = new ArrayList<>();
```

```
            boolean illegal=false;
            for(int j=0;j<num.length;j++){
                //当前位是 1
                if((i>>j&1)==1){
                    //当前是重复数字，并且前一位是 0，跳过这种情况
                    if(j>0&&num[j]==num[j-1]&&(i>>(j-1)&1)==0){
                        illegal=true;
                        break;
                    }else{
                        list.add(num[j]);
                    }
                }
            }
            if(!illegal){
                lists.add(list);
            }

        }
        return lists;
    }
```

总

解法一和解法二怎么改，分析一下比较容易想到。解法三就比较难了，突破口就是选一个特殊的结构做代表，和其他情况区分出来。而从头开始的连续 1 可能就会是我们第一个想到的数，然后分析一下，发现果然可以和其他所有情况区分开来。

91、题目描述（中等难度）

91. Decode Ways

Medium 1411 1635 Favorite Share

A message containing letters from `A-Z` is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given a **non-empty** string containing only digits, determine the total number of ways to decode it.

Example 1:

```
Input: "12"
Output: 2
Explanation: It could be decoded as "AB" (1 2) or "L" (12).
```

Example 2:

```
Input: "226"
Output: 3
Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).
```

每个数字对应一个字母，给一串数字，问有几种解码方式。例如 226 可以有三种，2|2|6，22|6，2|26。

解法一 递归

很容易想到递归去解决，将大问题化作小问题。

比如 232232323232。

对于第一个字母我们有两种划分方式。

2|**32232323232** 和 23|**2232323232**

所以，如果我们分别知道了上边划分的右半部分 32232323232 的解码方式是 ans1 种，2232323232 的解码方式是 ans2 种，那么整体 232232323232 的解码方式就是 ans1 + ans2 种。可能一下子，有些反应不过来，可以看一下下边的类比。

假如从深圳到北京可以经过**武汉**和**上海**两条路，而从**武汉**到北京有 8 条路，从**上海**到北京有 6 条路。那么从深圳到北京就有 8 + 6 = 14 条路。

```
public int numDecodings(String s) {
    return getAns(s, 0);
}
```

```
private int getAns(String s, int start) {
    //划分到了最后返回 1
    if (start == s.length()) {
        return 1;
    }
    //开头是 0,0 不对应任何字母，直接返回 0
    if (s.charAt(start) == '0') {
        return 0;
    }
    //得到第一种的划分的解码方式
    int ans1 = getAns(s, start + 1);
    int ans2 = 0;
    //判断前两个数字是不是小于等于 26 的
    if (start < s.length() - 1) {
        int ten = (s.charAt(start) - '0') * 10;
        int one = s.charAt(start + 1) - '0';
        if (ten + one <= 26) {
            ans2 = getAns(s, start + 2);
        }
    }
    return ans1 + ans2;
}
```

时间复杂度：

空间复杂度：

解法二 递归 memoization

解法一的递归中，走完左子树，再走右子树会把一些已经算过的结果重新算，所以我们可以用 memoization 技术，就是算出一个结果很就保存，第二次算这个的时候直接拿出来就可以了。

```
public int numDecodings(String s) {
    HashMap<Integer, Integer> memoization = new HashMap<>();
    return getAns(s, 0, memoization);
}

private int getAns(String s, int start, HashMap<Integer, Integer> memoization) {
    if (start == s.length()) {
        return 1;
    }
    if (s.charAt(start) == '0') {
        return 0;
    }
    //判断之前是否计算过
    int m = memoization.getOrDefault(start, -1);
```

```
    if (m != -1) {
        return m;
    }
    int ans1 = getAns(s, start + 1, memoization);
    int ans2 = 0;
    if (start < s.length() - 1) {
        int ten = (s.charAt(start) - '0') * 10;
        int one = s.charAt(start + 1) - '0';
        if (ten + one <= 26) {
            ans2 = getAns(s, start + 2, memoization);
        }
    }
    //将结果保存
    memoization.put(start, ans1 + ans2);
    return ans1 + ans2;
}
```

解法三 动态规划

同样的，递归就是压栈压栈压栈，出栈出栈出栈的过程，我们可以利用动态规划的思想，省略压栈的过程，直接从 bottom 到 top。

用一个 dp 数组， dp [i] 代表字符串 s [i, s.len-1]，也就是 s 从 i 开始到结尾的字符串的解码方式。

这样和递归完全一样的递推式。

如果 s [i] 和 s [i + 1] 组成的数字小于等于 26，那么

dp [i] = dp[i + 1] + dp [i + 2]

```
public int numDecodings(String s) {
    int len = s.length();
    int[] dp = new int[len + 1];
    dp[len] = 1; //将递归法的结束条件初始化为 1
    //最后一个数字不等于 0 就初始化为 1
    if (s.charAt(len - 1) != '0') {
        dp[len - 1] = 1;
    }
    for (int i = len - 2; i >= 0; i--) {
        //当前数字时 0 ，直接跳过，0 不代表任何字母
        if (s.charAt(i) == '0') {
            continue;
        }
        int ans1 = dp[i + 1];
        //判断两个字母组成的数字是否小于等于 26
        int ans2 = 0;
```

```
            int ten = (s.charAt(i) - '0') * 10;
            int one = s.charAt(i + 1) - '0';
            if (ten + one <= 26) {
                ans2 = dp[i + 2];
            }
            dp[i] = ans1 + ans2;

        }
        return dp[0];
    }
```

接下来就是，动态规划的空间优化了，例如5题，10题，53题，72题等等都是同样的思路。都是注意到一个特点，当更新到 dp [i] 的时候，我们只用到 dp [i + 1] 和 dp [i + 2]，之后的数据就没有用了。所以我们不需要 dp 开 len + 1 的空间。

简单的做法，我们只申请 3 个空间，然后把 dp 的下标对 3 求余就够了。

```
public int numDecodings4(String s) {
    int len = s.length();
    int[] dp = new int[3];
    dp[len % 3] = 1;
    if (s.charAt(len - 1) != '0') {
        dp[(len - 1) % 3] = 1;
    }
    for (int i = len - 2; i >= 0; i--) {
        if (s.charAt(i) == '0') {
            dp[i % 3] = 0; //这里很重要，因为空间复用了，不要忘记归零
            continue;
        }
        int ans1 = dp[(i + 1) % 3];
        int ans2 = 0;
        int ten = (s.charAt(i) - '0') * 10;
        int one = s.charAt(i + 1) - '0';
        if (ten + one <= 26) {
            ans2 = dp[(i + 2) % 3];
        }
        dp[i % 3] = ans1 + ans2;

    }
    return dp[0];
}
```

然后，如果多考虑以下，我们其实并不需要 3 个空间，我们只需要 2 个就够了，只需要更新的时候，指针移动一下，代码如下。

```java
public int numDecodings5(String s) {
    int len = s.length();
    int end = 1;
    int cur = 0;
    if (s.charAt(len - 1) != '0') {
        cur = 1;
    }
    for (int i = len - 2; i >= 0; i--) {
        if (s.charAt(i) == '0') {
            end = cur;//end 前移
            cur = 0;
            continue;
        }
        int ans1 = cur;
        int ans2 = 0;
        int ten = (s.charAt(i) - '0') * 10;
        int one = s.charAt(i + 1) - '0';
        if (ten + one <= 26) {
            ans2 = end;
        }
        end = cur; //end 前移
        cur = ans1 + ans2;

    }
    return cur;
}
```

总

从递归，到动态规划，到动态规划的空间复杂度优化，已经很多这样的题了，很经典。

92、题目描述（中等难度）

92. Reverse Linked List II

Medium 1243 92 Favorite Share

Reverse a linked list from position *m* to *n*. Do it in one-pass.

Note: 1 ≤ *m* ≤ *n* ≤ length of list.

Example:

```
Input: 1->2->3->4->5->NULL, m = 2, n = 4
Output: 1->4->3->2->5->NULL
```

给定链表的一个范围，将这个范围内的链表倒置。

解法一

首先找到 m 的位置，记录两端的节点 left1 和 left2 。

然后每遍历一个节点，就倒置一个节点。

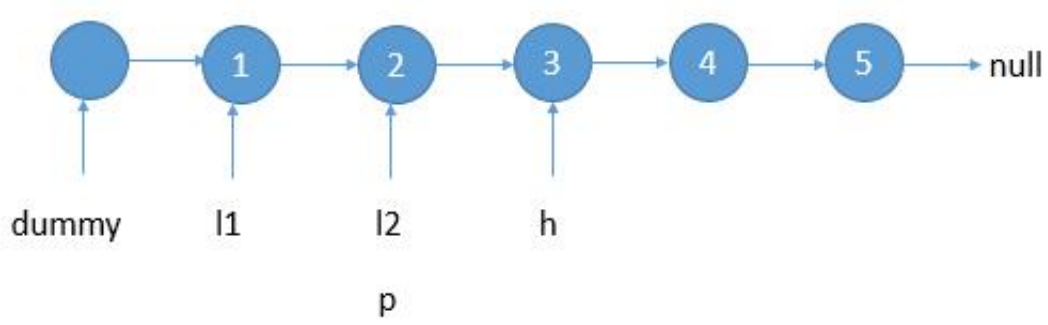
到 n 的位置后，利用之前的 left1 和 left2 完成连接。

为了完成链表的倒置需要两个指针 pre 和 head。为了少考虑边界条件，例如 m = 1 的倒置。加一个哨兵节点 dummy。

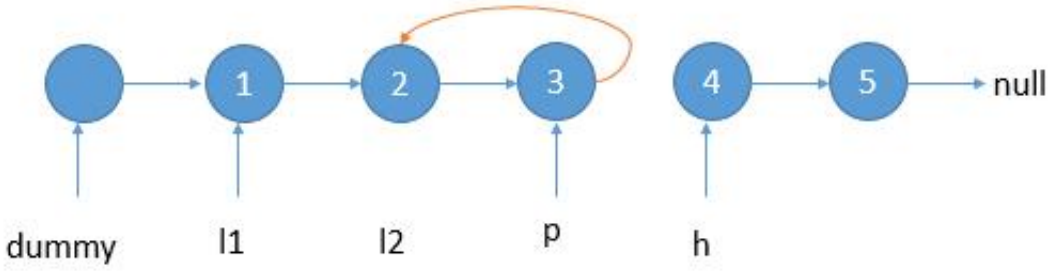
```
m = 2, n = 4

1 2 3 4 5 加入哨兵节点 d，pre 简写 p，head 简写 h

0 1 2 3 4 5 往后遍历
^ ^
d h
p

0 1 2 3 4 5 此时 h 指向 m 的位置，记录 p 和 h 为 l1 和 l2
^ ^ ^
d p h

0  1  2 3 4 5 然后继续遍历
^  ^  ^
d  p  h
   l1 l2

0  1  2  3 4 5 开始倒置链表，使得 h 指向 p
^  ^  ^  ^
d  l1 p  h
```

```
    l2
```

当前状态用图形描述

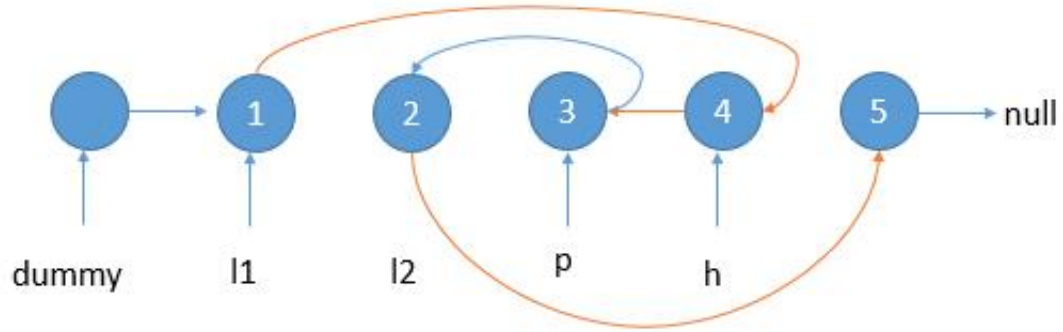


倒转链表，将 h 的 next 指向 p，并且后移 p 和 h。



然后上边一步会重复多次，直到 h 到达 n 的位置。当然这道题比较特殊，上图 h 已经到达了 n 的位置。

此时，我们需要将 h 指向 p，同时将 l1 指向 h，l2 指向 h.next，使得链表接起来。



操作完成，将 dummy.next 返回即可。

