53. Maximum Subarray

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.

这道题让求最大子数组之和，并且要用两种方法来解，分别是 O(n) 的解法，还有用分治法 Divide and Conquer Approach，这个解法的时间复杂度是 O(nlgn)，那就先来看 O(n) 的解法，定义两个变量 res 和 curSum，其中 res 保存最终要返回的结果，即最大的子数组之和，curSum 初始值为0，每遍历一个数字 num，比较 curSum + num 和 num 中的较大值存入 curSum，然后再把 res 和 curSum 中的较大值存入 res，以此类推直到遍历完整个数组，可得到最大子数组的值存在 res 中，代码如下：

C++ 解法一：

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int res = INT_MIN, curSum = 0;
        for (int num : nums) {
            curSum = max(curSum + num, num);
            res = max(res, curSum);
        }
        return res;
    }
};
```

Java 解法一：

```java
public class Solution {
    public int maxSubArray(int[] nums) {
        int res = Integer.MIN_VALUE, curSum = 0;
        for (int num : nums) {
            curSum = Math.max(curSum + num, num);
            res = Math.max(res, curSum);
        }
        return res;
    }
}
```

题目还要求我们用分治法 Divide and Conquer Approach 来解，这个分治法的思想就类似于二分搜索法，需要把数组一分为二，分别找出左边和右边的最大子数组之和，然后还要从中间开始向左右分别扫描，求出的最大值分别和左右两边得出的最大值相比较取最大的那一个，代码如下：

C++ 解法二:

```cpp
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        if (nums.empty()) return 0;
        return helper(nums, 0, (int)nums.size() - 1);
    }
    int helper(vector<int>& nums, int left, int right) {
        if (left >= right) return nums[left];
        int mid = left + (right - left) / 2;
        int lmax = helper(nums, left, mid - 1);
        int rmax = helper(nums, mid + 1, right);
        int mmax = nums[mid], t = mmax;
        for (int i = mid - 1; i >= left; --i) {
            t += nums[i];
            mmax = max(mmax, t);
        }
        t = mmax;
        for (int i = mid + 1; i <= right; ++i) {
            t += nums[i];
            mmax = max(mmax, t);
        }
        return max(mmax, max(lmax, rmax));
    }
};
```

Java 解法二:

```java
public class Solution {
    public int maxSubArray(int[] nums) {
        if (nums.length == 0) return 0;
        return helper(nums, 0, nums.length - 1);
```

```
    }
    public int helper(int[] nums, int left, int right) {
        if (left >= right) return nums[left];
        int mid = left + (right - left) / 2;
        int lmax = helper(nums, left, mid - 1);
        int rmax = helper(nums, mid + 1, right);
        int mmax = nums[mid], t = mmax;
        for (int i = mid - 1; i >= left; --i) {
            t += nums[i];
            mmax = Math.max(mmax, t);
        }
        t = mmax;
        for (int i = mid + 1; i <= right; ++i) {
            t += nums[i];
            mmax = Math.max(mmax, t);
        }
        return Math.max(mmax, Math.max(lmax, rmax));
    }
}
```

54. Spiral Matrix

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]
```

这道题让我们搓一个螺旋丸，将一个矩阵按照螺旋顺序打印出来，只能一条边一条边的打印，首先要从给定的 mxn 的矩阵中算出按螺旋顺序有几个环，注意最中间的环可以是一个数字，也可以是一行或者一列。环数的计算公式是 min(m, n) / 2，知道了环数，就可以对每个环的边按顺序打印，比如对于题目中给的那个例子，个边生成的顺序是(用颜色标记了数字，Github 上可能无法显示颜色，请参见博客园上的帖子) Red -> Green -> Blue -> Yellow -> Black

1 2 3

4 5 6

7 8 9

定义 p，q 为当前环的高度和宽度，当p或者q为1时，表示最后一个环只有一行或者一列，可以跳出循环。此题的难点在于下标的转换，如何正确的转换下标是解此题的关键，可以对照着上面的 3x3 的例子来完成下标的填写，代码如下：