```
public ListNode reverseBetween(ListNode head, int m, int n) {
    if (m == n) {
        return head;
    }
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    int count = 0;
    ListNode left1 = null;
```

```
    ListNode left2 = null;
    ListNode pre = dummy;
    while (head != null) {
        count++;
        //到达 m，保存 l1 和 l2
        if (count == m) {
            left1 = pre;
            left2 = head;
        }
        // m 和 n 之间，倒转链表
        if (count > m && count < n) {
            ListNode temp = head.next;
            head.next = pre;
            pre = head;
            head = temp;
            continue;
        }
        //到达 n
        if (count == n) {
            left2.next = head.next;
            head.next = pre;
            left1.next = head;
            break;
        }
        //两个指针后移
        head = head.next;
        pre = pre.next;
    }
    return dummy.next;
}
```

时间复杂度：O（n）。

空间复杂度：O（1）。

总

考察链表知识，如果对链表很熟悉，在纸上画一画，理清楚怎么指向，很快可以写出来。

93、题目描述（中等难度）

93. Restore IP Addresses

Medium 681 278 Favorite Share

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

Example:

```
Input: "25525511135"
Output: ["255.255.11.135", "255.255.111.35"]
```

给一个字符串，输出所有的可能的 ip 地址，注意一下，01.1.001.1 类似这种 0 开头的是非法字符串。

解法一 回溯 递归 DFS

很类似于刚做过的 91 题，对字符串进行划分。这个其实也是划分，划分的次数已经确定了，那就是分为 4 部分。那么就直接用回溯的思想，第一部分可能是 1 位数，然后进入递归。第一部分可能是 2 位数，然后进入递归。第一部分可能是 3 位数，然后进入递归。很好理解，直接看代码理解吧。

```
public List<String> restoreIpAddresses(String s) {
    List<String> ans = new ArrayList<>(); //保存最终的所有结果
    getAns(s, 0, new StringBuilder(), ans, 0);
    return ans;
}

/**
* @param:  start 字符串开始部分
* @param:  temp 已经划分的部分
* @param:  ans 保存所有的解
* @param:  count 当前已经加入了几部分
*/
private void getAns(String s, int start, StringBuilder temp, List<String> ans,
int count) {
    //如果剩余的长度大于剩下的部分都取 3 位数的长度，那么直接结束
    //例如 s = 121231312312, length = 12
    //当前 start = 1, count 等于 1
    //剩余字符串长度 11，剩余部分 4 - count = 3 部分，最多 3 * 3 是 9
    //所以一定是非法的，直接结束
    if (s.length() - start > 3 * (4 - count)) {
        return;
    }
    //当前刚好到达了末尾
    if (start == s.length()) {
        //当前刚好是 4 部分，将结果加入
        if (count == 4) {
```

```
            ans.add(new String(temp.substring(0, temp.length() - 1)));
        }
        return;
    }
    //当前超过末位，或者已经到达了 4 部分结束掉
    if (start > s.length() || count == 4) {
        return;
    }
    //保存的当前的解
    StringBuilder before = new StringBuilder(temp);

    //加入 1 位数
    temp.append(s.charAt(start) + "" + '.');
    getAns(s, start + 1, temp, ans, count + 1);

    //如果开头是 0，直接结束
    if (s.charAt(start) == '0')
        return;

    //加入 2 位数
    if (start + 1 < s.length()) {
        temp = new StringBuilder(before);//恢复为之前的解
        temp.append(s.substring(start, start + 2) + "" + '.');
        getAns(s, start + 2, temp, ans, count + 1);
    }

    //加入 3 位数
    if (start + 2 < s.length()) {
        temp = new StringBuilder(before);
        int num = Integer.parseInt(s.substring(start, start + 3));
        if (num >= 0 && num <= 255) {
            temp.append(s.substring(start, start + 3) + "" + '.');
            getAns(s, start + 3, temp, ans, count + 1);
        }
    }

}
```

解法二 迭代

参考这里，相当暴力直接。因为我们知道了，需要划分为 4 部分，所以我们直接用利用三个指针将字符串强行分为四部分，遍历所有的划分，然后选取合法的解。

```
public List<String> restoreIpAddresses(String s) {
    List<String> res = new ArrayList<String>();
```

```
    int len = s.length();
    //i < 4 保证第一部分不超过 3 位数
    //i < len - 2 保证剩余的字符串还能分成 3 部分
    for(int i = 1; i<4 && i<len-2; i++){
        for(int j = i+1; j<i+4 && j<len-1; j++){
            for(int k = j+1; k<j+4 && k<len; k++){
                //保存四部分的字符串
                String s1 = s.substring(0,i), s2 = s.substring(i,j), s3 =
s.substring(j,k), s4 = s.substring(k,len);
                //判断是否合法
                if(isValid(s1) && isValid(s2) && isValid(s3) && isValid(s4)){
                    res.add(s1+"."+s2+"."+s3+"."+s4);
                }
            }
        }
    }
    return res;
}
public boolean isValid(String s){
    if(s.length()>3 || s.length()==0 || (s.charAt(0)=='0' && s.length()>1) ||
Integer.parseInt(s)>255)
        return false;
    return true;
}
```

时间复杂度：如果不考虑我们调用的内部函数，Integer.parseInt，s.substring，那么就是 O（1）。因为每一层循环最多遍历 4 次。考虑的话每次调用的时间复杂度是 O（n），常数次调用，所以是 O（n）。

空间复杂度：O（1）。

总

回溯或者说深度优先遍历，经常遇到了。但是解法二的暴力方法竟然通过了，有些意外。另外分享下 discuss 里有趣的评论，哈哈哈哈。

WHO CAN BEAT THIS CODE ?

mitbbs8080 ★ 267 Last Edit: October 27, 2018 4:50 AM

```
// c++  code
vector<string> restoreIpAddresses(string s) {
    vector<string> ret;
    string ans;

    for (int a=1; a<=3; a++)
    for (int b=1; b<=3; b++)
    for (int c=1; c<=3; c++)
    for (int d=1; d<=3; d++)
        if (a+b+c+d == s.length()) {
            int A = stoi(s.substr(0, a));
            int B = stoi(s.substr(a, b));
            int C = stoi(s.substr(a+b, c));
            int D = stoi(s.substr(a+b+c, d));
            if (A<=255 && B<=255 && C<=255 && D<=255)
                if ( (ans=to_string(A)+"."+to_string(B)+"."+to_string(C)+"."+to_string(D)).length() == s.length()+3)
                    ret.push_back(ans);
        }

    return ret;
}
```

leetcodefan ★ 435 February 9, 2019 9:58 AM

Comrade Chen Duxiu, please sit down.

28 Show 2 replies Reply

94、题目描述（中等难度）

93. Restore IP Addresses

Medium 681 278 Favorite Share

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

Example:

```
Input: "25525511135"
Output: ["255.255.11.135", "255.255.111.35"]
```

给一个字符串，输出所有的可能的 ip 地址，注意一下，01.1.001.1 类似这种 0 开头的是非法字符串。

解法一 回溯 递归 DFS

很类似于刚做过的 91 题，对字符串进行划分。这个其实也是划分，划分的次数已经确定了，那就是分为 4 部分。那么就直接用回溯的思想，第一部分可能是 1 位数，然后进入递归。第一部分可能是 2 位数，然后进入递归。第一部分可能是 3 位数，然后进入递归。很好理解，直接看代码理解吧。

```
public List<String> restoreIpAddresses(String s) {
    List<String> ans = new ArrayList<>(); //保存最终的所有结果
    getAns(s, 0, new StringBuilder(), ans, 0);
    return ans;
}

/**
* @param:  start 字符串开始部分
* @param:  temp 已经划分的部分
* @param:  ans 保存所有的解
* @param:  count 当前已经加入了几部分
*/
private void getAns(String s, int start, StringBuilder temp, List<String> ans,
int count) {
    //如果剩余的长度大于剩下的部分都取 3 位数的长度，那么直接结束
    //例如 s = 121231312312, length = 12
    //当前 start = 1, count 等于 1
    //剩余字符串长度 11，剩余部分 4 - count = 3 部分，最多 3 * 3 是 9
    //所以一定是非法的，直接结束
    if (s.length() - start > 3 * (4 - count)) {
        return;
    }
    //当前刚好到达了末尾
    if (start == s.length()) {
        //当前刚好是 4 部分，将结果加入
        if (count == 4) {
            ans.add(new String(temp.substring(0, temp.length() - 1)));
        }
        return;
    }
    //当前超过末位，或者已经到达了 4 部分结束掉
    if (start > s.length() || count == 4) {
        return;
    }
    //保存的当前的解
    StringBuilder before = new StringBuilder(temp);

    //加入 1 位数
    temp.append(s.charAt(start) + "" + '.');
    getAns(s, start + 1, temp, ans, count + 1);

    //如果开头是 0，直接结束
```

```
    if (s.charAt(start) == '0')
        return;

    //加入 2 位数
    if (start + 1 < s.length()) {
        temp = new StringBuilder(before);//恢复为之前的解
        temp.append(s.substring(start, start + 2) + "" + '.');
        getAns(s, start + 2, temp, ans, count + 1);
    }

    //加入 3 位数
    if (start + 2 < s.length()) {
        temp = new StringBuilder(before);
        int num = Integer.parseInt(s.substring(start, start + 3));
        if (num >= 0 && num <= 255) {
            temp.append(s.substring(start, start + 3) + "" + '.');
            getAns(s, start + 3, temp, ans, count + 1);
        }
    }

}
```

解法二 迭代

参考这里，相当暴力直接。因为我们知道了，需要划分为 4 部分，所以我们直接用利用三个指针将字符串强行分为四部分，遍历所有的划分，然后选取合法的解。

```
public List<String> restoreIpAddresses(String s) {
    List<String> res = new ArrayList<String>();
    int len = s.length();
    //i < 4 保证第一部分不超过 3 位数
    //i < len - 2 保证剩余的字符串还能分成 3 部分
    for(int i = 1; i<4 && i<len-2; i++){
        for(int j = i+1; j<i+4 && j<len-1; j++){
            for(int k = j+1; k<j+4 && k<len; k++){
                //保存四部分的字符串
                String s1 = s.substring(0,i), s2 = s.substring(i,j), s3 =
s.substring(j,k), s4 = s.substring(k,len);
                //判断是否合法
                if(isValid(s1) && isValid(s2) && isValid(s3) && isValid(s4)){
                    res.add(s1+"."+s2+"."+s3+"."+s4);
                }
            }
        }
    }
```

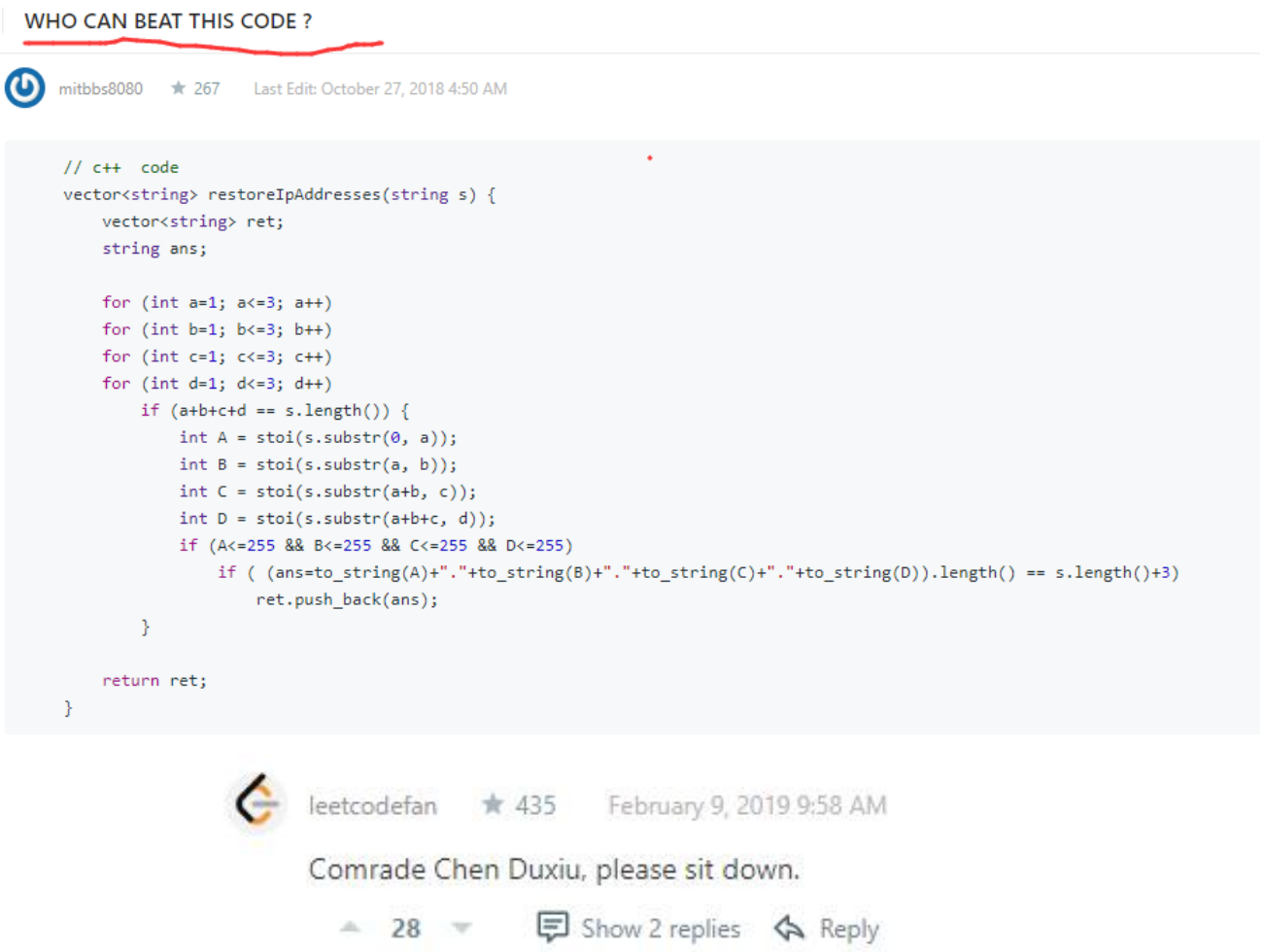
```
    return res;
}
public boolean isValid(String s){
    if(s.length()>3 || s.length()==0 || (s.charAt(0)=='0' && s.length()>1) ||
Integer.parseInt(s)>255)
        return false;
    return true;
}
```

时间复杂度：如果不考虑我们调用的内部函数，Integer.parseInt，s.substring，那么就是 O（1）。因为每一层循环最多遍历 4 次。考虑的话每次调用的时间复杂度是 O（n），常数次调用，所以是 O（n）。

空间复杂度：O（1）。

总

回溯或者说深度优先遍历，经常遇到了。但是解法二的暴力方法竟然通过了，有些意外。另外分享下 discuss 里有趣的评论，哈哈哈哈。



95、题目描述（中等难度）

94. Binary Tree Inorder Traversal

Medium 1672 72 Favorite Share

Given a binary tree, return the *inorder* traversal of its nodes' values.

Example:

```
Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [1,3,2]
```

Follow up: Recursive solution is trivial, could you do it iteratively?

二叉树的中序遍历。

解法一 递归

学二叉树的时候，必学的算法。用递归写简洁明了，就不多说了。

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> ans = new ArrayList<>();
    getAns(root, ans);
    return ans;
}

private void getAns(TreeNode node, List<Integer> ans) {
    if (node == null) {
        return;
    }
    getAns(node.left, ans);
    ans.add(node.val);
    getAns(node.right, ans);
}
```

时间复杂度：O（n），遍历每个节点。

空间复杂度：O（h），压栈消耗，h 是二叉树的高度。

官方解法中还提供了两种解法，这里总结下。

解法二 栈

利用栈，去模拟递归。递归压栈的过程，就是保存现场，就是保存当前的变量，而解法一中当前有用的变量就是 node，所以我们用栈把每次的 node 保存起来即可。

模拟下递归的过程，只考虑 node 的压栈。

```
//当前节点为空，出栈
if (node == null) {
    return;
}
//当前节点不为空
getAns(node.left, ans);  //压栈
ans.add(node.val); //出栈后添加
getAns(node.right, ans); //压栈
//左右子树遍历完，出栈
```

看一个具体的例子，想象一下吧。

```
        1
      /   \
     2     3
    / \   /
   4   5 6

 push   push   push   pop     pop    push     pop     pop
|   |  |   |  |_4_|  |   |   |   |  |   |    |   |   |   |
|   |  |_2_|  |_2_|  |_2_|   |   |  |_5_|    |   |   |   |
|_1_|  |_1_|  |_1_|  |_1_|   |_1_|  |_1_|    |_1_|   |   |
ans                  add 4   add 2           add 5   add 1
[]                   [4]     [4 2]           [4 2 5] [4 2 5 1]
 push   push   pop          pop
|   |  |   |  |   |        |   |
|   |  |_6_|  |   |        |   |
|_3_|  |_3_|  |_3_|        |   |
              add 6        add 3
              [4 2 5 1 6]  [4 2 5 1 6 3]
```

结合代码。

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> ans = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
```