解法一：

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int> > &matrix) {
        if (matrix.empty() || matrix[0].empty()) return {};
        int m = matrix.size(), n = matrix[0].size();
        vector<int> res;
        int c = m > n ? (n + 1) / 2 : (m + 1) / 2;
        int p = m, q = n;
        for (int i = 0; i < c; ++i, p -= 2, q -= 2) {
            for (int col = i; col < i + q; ++col)
                res.push_back(matrix[i][col]);
            for (int row = i + 1; row < i + p; ++row)
                res.push_back(matrix[row][i + q - 1]);
            if (p == 1 || q == 1) break;
            for (int col = i + q - 2; col >= i; --col)
                res.push_back(matrix[i + p - 1][col]);
            for (int row = i + p - 2; row > i; --row)
                res.push_back(matrix[row][i]);
        }
        return res;
    }
};
```

如果觉得上面解法中的下标的转换比较难弄的话，也可以使用下面这种坐标稍稍简洁一些的方法。对于这种螺旋遍历的方法，重要的是要确定上下左右四条边的位置，那么初始化的时候，上边 up 就是0，下边 down 就是 m-1，左边 left 是0，右边 right 是 n-1。然后进行 while 循环，先遍历上边，将所有元素加入结果 res，然后上边下移一位，如果此时上边大于下边，说明此时已经遍历完成了，直接 break。同理对于下边，左边，右边，依次进行相对应的操作，这样就会使得坐标很有规律，并且不易出错，参见代码如下：

解法二：

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) return {};
        int m = matrix.size(), n = matrix[0].size();
        vector<int> res;
        int up = 0, down = m - 1, left = 0, right = n - 1;
        while (true) {
            for (int j = left; j <= right; ++j) res.push_back(matrix[up][j]);
            if (++up > down) break;
            for (int i = up; i <= down; ++i) res.push_back(matrix[i][right]);
            if (--right < left) break;
            for (int j = right; j >= left; --j) res.push_back(matrix[down][j]);
            if (--down < up) break;
            for (int i = down; i >= up; --i) res.push_back(matrix[i][left]);
            if (++left > right) break;
        }
        return res;
    }
};
```

若对上面解法中的多个变量还是晕的话，也可以使用类似迷宫遍历的方法，这里只要设定正确的遍历策略，还是可以按螺旋的方式走完整个矩阵的，起点就是（0，0）位置，但是方向数组一定要注意，不能随便写，开始时是要往右走，到了边界或者访问过的位置后，就往下，然后往左，再往上，所以 dirs 数组的顺序是 右->下->左->上，由于原数组中不会有0，所以就可以将访问过的位置标记为0，这样再判断新位置的时候，只要其越界了，或者是遇到0了，就表明此时需要转弯了，到 dirs 数组中去取转向的 offset，得到新位置，注意这里的 dirs 数组中取是按循环数组的方式来操作，加1然后对4取余，按照这种类似迷宫遍历的方法也可以螺旋遍历矩阵，参见代码如下：

解法三：

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) return {};
        int m = matrix.size(), n = matrix[0].size(), idx = 0, i = 0, j = 0;
        vector<int> res;
        vector<vector<int>> dirs{{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
        for (int k = 0; k < m * n; ++k) {
            res.push_back(matrix[i][j]);
            matrix[i][j] = 0;
            int x = i + dirs[idx][0], y = j + dirs[idx][1];
            if (x < 0 || x >= m || y < 0 || y >= n || matrix[x][y] == 0) {
```

```
                idx = (idx + 1) % 4;
                x = i + dirs[idx][0];
                y = j + dirs[idx][1];
            }
            i = x;
            j = y;
        }
        return res;
    }
};
```

55. Jump Game

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.
```