```
    TreeNode cur = root;
    while (cur != null || !stack.isEmpty()) {
        //节点不为空一直压栈
        while (cur != null) {
            stack.push(cur);
            cur = cur.left; //考虑左子树
        }
        //节点为空，就出栈
        cur = stack.pop();
        //当前值加入
        ans.add(cur.val);
        //考虑右子树
        cur = cur.right;
    }
    return ans;
}
```

时间复杂度：O（n）。

空间复杂度：O（h），栈消耗，h 是二叉树的高度。

解法三 Morris Traversal

解法一和解法二本质上是一致的，都需要 O（h）的空间来保存上一层的信息。而我们注意到中序遍历，就是遍历完左子树，然后遍历根节点。如果我们把当前根节点存起来，然后遍历左子树，左子树遍历完以后回到当前根节点就可以了，怎么做到呢?

我们知道，左子树最后遍历的节点一定是一个叶子节点，它的左右孩子都是 null，我们把它右孩子指向当前根节点存起来，这样的话我们就不需要额外空间了。这样做，遍历完当前左子树，就可以回到根节点了。

当然如果当前根节点左子树为空，那么我们只需要保存根节点的值，然后考虑右子树即可。

所以总体思想就是：记当前遍历的节点为 cur。

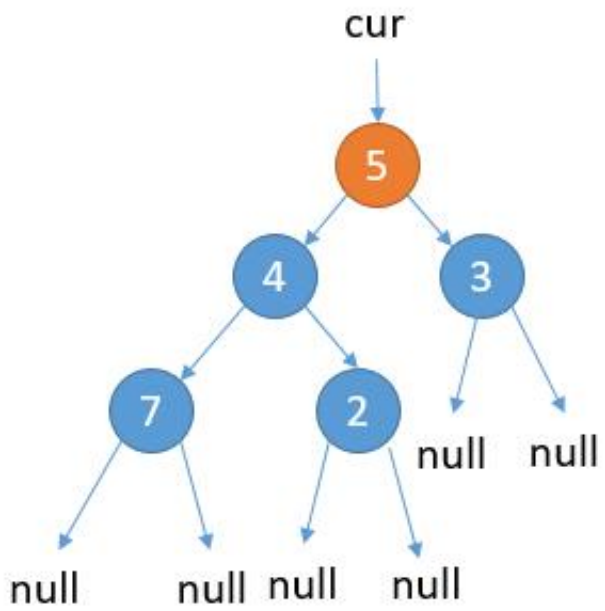
1、cur.left 为 null，保存 cur 的值，更新 cur = cur.right

2、cur.left 不为 null，找到 cur.left 这颗子树最右边的节点记做 last

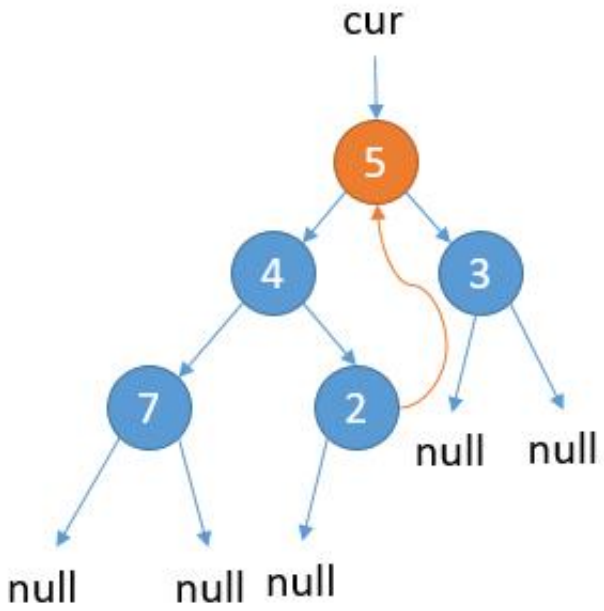
2.1 last.right 为 null，那么将 last.right = cur，更新 cur = cur.left

2.2 last.right 不为 null，说明之前已经访问过，第二次来到这里，表明当前子树遍历完成，保存 cur 的值，更新 cur = cur.right

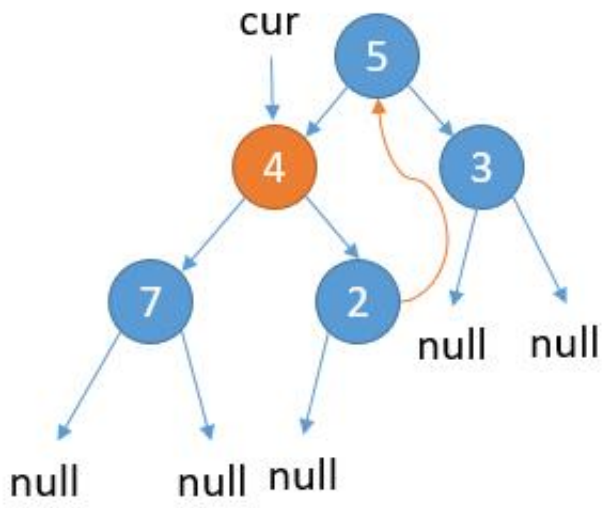
结合图示：



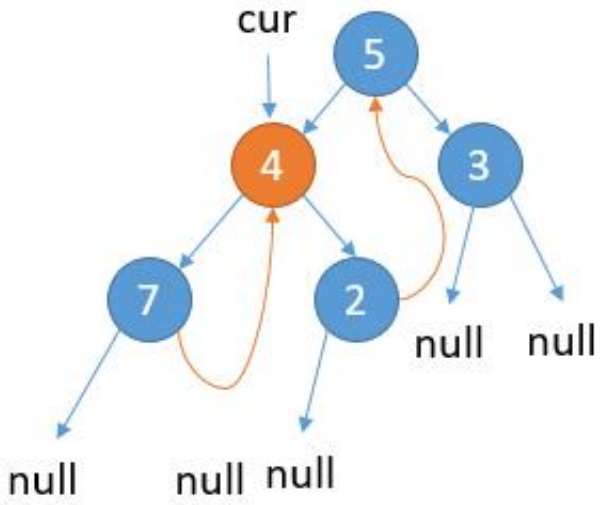
如上图，cur 指向根节点。 当前属于 2.1 的情况，cur.left 不为 null，cur 的左子树最右边的节点的右孩子为 null，那么我们把最右边的节点的右孩子指向 cur。



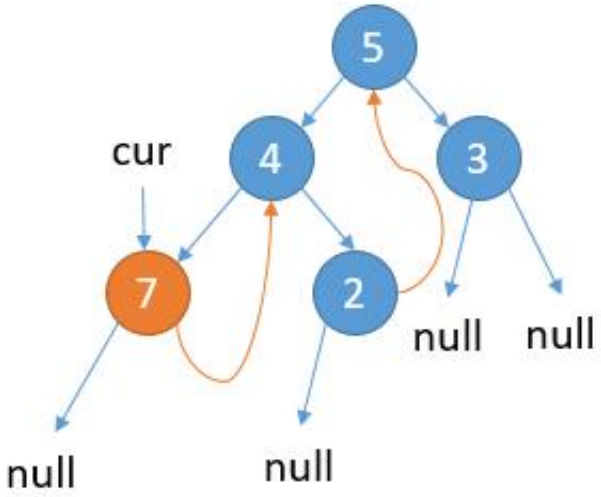
接着，更新 cur = cur.left。



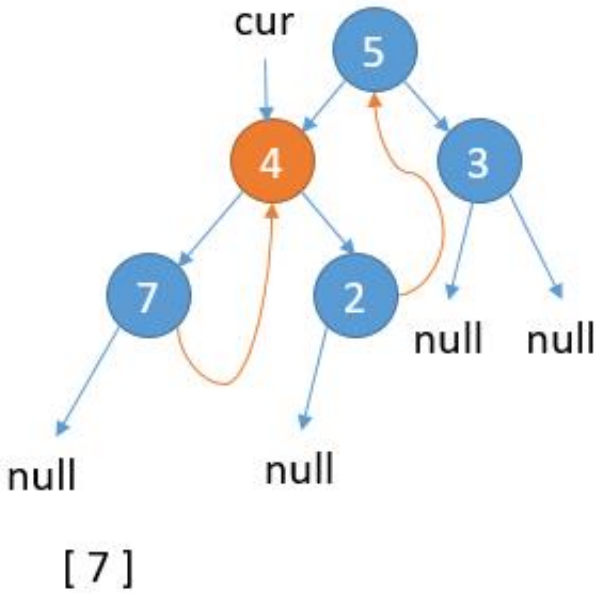
如上图，当前属于 2.1 的情况，cur.left 不为 null，cur 的左子树最右边的节点的右孩子为 null，那么我们把最右边的节点的右孩子指向 cur。



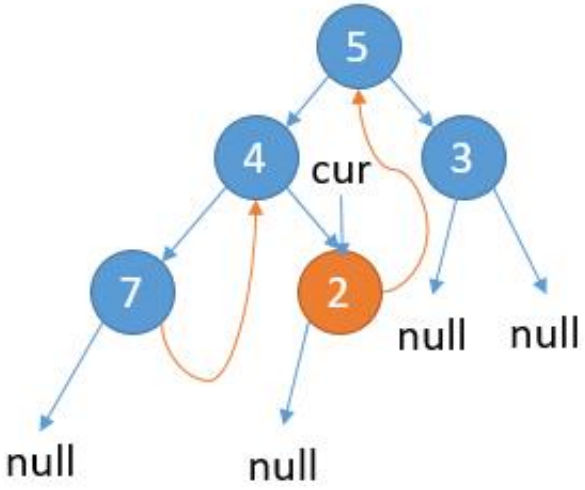
更新 cur = cur.left。



如上图，当前属于情况 1，cur.left 为 null，保存 cur 的值，更新 cur = cur.right。

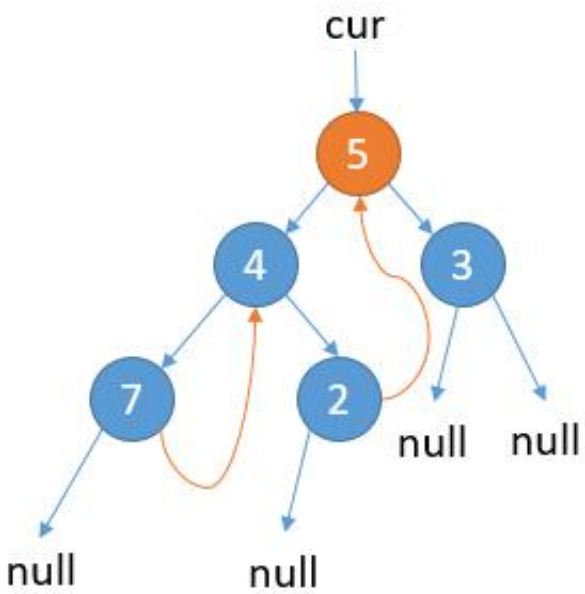


如上图，当前属于 2.2 的情况，cur.left 不为 null，cur 的左子树最右边的节点的右孩子已经指向 cur，保存 cur 的值，更新 cur = cur.right。



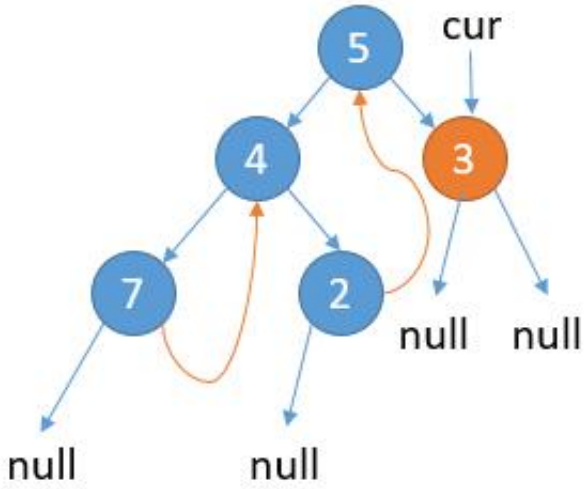
[7 4]

如上图，当前属于情况 1，cur.left 为 null，保存 cur 的值，更新 cur = cur.right。



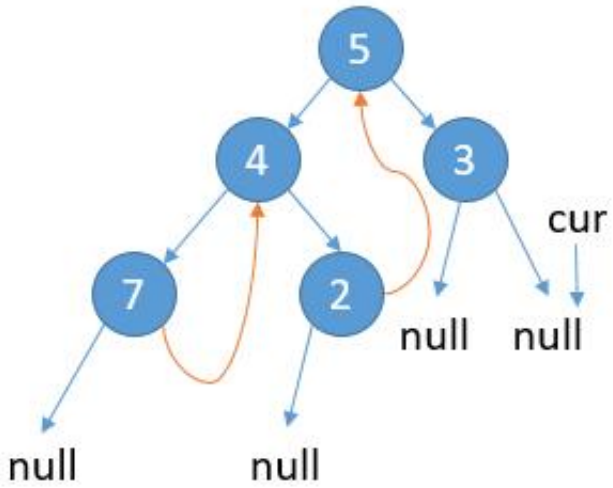
[7 4 2]

如上图，当前属于 2.2 的情况，cur.left 不为 null，cur 的左子树最右边的节点的右孩子已经指向 cur，保存 cur 的值，更新 cur = cur.right。



[7 4 2 5]

当前属于情况 1，cur.left 为 null，保存 cur 的值，更新 cur = cur.right。



[7 4 2 5 3]

cur 指向 null，结束遍历。

根据这个关系，写代码

记当前遍历的节点为 cur。

1、cur.left 为 null，保存 cur 的值，更新 cur = cur.right

2、cur.left 不为 null，找到 cur.left 这颗子树最右边的节点记做 last

2.1 last.right 为 null，那么将 last.right = cur，更新 cur = cur.left

2.2 last.right 不为 null，说明之前已经访问过，第二次来到这里，表明当前子树遍历完成，保存 cur 的值，更新 cur = cur.right

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> ans = new ArrayList<>();
    TreeNode cur = root;
    while (cur != null) {
        //情况 1
        if (cur.left == null) {
            ans.add(cur.val);
            cur = cur.right;
        } else {
            //找左子树最右边的节点
            TreeNode pre = cur.left;
            while (pre.right != null && pre.right != cur) {
                pre = pre.right;
            }
            //情况 2.1
            if (pre.right == null) {
                pre.right = cur;
                cur = cur.left;
            }
            //情况 2.2
            if (pre.right == cur) {
                pre.right = null; //这里可以恢复为 null
                ans.add(cur.val);
                cur = cur.right;
            }
        }
    }
    return ans;
}
```

时间复杂度：O（n）。每个节点遍历常数次。

空间复杂度：O（1）。

总

解法三是自己第一次见到，充分利用原来的空间的遍历，太强了。这么好的算法，当时上课的时候为什么没有讲，可惜了。

96、题目描述（中等难度）

95. Unique Binary Search Trees II

Medium 1275 116 Favorite Share

Given an integer *n*, generate all structurally unique **BST's** (binary search trees) that store values 1 ... *n*.

Example:

```
Input: 3
Output:
[
  [1,null,3,2],
  [3,2,null,1],
  [3,1,null,null,2],
  [2,1,3],
  [1,null,2,null,3]
]
Explanation:
The above output corresponds to the 5 unique BST's shown below:

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

给一个 n，用1...n 这些数字生成所有可能的二分查找树。所谓二分查找树，定义如下：

> 1. 若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
> 2. 若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
> 3. 任意节点的左、右子树也分别为二叉查找树；
> 4. 没有键值相等的节点。

解法一 回溯法

这是自己最早想到的一个思路。常规的回溯思想，就是普通的一个 for 循环，尝试插入 1, 2 ... n，然后进入递归，在原来的基础上继续尝试插入 1, 2... n。直到树包含了所有的数字。就是差不多下边这样的框架。

```
递归{
    递归出口;
    for(int i = 1;i<=n;i++){
      add(i);
      进入递归;
      remove(i);
  }
}
```

看一下详细的代码。

```
public List<TreeNode> generateTrees(int n) {
    List<TreeNode> ans = new ArrayList<TreeNode>();
    if (n == 0) {
        return ans;
    }
    TreeNode root = new TreeNode(0); //作为一个哨兵节点
    getAns(n, ans, root, 0);
    return ans;
}

private void getAns(int n, List<TreeNode> ans, TreeNode root, int count) {
    if (count == n) {
        //复制当前树并且加到结果中
        TreeNode newRoot = treeCopy(root);
        ans.add(newRoot.right);
        return;
    }
    TreeNode root_copy = root;
    //尝试插入每个数
    for (int i = 1; i <= n; i++) {
        root = root_copy;
        //寻找要插入的位置
        while (root != null) {
            //在左子树中插入
            if (i < root.val) {
                //到了最左边
                if (root.left==null) {
                    //插入当前数字
                    root.left = new TreeNode(i);
                    //进入递归
                    getAns(n, ans, root_copy, count + 1);
                    //还原为 null，尝试插入下个数字
                    root.left = null;
```

```
                    break;
                }
                root = root.left;
            //在右子树中插入
            } else if (i > root.val){
                 //到了最右边
                if (root.right == null){
                    //插入当前数字
                    root.right = new TreeNode(i);
                    //进入递归
                    getAns(n, ans, root_copy, count + 1);
                    //还原为 null，尝试插入下个数字
                    root.right = null;
                    break;
                }
                root = root.right;
             //如果找到了相等的数字，就结束，尝试下一个数字
            } else {
                break;
            }
        }
    }
}
```

然而，理想很美丽，现实很骨感，出错了，就是回溯经常遇到的问题，出现了重复解。

```
//第一种情况

第一次循环添加 2
    2

第二次循环添加 1
    2
  /
 1

第三次循环添加 3
    2
  /   \
 1     3

//第二种情况

第一次循环添加 2
    2
```

```
第二次循环添加 3
    2
     \
      3

第三次循环添加 1
    2
  /   \
 1     3
```

是的，因为每次循环都尝试了所有数字，所以造成了重复。所以接下来就是解决避免重复数字的发生，然而经过种种努力，都失败了，所以这种思路就此结束，如果大家想出了避免重复的方法，欢迎和我交流。

解法二 递归

解法一完全没有用到查找二叉树的性质，暴力尝试了所有可能从而造成了重复。我们可以利用一下查找二叉树的性质。左子树的所有值小于根节点，右子树的所有值大于根节点。

所以如果求 1...n 的所有可能。

我们只需要把 1 作为根节点，[ ] 空作为左子树，[2 ... n] 的所有可能作为右子树。

2 作为根节点，[1] 作为左子树，[3...n] 的所有可能作为右子树。

3 作为根节点，[1 2] 的所有可能作为左子树，[4 ... n] 的所有可能作为右子树，然后左子树和右子树两两组合。

4 作为根节点，[1 2 3] 的所有可能作为左子树，[5 ... n] 的所有可能作为右子树，然后左子树和右子树两两组合。

...

n 作为根节点，[1... n] 的所有可能作为左子树，[ ] 作为右子树。

至于，[2 ... n] 的所有可能以及 [4 ... n] 以及其他情况的所有可能，可以利用上边的方法，把每个数字作为根节点，然后把所有可能的左子树和右子树组合起来即可。

如果只有一个数字，那么所有可能就是一种情况，把该数字作为一棵树。而如果是 [ ]，那就返回 null。

```
public List<TreeNode> generateTrees(int n) {
    List<TreeNode> ans = new ArrayList<TreeNode>();
    if (n == 0) {
        return ans;
    }
    return getAns(1, n);

}
```

```
private List<TreeNode> getAns(int start, int end) {
    List<TreeNode> ans = new ArrayList<TreeNode>();
    //此时没有数字，将 null 加入结果中
    if (start > end) {
        ans.add(null);
        return ans;
    }
    //只有一个数字，当前数字作为一棵树加入结果中
    if (start == end) {
        TreeNode tree = new TreeNode(start);
        ans.add(tree);
        return ans;
    }
    //尝试每个数字作为根节点
    for (int i = start; i <= end; i++) {
        //得到所有可能的左子树
        List<TreeNode> leftTrees = getAns(start, i - 1);
         //得到所有可能的右子树
        List<TreeNode> rightTrees = getAns(i + 1, end);
        //左子树右子树两两组合
        for (TreeNode leftTree : leftTrees) {
            for (TreeNode rightTree : rightTrees) {
                TreeNode root = new TreeNode(i);
                root.left = leftTree;
                root.right = rightTree;
                //加入到最终结果中
                ans.add(root);
            }
        }
    }
    return ans;
}
```

解法三 动态规划

大多数递归都可以用动态规划的思想重写，这道也不例外。从底部往上走，参考这里。

举个例子，n = 3

```
数字个数是 0 的所有解
null
数字个数是 1 的所有解
1
2
3
```

数字个数是 2 的所有解，我们只需要考虑连续数字

[1 2]

```
  1
   \
    2
```

```
   2
  /
 1
```

[2 3]

```
  2
   \
    3
```

```
   3
  /
 2
```

如果求 3 个数字的所有情况。

[1 2 3]

利用解法二递归的思路，就是分别把每个数字作为根节点，然后考虑左子树和右子树的可能

1 作为根节点，左子树是 [] 的所有可能，右子树是 [2 3] 的所有可能，利用之前求出的结果进行组合。

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

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

2 作为根节点，左子树是 [1] 的所有可能，右子树是 [3] 的所有可能，利用之前求出的结果进行组合。

```
    2
  /   \
 1     3
```

3 作为根节点，左子树是 [1 2] 的所有可能，右子树是 [] 的所有可能，利用之前求出的结果进行组合。

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

```
      3
```

```
    /   \
   2   null
  /
 1
```

然后利用上边的思路基本上可以写代码了，就是求出长度为 1 的所有可能，长度为 2 的所有可能 ... 直到 n。

但是我们注意到，求长度为 2 的所有可能的时候，我们需要求 [1 2] 的所有可能，[2 3] 的所有可能，这只是 n = 3 的情况。如果 n 等于 100，我们需要求的更多了 [1 2]，[2 3]，[3 4] ... [99 100] 太多了。能不能优化呢?

仔细观察，我们可以发现长度是为 2 的所有可能其实只有两种结构。

```
  x
 /
y

y
 \
  x
```

看之前推导的 [1 2] 和 [2 3]，只是数字不一样，结构是完全一样的。

```
[ 1 2 ]
  1
   \
    2
   2
  /
 1

[ 2 3 ]
  2
   \
    3
   3
  /
 2
```

所以我们 n = 100 的时候，求长度是 2 的所有情况的时候，我们没必要把 [1 2]，[2 3]，[3 4] ... [99 100] 所有的情况都求出来，只需要求出 [1 2] 的所有情况即可。

推广到任意长度 len，我们其实只需要求 [1 2 ... len] 的所有情况就可以了。下一个问题随之而来，这些 [2 3]，[3 4] ... [99 100] 没求的怎么办呢?

举个例子。n = 100，此时我们求把 98 作为根节点的所有情况，根据之前的推导，我们需要长度是 97 的 [1 2 ... 97] 的所有情况作为左子树，长度是 2 的 [99 100] 的所有情况作为右子树。

[1 2 ... 97] 的所有情况刚好是 [1 2 ... len]，已经求出来了。但 [99 100] 怎么办呢？我们只求了 [1 2] 的所有情况。答案很明显了，在 [1 2] 的所有情况每个数字加一个偏差 98，即加上根节点的值就可以了。

```
[ 1 2 ]
  1
   \
    2
   2
  /
 1

[ 99 100 ]
  1 + 98
   \
    2 + 98
   2 + 98
  /
 1 + 98

即
  99
   \
    100
   100
  /
 99
```

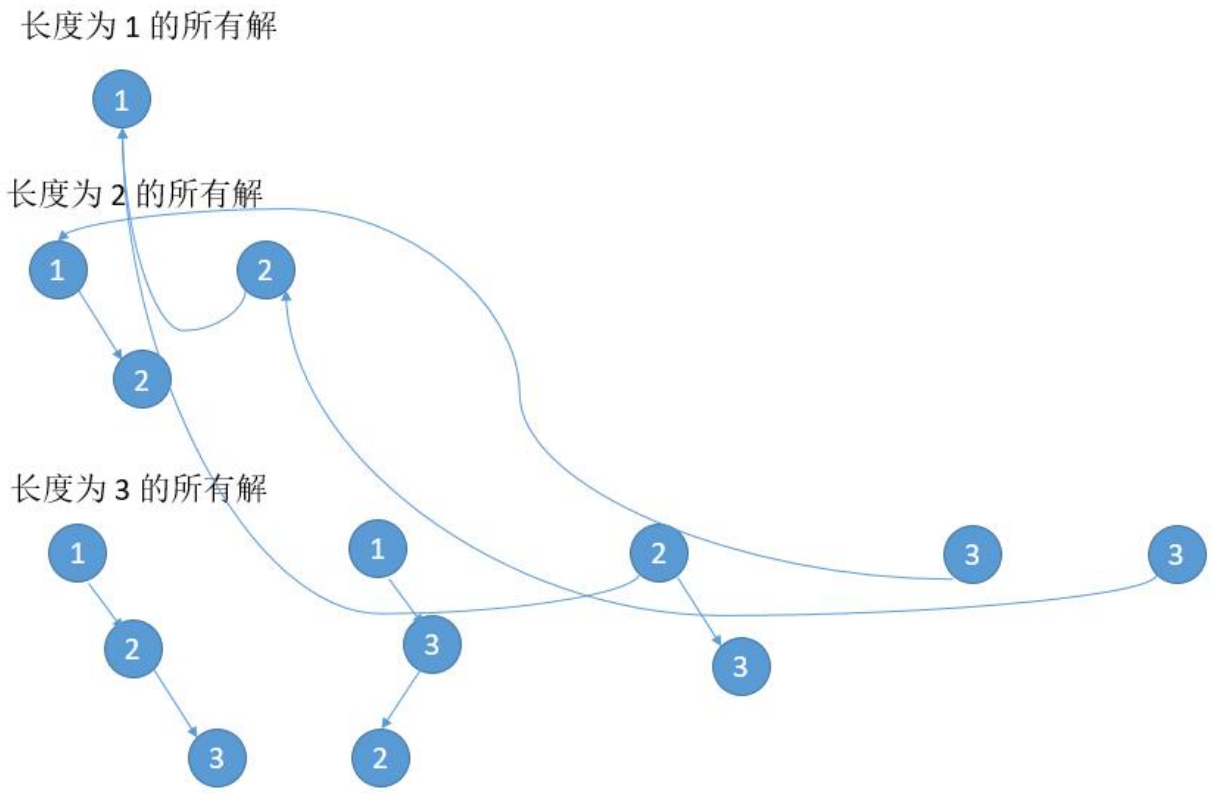
所以我们需要一个函数，实现树的复制并且加上偏差。

```
private TreeNode clone(TreeNode n, int offset) {
    if (n == null) {
        return null;
    }
    TreeNode node = new TreeNode(n.val + offset);
    node.left = clone(n.left, offset);
    node.right = clone(n.right, offset);
    return node;
}
```

通过上边的所有分析，代码可以写了，总体思想就是求长度为 2 的所有情况，求长度为 3 的所有情况直到 n。而求长度为 len 的所有情况，我们只需要求出一个代表 [1 2 ... len] 的所有情况，其他的话加上一个偏差，加上当前根节点即可。