这道题说的是有一个非负整数的数组，每个数字表示在当前位置的最大跳力（这里的跳力指的是在当前位置为基础上能到达的最远位置），求判断能不能到达最后一个位置，开始博主以为是必须刚好到达最后一个位置，超过了不算，其实是理解题意有误，因为每个位置上的数字表示的是最大的跳力而不是像玩大富翁一样摇骰子摇出几一定要走几。这里可以用动态规划 Dynamic Programming 来解，维护一个一维数组 dp，其中 dp[i] 表示达到i位置时剩余的跳力，若到达某个位置时跳力为负了，说明无法到达该位置。接下来难点就是推导状态转移方程啦，想想啊，到达当前位置的剩余跳力跟什么有关呢，其实是跟上一个位置的剩余跳力（dp 值）和上一个位置新的跳力（nums 数组中的值）有关，这里新的跳力就是原数组中每个位置的数字，因为其代表了以当前位置为起点能到达的最远位置。所以当前位置的剩余跳力（dp 值）和当前位置新的跳力中的较大那个数决定了当前能到的最远距离，而下一个位置的剩余跳力（dp 值）就等于当前的这个较大值减去1，因为需要花一个跳力到达下一个位置，所以就有状态转移方程了：dp[i] = max(dp[i - 1], nums[i - 1]) - 1，如果当某一个时刻 dp 数组的值为负了，说明无法抵达当前位置，则直接返回 false，最后循环结束后直接返回 true 即可，参见代码如下：

解法一：

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        vector<int> dp(nums.size(), 0);
        for (int i = 1; i < nums.size(); ++i) {
            dp[i] = max(dp[i - 1], nums[i - 1]) - 1;
            if (dp[i] < 0) return false;
        }
        return true;
    }
};
```

其实这题最好的解法不是 DP，而是贪婪算法 Greedy Algorithm，因为这里并不是很关心每一个位置上的剩余步数，而只希望知道能否到达末尾，也就是说我们只对最远能到达的位置感兴趣，所以维护一个变量 reach，表示最远能到达的位置，初始化为0。遍历数组中每一个数字，如果当前坐标大于 reach 或者 reach 已经抵达最后一个位置则跳出循环，否则就更新 reach 的值为其和 i + nums[i] 中的较大值，其中 i + nums[i] 表示当前位置能到达的最大位置，参见代码如下：

解法二：

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int n = nums.size(), reach = 0;
        for (int i = 0; i < n; ++i) {
            if (i > reach || reach >= n - 1) break;
            reach = max(reach, i + nums[i]);
        }
        return reach >= n - 1;
    }
};
```

56. Merge Intervals

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.
```

NOTE: input types have been changed on April 15, 2019. Please reset to default code definition to get new method signature.

这道和之前那道 Insert Interval 很类似，这次题目要求我们合并区间，之前那题明确了输入区间集是有序的，而这题没有，所以我们首先要做的就是给区间集排序，由于我们要排序的是个结构体，所以我们要定义自己的 comparator，才能用 sort 来排序，我们以 start 的值从小到大来排序，排完序我们就可以开始合并了，首先把第一个区间存入结果中，然后从第二个开始遍历区间集，如果结果中最后一个区间和遍历的当前区间无重叠，直接将当前区间存入结果中，如果有重叠，将结果中最后一个区间的 end 值更新为结果中最后一个区间的 end 和当前 end 值之中的较大值，然后继续遍历区间集，以此类推可以得到最终结果，代码如下：

解法一：

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        if (intervals.empty()) return {};
        sort(intervals.begin(), intervals.end());
        vector<vector<int>> res{intervals[0]};
        for (int i = 1; i < intervals.size(); ++i) {
            if (res.back()[1] < intervals[i][0]) {
                res.push_back(intervals[i]);
            } else {
                res.back()[1] = max(res.back()[1], intervals[i][1]);
            }
        }
        return res;
    }
};
```

下面这种解法将起始位置和结束位置分别存到了两个不同的数组 starts 和 ends 中，然后分别进行排序，之后用两个指针i和j，初始化时分别指向 starts 和 ends 数组的首位置，然后如果i指向 starts 数组中的最后一个位置，或者当 starts 数组上 i+1 位置上的数字大于 ends 数组的i位置上的数时，此时说明区间已经不连续了，我们来看题目中的例子，排序后的 starts 和 ends 为：

starts: 1 2 8 15

ends: 3 6 10 18

红色为i的位置，蓝色为j的位置，那么此时 starts[i+1] 为8，ends[i] 为6，8大于6，所以此时不连续了，将区间 [starts[j], ends[i]]，即 [1, 6] 加入结果 res 中，然后j赋值为 i+1 继续循环，参见代码如下：