```java
public List<TreeNode> generateTrees(int n) {
    ArrayList<TreeNode>[] dp = new ArrayList[n + 1];
    dp[0] = new ArrayList<TreeNode>();
    if (n == 0) {
        return dp[0];
    }
    dp[0].add(null);
    //长度为 1 到 n
    for (int len = 1; len <= n; len++) {
        dp[len] = new ArrayList<TreeNode>();
        //将不同的数字作为根节点，只需要考虑到 len
        for (int root = 1; root <= len; root++) {
            int left = root - 1;  //左子树的长度
            int right = len - root; //右子树的长度
            for (TreeNode leftTree : dp[left]) {
                for (TreeNode rightTree : dp[right]) {
                    TreeNode treeRoot = new TreeNode(root);
                    treeRoot.left = leftTree;
                    //克隆右子树并且加上偏差
                    treeRoot.right = clone(rightTree, root);
                    dp[len].add(treeRoot);
                }
            }
        }
    }
    return dp[n];
}

private TreeNode clone(TreeNode n, int offset) {
    if (n == null) {
        return null;
    }
    TreeNode node = new TreeNode(n.val + offset);
    node.left = clone(n.left, offset);
    node.right = clone(n.right, offset);
    return node;
}
```

值得注意的是，所有的左子树我们没有 clone ，也就是很多子树被共享了，在内存中就会是下边的样子。



也就是左子树用的都是之前的子树，没有开辟新的空间。

解法四 动态规划 2

解法三的动态规划完全是模仿了解法二递归的思想，这里再介绍另一种思想，会更好理解一些。参考这里。

```
考虑 [] 的所有解
null

考虑 [ 1 ] 的所有解
1

考虑 [ 1 2 ] 的所有解
  2
 /
1

 1
  \
   2

考虑 [ 1 2 3 ] 的所有解
    3
   /
```

```
  2
 /
1


  2
 / \
1   3


  3
 /
1
 \
  2


  1
   \
    3
   /
  2


  1
   \
    2
     \
      3
```

仔细分析，可以发现一个规律。首先我们每次新增加的数字大于之前的所有数字，所以新增加的数字出现的位置只可能是根节点或者是根节点的右孩子，右孩子的右孩子，右孩子的右孩子的右孩子等等，总之一定是右边。其次，新数字所在位置原来的子树，改为当前插入数字的左孩子即可，因为插入数字是最大的。

对于下边的解

```
  2
 /
1
```

然后增加 3

1.把 3 放到根节点

```
    3
   /
  2
 /
1
```

2. 把 3 放到根节点的右孩子

```
  2
```

```
  / \
 1   3
```

对于下边的解

```
 1
  \
   2
```

然后增加 3

1.把 3 放到根节点

```
    3
   /
  1
   \
    2
```

2. 把 3 放到根节点的右孩子，原来的子树作为 3 的左孩子

```
      1
       \
        3
       /
      2
```

3. 把 3 放到根节点的右孩子的右孩子

```
  1
   \
    2
     \
      3
```

以上就是根据 [1 2] 推出 [1 2 3] 的所有过程，可以写代码了。由于求当前的所有解只需要上一次的解，所有我们只需要两个 list，pre 保存上一次的所有解， cur 计算当前的所有解。

```
public List<TreeNode> generateTrees(int n) {
    List<TreeNode> pre = new ArrayList<TreeNode>();
    if (n == 0) {
        return pre;
    }
    pre.add(null);
    //每次增加一个数字
    for (int i = 1; i <= n; i++) {
        List<TreeNode> cur = new ArrayList<TreeNode>();
        //遍历之前的所有解
        for (TreeNode root : pre) {
            //插入到根节点
```

```
                TreeNode insert = new TreeNode(i);
                insert.left = root;
                cur.add(insert);
                //插入到右孩子，右孩子的右孩子...最多找 n 次孩子
                for (int j = 0; j <= n; j++) {
                    TreeNode root_copy = treeCopy(root); //复制当前的树
                    TreeNode right = root_copy; //找到要插入右孩子的位置
                    int k = 0;
                    //遍历 j 次找右孩子
                    for (; k < j; k++) {
                        if (right == null)
                            break;
                        right = right.right;
                    }
                    //到达 null 提前结束
                    if (right == null)
                        break;
                    //保存当前右孩子的位置的子树作为插入节点的左孩子
                    TreeNode rightTree = right.right;
                    insert = new TreeNode(i);
                    right.right = insert; //右孩子是插入的节点
                    insert.left = rightTree; //插入节点的左孩子更新为插入位置之前的子树
                    //加入结果中
                    cur.add(root_copy);
                }
            }
            pre = cur;

        }
        return pre;
    }


    private TreeNode treeCopy(TreeNode root) {
        if (root == null) {
            return root;
        }
        TreeNode newRoot = new TreeNode(root.val);
        newRoot.left = treeCopy(root.left);
        newRoot.right = treeCopy(root.right);
        return newRoot;
    }
```

总

解法二和解法四算作常规的思路，比较容易想到。解法三，发现同构的操作真的是神仙操作了，服！

97、题目描述（中等难度）

96. Unique Binary Search Trees

Medium 1769 68 Favorite Share

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1 ... *n*?

Example:

```
Input: 3
Output: 5
Explanation:
Given n = 3, there are a total of 5 unique BST's:

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

和 95 题一样，只不过这道题不需要输出所有的树，只需要输出所有可能的二分查找树的数量。所以完全按照 95 题思路写，大家可以先到 95 题看一看。

解法一 递归

下边是 95 题的分析。

> 我们可以利用一下查找二叉树的性质。左子树的所有值小于根节点，右子树的所有值大于根节点。
>
> 所以如果求 1...n 的所有可能。
>
> 我们只需要把 1 作为根节点，[ ] 空作为左子树，[2 ... n] 的所有可能作为右子树。
>
> 2 作为根节点，[1] 作为左子树，[3...n] 的所有可能作为右子树。
>
> 3 作为根节点，[1 2] 的所有可能作为左子树，[4 ... n] 的所有可能作为右子树，然后左子树和右子树两两组合。
>
> 4 作为根节点，[1 2 3] 的所有可能作为左子树，[5 ... n] 的所有可能作为右子树，然后左子树和右子树两两组合。
>
> ...
>
> n 作为根节点，[1... n] 的所有可能作为左子树，[ ] 作为右子树。

> 至于，[2 ... n] 的所有可能以及 [4 ... n] 以及其他情况的所有可能，可以利用上边的方法，把每个数字作为根节点，然后把所有可能的左子树和右子树组合起来即可。
>
> 如果只有一个数字，那么所有可能就是一种情况，把该数字作为一棵树。而如果是 [ ]，那就返回 null。

对于这道题，我们会更简单些，只需要返回树的数量即可。求当前根的数量，只需要左子树的数量乘上右子树。

```
public int numTrees(int n) {
    if (n == 0) {
        return 0;
    }
    return getAns(1, n);

}
private int getAns(int start, int end) {
    int ans = 0;
    //此时没有数字，只有一个数字,返回 1
    if (start >= end) {
        return 1;
    }
    //尝试每个数字作为根节点
    for (int i = start; i <= end; i++) {
        //得到所有可能的左子树
        int leftTreesNum = getAns(start, i - 1);
        //得到所有可能的右子树
        int rightTreesNum  = getAns(i + 1, end);
        //左子树右子树两两组合
        ans+=leftTreesNum * rightTreesNum;
    }
    return ans;
}
```

受到这里的启发，我们甚至可以改写的更简单些。因为 95 题要把每颗树返回，所有传的参数是 start 和 end。这里的话，我们只关心数量，所以不需要具体的范围，而是传树的节点的数量即可。

```
public int numTrees(int n) {
    if (n == 0) {
        return 0;
    }
    return getAns(n);

}

private int getAns(int n) {
```

```
    int ans = 0;
    //此时没有数字或者只有一个数字,返回 1
    if (n==0 ||n==1) {
        return 1;
    }
    //尝试每个数字作为根节点
    for (int i = 1; i <= n; i++) {
        //得到所有可能的左子树
        // i - 1 代表左子树节点的数量
        int leftTreesNum = getAns(i-1);
        //得到所有可能的右子树
        //n - i 代表左子树节点的数量
        int rightTreesNum  = getAns(n-i);
        //左子树右子树两两组合
        ans+=leftTreesNum * rightTreesNum;
    }
    return ans;
}
```

然后，由于递归的分叉，所以会导致很多重复解的计算，所以使用 memoization 技术，把递归过程中求出的解保存起来，第二次需要的时候直接拿即可。

```
public int numTrees(int n) {
    if (n == 0) {
        return 0;
    }
    HashMap<Integer,Integer> memoization = new HashMap<>();
    return getAns(n,memoization);

}

private int getAns(int n, HashMap<Integer,Integer> memoization) {
    if(memoization.containsKey(n)){
        return memoization.get(n);
    }
    int ans = 0;
    //此时没有数字，只有一个数字,返回 1
    if (n==0 ||n==1) {
        return 1;
    }
    //尝试每个数字作为根节点
    for (int i = 1; i <= n; i++) {
        //得到所有可能的左子树
        int leftTreesNum = getAns(i-1,memoization);
        //得到所有可能的右子树
```

```
            int rightTreesNum  = getAns(n-i,memoization);
            //左子树右子树两两组合
            ans+=leftTreesNum * rightTreesNum;
        }
        memoization.put(n, ans);
        return ans;
    }
```

解法二 动态规划

直接利用95题解法三的思路，讲解比较长就不贴过来了，可以过去看一下。

或者直接从这里的解法一的思路考虑，因为递归是从顶层往下走，压栈压栈压栈，到了长度是 0 或者是 1 就出栈出栈出栈。我们可以利用动态规划的思想，直接从底部往上走。求出长度是 0，长度是 1，长度是 2....长度是 n 的解。用一个数组 dp 把这些结果全部保存起来。

```
public int numTrees(int n) {
    int[] dp = new int[n + 1];
    dp[0] = 1;
    if (n == 0) {
        return 0;
    }
    // 长度为 1 到 n
    for (int len = 1; len <= n; len++) {
        // 将不同的数字作为根节点，只需要考虑到 len
        for (int root = 1; root <= len; root++) {
            int left = root - 1; // 左子树的长度
            int right = len - root; // 右子树的长度
            dp[len] += dp[left] * dp[right];
        }
    }
    return dp[n];
}
```

参考这里还有优化的空间。

利用对称性，可以使得循环减少一些。

- n 是偶数的时候
  1 2 | 3 4 ，for 循环中我们以每个数字为根求出每个的解。我们其实可以只求一半，根据对称性我们可以知道 1 和 4，2 和 3 求出的解分别是相等的。
- n 是奇数的时候

  1 2 | 3 | 4 5，和偶数同理，只求一半，此外最中间的 3 的解也要加上。

```
public int numTrees6(int n) {
```

```
    if (n == 0) {
        return 0;
    }
    int[] dp = new int[n + 1];
    dp[0] = 1;
    dp[1] = 1;
    // 长度为 1 到 n
    for (int len = 2; len <= n; len++) {
        // 将不同的数字作为根节点，只需要考虑到 len
        for (int root = 1; root <= len / 2; root++) {
            int left = root - 1; // 左子树的长度
            int right = len - root; // 右子树的长度
            dp[len] += dp[left] * dp[right];
        }
        dp[len] *= 2;// 利用对称性乘 2
        // 考虑奇数的情况
        if ((len & 1) == 1) {
            int root = (len >> 1) + 1;
            int left = root - 1; // 左子树的长度
            int right = len - root; // 右子树的长度
            dp[len] += dp[left] * dp[right];
        }
    }
    return dp[n];
}
```

解法三 公式法

参考这里。其实利用的是卡塔兰数列，这是第二次遇到了，之前是第 22 题 ，生成合法的括号序列。

这道题，为什么和卡塔兰数列联系起来呢?

看一下卡塔兰树数列的定义：

> 令h(0)=1，catalan 数满足递推式：
>
> **h(n)= h(0)*h(n-1)+h(1)*h(n-2) +... +h(n-1)*h(0) (n>=1)**
>
> 例如：h(2)=h(0)*h(1)+h(1)*h(0)=1*1+1*1=2
>
> h(3)=h(0)*h(2)+h(1)*h(1)+h(2)*h(0)=1*2+1*1+2*1=5

再看看解法二的算法

```
public int numTrees(int n) {
    int[] dp = new int[n + 1];
    dp[0] = 1;
    if (n == 0) {
```

```
        return 0;
    }
    // 长度为 1 到 n
    for (int len = 1; len <= n; len++) {
        // 将不同的数字作为根节点，只需要考虑到 len
        for (int root = 1; root <= len; root++) {
            int left = root - 1; // 左子树的长度
            int right = len - root; // 右子树的长度
            dp[len] += dp[left] * dp[right];
        }
    }
    return dp[n];
}
```

完美符合，而卡塔兰数有一个通项公式。

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

注：$$\binom{2n}{n}$$ 代表 $$C^n_{2n}$$

化简一下上边的公式

$$C_n = (2n)!/(n+1)!n! = (2n)*(2n-1)*...*(n+1)/(n+1)!$$

所以用一个循环即可。

```
int numTrees(int n) {
    long ans = 1, i;
    for (i = 1; i <= n; i++)
        ans = ans * (i + n) / i;
    return (int) (ans / i);
}
```

总

上道题会了以后，这道题很好写。解法二中利用对称的优化，解法三的公式太强了。

98、题目描述（中等难度）

98. Validate Binary Search Tree

Medium 2061 301 Favorite Share

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:

```
    2
   / \
  1   3

Input: [2,1,3]
Output: true
```

Example 2:

```
    5
   / \
  1   4
     / \
    3   6

Input: [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.
```

输入一个树，判断该树是否是合法二分查找树，95题做过生成二分查找树。二分查找树定义如下：

> 1. 若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
> 2. 若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
> 3. 任意节点的左、右子树也分别为二叉查找树；
> 4. 没有键值相等的节点。

解法一

开始的时候以为可以很简单的用递归写出来。想法是，左子树是合法二分查找树，右子树是合法二分查找树，并且根节点大于左孩子，小于右孩子，那么当前树就是合法二分查找树。代码如下：

```
public boolean isValidBST(TreeNode root) {
```

```
    if (root == null) {
        return true;
    }
    boolean leftVailid = true;
    boolean rightVaild = true;
    if (root.left != null) {
        //大于左孩子并且左子树是合法二分查找树
        leftVailid = root.val > root.left.val && isValidBST(root.left);
    }
    if (!leftVailid) {
        return false;
    }
    if (root.right != null) {
        //小于右孩子并且右子树是合法二分查找树
        rightVaild = root.val < root.right.val && isValidBST(root.right);
    }
    return rightVaild;
}
```

当然，这个解法没有通过。对于下面的解，结果利用上边的解法是错误的。

```
     10
    /  \
   5   15
      /  \
     6   20
```

虽然满足左子树是合法二分查找树，右子树是合法二分查找树，并且根节点大于左孩子，小于右孩子，但这个树不是合法的二分查找树。因为右子树中的 6 小于当前根节点 10。所以我们不应该判断「根节点大于左孩子，小于右孩子」，而是判断「根节点大于左子树中最大的数，小于右子树中最小的数」。

```
public boolean isValidBST(TreeNode root) {
    if (root == null || root.left == null && root.right == null) {
        return true;
    }
    //左子树是否合法
    if (isValidBST(root.left)) {
        if (root.left != null) {
            int max = getMaxOfBST(root.left);//得到左子树中最大的数
            if (root.val <= max) { //相等的情况，代表有重复的数字
                return false;
            }
        }

    } else {
```

```
            return false;
        }

        //右子树是否合法
        if (isValidBST(root.right)) {
            if (root.right != null) {
                int min = getMinOfBST(root.right);//得到右子树中最小的数
                if (root.val >= min) { //相等的情况，代表有重复的数字
                    return false;
                }
            }

        } else {
            return false;
        }
        return true;
    }

    private int getMinOfBST(TreeNode root) {
        int min = root.val;
        while (root != null) {
            if (root.val <= min) {
                min = root.val;
            }
            root = root.left;
        }
        return min;
    }

    private int getMaxOfBST(TreeNode root) {
        int max = root.val;
        while (root != null) {
            if (root.val >= max) {
                max = root.val;
            }
            root = root.right;
        }
        return max;
    }
```

解法二

来利用另一种思路，参考官方题解。

解法一中，我们是判断根节点是否合法，找到了左子树中最大的数，右子树中最小的数。 由左子树和右子树决定当前根节点是否合法。

但如果正常的来讲，明明先有的根节点，按理说根节点是任何数都行，而不是由左子树和右子树限定。相反，根节点反而决定了左孩子和右孩子的合法取值范围。

所以，我们可以从根节点进行 DFS，然后计算每个节点应该的取值范围，如果当前节点不符合就返回 false。

```
      10
    /    \
   5      15
  / \     /
 3   6   7

  考虑 10 的范围
    10(-inf,+inf)

  考虑 5 的范围
    10(-inf,+inf)
   /
  5(-inf,10)

  考虑 3 的范围
      10(-inf,+inf)
     /
  5(-inf,10)
   /
 3(-inf,5)

  考虑 6 的范围
      10(-inf,+inf)
     /
  5(-inf,10)
   /       \
 3(-inf,5)  6(5,10)

  考虑 15 的范围
     10(-inf,+inf)
   /          \
  5(-inf,10) 15(10,+inf)
   /       \
 3(-inf,5)  6(5,10)

  考虑 7 的范围，出现不符合返回 false
     10(-inf,+inf)
   /              \
5(-inf,10)          15(10,+inf)
  /       \            /
```

```
3(-inf,5)  6(5,10)   7 (10,15)
```

可以观察到，左孩子的范围是 （父结点左边界，父节点的值），右孩子的范围是（父节点的值，父节点的右边界）。

还有个问题，java 里边没有提供负无穷和正无穷，用什么数来表示呢?

方案一，假设我们的题目的数值都是 Integer 范围的，那么我们用不在 Integer 范围的数字来表示负无穷和正无穷。用 long 去存储。

```
public boolean isValidBST(TreeNode root) {
    long maxValue = (long)Integer.MAX_VALUE + 1;
    long minValue = (long)Integer.MIN_VALUE - 1;
    return getAns(root, minValue, maxValue);
}

private boolean getAns(TreeNode node, long minVal, long maxVal) {
    if (node == null) {
        return true;
    }
    if (node.val <= minVal) {
        return false;
    }
    if (node.val >= maxVal) {
        return false;
    }
    return getAns(node.left, minVal, node.val) && getAns(node.right, node.val,
maxVal);
}
```

方案二：传入 Integer 对象，然后 null 表示负无穷和正无穷。然后利用 JAVA 的自动装箱拆箱，数值的比较可以直接用不等号。

```
public boolean isValidBST(TreeNode root) {
    return getAns(root, null, null);
}

private boolean getAns(TreeNode node, Integer minValue, Integer maxValue) {
    if (node == null) {
        return true;
    }
    if (minValue != null && node.val <= minValue) {
```

```
        return false;
    }
    if (maxValue != null && node.val >= maxValue) {
        return false;
    }
    return getAns(node.left, minValue, node.val) && getAns(node.right, node.val,
maxValue);
}
```

解法三 DFS BFS

解法二其实就是树的 DFS，也就是二叉树的先序遍历，然后在遍历过程中，判断当前的值是是否在区间中。所以我们可以用栈来模拟递归过程。

```
public boolean isValidBST(TreeNode root) {
    if (root == null || root.left == null && root.right == null) {
        return true;
    }
    //利用三个栈来保存对应的节点和区间
    LinkedList<TreeNode> stack = new LinkedList<>();
    LinkedList<Integer> minValues = new LinkedList<>();
    LinkedList<Integer> maxValues = new LinkedList<>();
    //头结点入栈
    TreeNode pNode = root;
    stack.push(pNode);
    minValues.push(null);
    maxValues.push(null);
    while (pNode != null || !stack.isEmpty()) {
        if (pNode != null) {
            //判断栈顶元素是否符合
            Integer minValue = minValues.peek();
            Integer maxValue = maxValues.peek();
            TreeNode node = stack.peek();
            if (minValue != null && node.val <= minValue) {
                return false;
            }
            if (maxValue != null && node.val >= maxValue) {
                return false;
            }
            //将左孩子加入到栈
            if(pNode.left!=null){
                stack.push(pNode.left);
                minValues.push(minValue);
                maxValues.push(pNode.val);
            }
```

```
            pNode = pNode.left;
        } else { // pNode == null && !stack.isEmpty()
            //出栈，将右孩子加入栈中
            TreeNode node = stack.pop();
            minValues.pop();
            Integer maxValue = maxValues.pop();
            if(node.right!=null){
                stack.push(node.right);
                minValues.push(node.val);
                maxValues.push(maxValue);
            }
            pNode = node.right;
        }
    }
    return true;
}
```

上边的 DFS 可以看出来一个缺点，就是我们判断完当前元素后并没有出栈，后续还会回来得到右孩子后才会出栈。所以其实我们可以用 BFS，利用一个队列，一层一层的遍历，遍历完一个就删除一个。

```
public boolean isValidBST(TreeNode root) {
    if (root == null || root.left == null && root.right == null) {
        return true;
    }
    //利用三个队列来保存对应的节点和区间
    Queue<TreeNode> queue = new LinkedList<>();
    Queue<Integer> minValues = new LinkedList<>();
    Queue<Integer> maxValues = new LinkedList<>();
    //头结点入队列
    TreeNode pNode = root;
    queue.offer(pNode);
    minValues.offer(null);
    maxValues.offer(null);
    while (!queue.isEmpty()) {
        //判断队列的头元素是否符合条件并且出队列
        Integer minValue = minValues.poll();
        Integer maxValue = maxValues.poll();
        pNode = queue.poll();
        if (minValue != null && pNode.val <= minValue) {
            return false;
        }
        if (maxValue != null && pNode.val >= maxValue) {
            return false;
        }
```

```
            //左孩子入队列
            if(pNode.left!=null){
                queue.offer(pNode.left);
                minValues.offer(minValue);
                maxValues.offer(pNode.val);
            }
            //右孩子入队列
            if(pNode.right!=null){
                queue.offer(pNode.right);
                minValues.offer(pNode.val);
                maxValues.offer(maxValue);
            }
        }
        return true;
    }
```

解法四 中序遍历

参考这里。

解法三中我们用了先序遍历 和 BFS，现在来考虑中序遍历。中序遍历在 94 题中已经考虑过了。那么中序遍历在这里有什么好处呢?

中序遍历顺序会是左孩子，根节点，右孩子。二分查找树的性质，左孩子小于根节点，根节点小于右孩子。

是的，如果我们将中序遍历的结果输出，那么将会到的一个从小到大排列的序列。

所以我们只需要进行一次中序遍历，将遍历结果保存，然后判断该数组是否是从小到大排列的即可。

更近一步，由于我们只需要临近的两个数的相对关系，所以我们只需要在遍历过程中，把当前遍历的结果和上一个结果比较即可。

```
public boolean isValidBST(TreeNode root) {
    if (root == null) return true;
    Stack<TreeNode> stack = new Stack<>();
    TreeNode pre = null;
    while (root != null || !stack.isEmpty()) {
        while (root != null) {
            stack.push(root);
            root = root.left;
        }
        root = stack.pop();
        if(pre != null && root.val <= pre.val) return false;
        pre = root;
        root = root.right;
    }
    return true;
```

```
}
```

总

这几天都是二叉树的相关题，主要是对前序遍历，中序遍历的理解，以及 DFS，如果再用好递归，利用栈模拟递归，题目就很好解了。

99、题目描述（困难难度）

99. Recover Binary Search Tree

Description | Hints | Submissions | Discuss | Solution

Pick One

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Example 1:

```
Input: [1,3,null,null,2]

   1
  /
 3
  \
   2

Output: [3,1,null,null,2]

   3
  /
 1
  \
   2
```

Example 2:

```
Input: [3,1,4,null,null,2]

  3
 / \
1   4
   /
  2

Output: [2,1,4,null,null,3]

  2
 / \
1   4
   /
  3
```

Follow up:

- A solution using O(*n*) space is pretty straight forward.
- Could you devise a constant space solution?

依旧是二分查找树的题，一个合法的二分查找树随机交换了两个数的位置，然后让我们恢复二分查找树。不能改变原来的结构，只是改变两个数的位置。二分查找树定义如下：

> 1. 若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
> 2. 若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
> 3. 任意节点的左、右子树也分别为二叉查找树；
> 4. 没有键值相等的节点。

解法一 递归

和 98 题有些像。这里的思路如下：

让我们来考虑交换的位置的可能：

1. 根节点和左子树的某个数字交换 -> 由于根节点大于左子树中的所有数，所以交换后我们只要找左子树中最大的那个数，就是所交换的那个数
2. 根节点和右子树的某个数字交换 -> 由于根节点小于右子树中的所有数，所以交换后我们只要在右子树中最小的那个数，就是所交换的那个数
3. 左子树和右子树的两个数字交换 -> 找左子树中最大的数，右子树中最小的数，即对应两个交换的数
4. 左子树中的两个数字交换
5. 右子树中的两个数字交换

思想有了，代码很好写了。

```
public void recoverTree2(TreeNode root) {
    if (root == null) {
        return;
    }
    //寻找左子树中最大的节点
    TreeNode maxLeft = getMaxOfBST(root.left);
    //寻找右子树中最小的节点
    TreeNode minRight = getMinOfBST(root.right);

    if (minRight != null && maxLeft != null) {
        //左边的大于根节点，右边的小于根节点，对应情况 3，左右子树中的两个数字交换
        if ( maxLeft.val > root.val && minRight.val < root.val) {
            int temp = minRight.val;
            minRight.val = maxLeft.val;
            maxLeft.val = temp;
        }
    }

    if (maxLeft != null) {
        //左边最大的大于根节点，对应情况 1，根节点和左子树的某个数做了交换
        if (maxLeft.val > root.val) {
            int temp = maxLeft.val;
            maxLeft.val = root.val;
            root.val = temp;
        }
    }

    if (minRight != null) {
        //右边最小的小于根节点，对应情况 2，根节点和右子树的某个数做了交换
        if (minRight.val < root.val) {
```

```
            int temp = minRight.val;
            minRight.val = root.val;
            root.val = temp;
        }
    }
    //对应情况 4，左子树中的两个数进行了交换
    recoverTree(root.left);
    //对应情况 5，右子树中的两个数进行了交换
    recoverTree(root.right);

}
//寻找树中最小的节点
private TreeNode getMinOfBST(TreeNode root) {
    if (root == null) {
        return null;
    }
    TreeNode minLeft = getMinOfBST(root.left);
    TreeNode minRight = getMinOfBST(root.right);
    TreeNode min = root;
    if (minLeft != null && min.val > minLeft.val) {
        min = minLeft;
    }
    if (minRight != null && min.val > minRight.val) {
        min = minRight;
    }
    return min;
}

//寻找树中最大的节点
private TreeNode getMaxOfBST(TreeNode root) {
    if (root == null) {
        return null;
    }
    TreeNode maxLeft = getMaxOfBST(root.left);
    TreeNode maxRight = getMaxOfBST(root.right);
    TreeNode max = root;
    if (maxLeft != null && max.val < maxLeft.val) {
        max = maxLeft;
    }
    if (maxRight != null && max.val < maxRight.val) {
        max = maxRight;
    }
    return max;
}
```

解法二

参考 这里。

如果记得 98 题，我们判断是否是一个合法的二分查找树是使用到了中序遍历。原因就是二分查找树的一个性质，左孩子小于根节点，根节点小于右孩子。所以做一次中序遍历，产生的序列就是从小到大排列的有序序列。

回到这道题，题目交换了两个数字，其实就是在有序序列中交换了两个数字。而我们只需要把它还原。

交换的位置的话就是两种情况。

- 相邻的两个数字交换

  [1 2 3 4 5] 中 2 和 3 进行交换，[1 3 2 4 5]，这样的话只产生一组逆序的数字（正常情况是从小到大排序，交换后产生了从大到小），3 2。

  我们只需要遍历数组，找到后，把这一组的两个数字进行交换即可。

- 不相邻的两个数字交换

  [1 2 3 4 5] 中 2 和 5 进行交换，[1 5 3 4 2]，这样的话其实就是产生了两组逆序的数字对。5 3 和 4 2。

  所以我们只需要遍历数组，然后找到这两组逆序对，然后把第一组前一个数字和第二组后一个数字进行交换即完成了还原。

所以在中序遍历中，只需要利用一个 pre 节点和当前节点比较，如果 pre 节点的值大于当前节点的值，那么就是我们要找的逆序的数字。分别用两个指针 first 和 second 保存即可。如果找到第二组逆序的数字，我们就把 second 更新为当前节点。最后把 first 和 second 两个的数字交换即可。

中序遍历，参考 94 题，有三种方法，递归，栈，Morris 。这里的话，我们都改一下。

递归版中序遍历

```
TreeNode first = null;
TreeNode second = null;
public void recoverTree(TreeNode root) {
    inorderTraversal(root);
    int temp = first.val;
    first.val = second.val;
    second.val = temp;
}
TreeNode pre = null;
private void inorderTraversal(TreeNode root) {
    if (root == null) {
        return;
    }
    inorderTraversal(root.left);
    /*******************************************************/
```

```
    if(pre != null && root.val < pre.val) {
        //第一次遇到逆序对
        if(first==null){
            first = pre;
            second = root;
        //第二次遇到逆序对
        }else{
            second = root;
        }
    }
    pre = root;
    /*****************************************************/
    inorderTraversal(root.right);
}
```

栈版中序遍历

```
TreeNode first = null;
TreeNode second = null;

public void recoverTree(TreeNode root) {
    inorderTraversal(root);
    int temp = first.val;
    first.val = second.val;
    second.val = temp;
}

public void inorderTraversal(TreeNode root) {
    if (root == null)
        return;
    Stack<TreeNode> stack = new Stack<>();
    TreeNode pre = null;
    while (root != null || !stack.isEmpty()) {
        while (root != null) {
            stack.push(root);
            root = root.left;
        }
        root = stack.pop();
        /*****************************************************/
        if (pre != null && root.val < pre.val) {
            if (first == null) {
                first = pre;
                second = root;
            } else {
                second = root;
```

```
            }
        }
        pre = root;
        /**********************************************************/
        root = root.right;
    }
}
```

Morris 版中序遍历

因为之前这个方法中用了 pre 变量，为了方便，这里也需要 pre 变量，我们用 pre_new 代替。具体 Morris 遍历算法参见 94 题 。利用 Morris 的话，我们的空间复杂度终于达到了 O（1）。

```
public void recoverTree(TreeNode root) {
    TreeNode first = null;
    TreeNode second = null;
    TreeNode cur = root;
    TreeNode pre_new = null;
    while (cur != null) {
        // 情况 1
        if (cur.left == null) {
            /**********************************************************/
            if (pre_new != null && cur.val < pre_new.val) {
                if (first == null) {
                    first = pre_new;
                    second = cur;
                } else {
                    second = cur;
                }
            }
            pre_new = cur;
            /**********************************************************/
            cur = cur.right;
        } else {
            // 找左子树最右边的节点
            TreeNode pre = cur.left;
            while (pre.right != null && pre.right != cur) {
                pre = pre.right;
            }
            // 情况 2.1
            if (pre.right == null) {
                pre.right = cur;
                cur = cur.left;
            }
            // 情况 2.2
```

```
            if (pre.right == cur) {
                pre.right = null; // 这里可以恢复为 null
                /*****************************************************/
                if (pre_new != null && cur.val < pre_new.val) {
                    if (first == null) {
                        first = pre_new;
                        second = cur;
                    } else {
                        second = cur;
                    }
                }
                pre_new = cur;
                /*****************************************************/
                cur = cur.right;
            }
        }
    }

    int temp = first.val;
    first.val = second.val;
    second.val = temp;
}
```

总

自己开始看到二分查找树，还是没有想到中序遍历，而是用了递归的思路去分析。可以看到如果想到中序遍历，题目会简单很多。

100、题目描述（简单难度）

100. Same Tree

Easy 1198 39 Favorite Share

Given two binary trees, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical and the nodes have the same value.

Example 1:

```
Input:     1         1
          / \       / \
         2   3     2   3

        [1,2,3],   [1,2,3]

Output: true
```

Example 2:

```
Input:     1         1
          /           \
         2             2

        [1,2],     [1,null,2]

Output: false
```

Example 3:

```
Input:     1         1
          / \       / \
         2   1     1   2

        [1,2,1],   [1,1,2]

Output: false
```

判断两个二叉树是否相同。

解法一

这道题就很简单了，只要把两个树同时遍历一下，遍历过程中判断数值是否相等或者同时为 null 即可。而遍历的方法，当然可以选择 DFS 里的先序遍历，中序遍历，后序遍历，或者 BFS。

当然实现的话，可以用递归，用栈，或者中序遍历提到的 Morris。也可以参照 98 题 、 94 题 ，对二叉树的遍历讨论了很多。

这里的话，由于最近几题对中序遍历用的多，所以就直接用中序遍历了。

```java
public boolean isSameTree(TreeNode p, TreeNode q) {
    return inorderTraversal(p,q);
}
private boolean inorderTraversal(TreeNode p, TreeNode q) {
    if(p==null&&q==null){
        return true;
    }else if(p==null || q==null){
        return false;
    }
    //考虑左子树是否符合
    if(!inorderTraversal(p.left,q.left)){
        return false;
    }
    //考虑当前节点是否符合
    if(p.val!=q.val){
        return false;
    }
    //考虑右子树是否符合
    if(!inorderTraversal(p.right,q.right)){
        return false;
    }
    return true;
}
```

时间复杂度：O（N）。对每个节点进行了访问。

空间复杂度：O（h），h 是树的高度，也就是压栈所耗费的空间。当然 h 最小为 log（N），最大就等于 N。

最好情况例子

最差情况例子

总

这道题比较简单，本质上考察的就是二叉树的遍历。

结语

终于更新了 51-100 题，手指都麻了，麻了。

我是爱学习爱分享的沉默王二。微信搜索「**沉默王二**」可以关注我的原创公众号，回复关键字「**Java**」就可以拉取更多离线版资料下载地址。

> 再贴一下作者的在线地址！

题解预览地址:https://leetcode.wang，推荐电脑端打开，手机打开的话将页面滑到最上边，左上角是 菜单 leetcode 题目地址 https://leetcode.com/problemset/all/