解法二：

```
class Solution {
public:
```

```
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        int n = intervals.size();
        vector<vector<int>> res;
        vector<int> starts, ends;
        for (int i = 0; i < n; ++i) {
            starts.push_back(intervals[i][0]);
            ends.push_back(intervals[i][1]);
        }
        sort(starts.begin(), starts.end());
        sort(ends.begin(), ends.end());
        for (int i = 0, j = 0; i < n; ++i) {
            if (i == n - 1 || starts[i + 1] > ends[i]) {
                res.push_back({starts[j], ends[i]});
                j = i + 1;
            }
        }
        return res;
    }
};
```

这道题还有另一种解法，这个解法直接调用了之前那道题 Insert Interval 的函数，由于插入的过程中也有合并的操作，所以我们可以建立一个空的集合，然后把区间集的每一个区间当做一个新的区间插入结果中，也可以得到合并后的结果，那道题中的四种解法都可以在这里使用，但是没必要都列出来，这里只选了那道题中的解法二放到这里，代码如下：

解法三：

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        vector<vector<int>> res;
        for (int i = 0; i < intervals.size(); ++i) {
            res = insert(res, intervals[i]);
        }
        return res;
    }
    vector<vector<int>> insert(vector<vector<int>>& intervals, vector<int> newInterval) {
        vector<vector<int>> res;
        int n = intervals.size(), cur = 0;
        for (int i = 0; i < n; ++i) {
            if (intervals[i][1] < newInterval[0]) {
                res.push_back(intervals[i]);
                ++cur;
            } else if (intervals[i][0] > newInterval[1]) {
                res.push_back(intervals[i]);
            } else {
                newInterval[0] = min(newInterval[0], intervals[i][0]);
                newInterval[1] = max(newInterval[1], intervals[i][1]);
```

```
            }
        }
        res.insert(res.begin() + cur, newInterval);
        return res;
    }
};
```

57. Insert Interval

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].
```

NOTE: input types have been changed on April 15, 2019. Please reset to default code definition to get new method signature.

这道题让我们在一系列非重叠的区间中插入一个新的区间，可能还需要和原有的区间合并，可以对给定的区间集进行一个一个的遍历比较，那么会有两种情况，重叠或是不重叠，不重叠的情况最好，直接将新区间插入到对应的位置即可，重叠的情况比较复杂，有时候会有多个重叠，需要更新新区间的范围以便包含所有重叠，之后将新区间加入结果 res，最后将后面的区间再加入结果 res 即可。具体思路是，用一个变量 cur 来遍历区间，如果当前 cur 区间的结束位置小于要插入的区间的起始位置的话，说明没有重叠，则将 cur 区间加入结果 res 中，然后 cur 自增1。直到有 cur 越界或有重叠 while 循环退出，然后再用一个 while 循环处理所有重叠的区间，每次用取两个区间起始位置的较小值，和结束位置的较大值来更新要插入的区间，然后 cur 自增1。直到 cur 越界或者没有重叠时 while 循环退出。之后将更新好的新区间加入结果 res，然后将 cur 之后的区间再加入结果 res 中即可，参见代码如下：

解法一：

```
class Solution {
public:
    vector<vector<int>> insert(vector<vector<int>>& intervals,
vector<int>& newInterval) {
        vector<vector<int>> res;
        int n = intervals.size(), cur = 0;
```

```
        while (cur < n && intervals[cur][1] < newInterval[0]) {
            res.push_back(intervals[cur++]);
        }
        while (cur < n && intervals[cur][0] <= newInterval[1]) {
            newInterval[0] = min(newInterval[0], intervals[cur][0]);
            newInterval[1] = max(newInterval[1], intervals[cur][1]);
            ++cur;
        }
        res.push_back(newInterval);
        while (cur < n) {
            res.push_back(intervals[cur++]);
        }
        return res;
    }
};
```

下面这种方法的思路跟上面的解法很像，只不过没有用 while 循环，而是使用的是 for 循环，但是思路上没有太大的区别，变量 cur 还是用来记录新区间该插入的位置，稍有不同的地方在于在 for 循环中已经将新区间后面不重叠的区间也加进去了，for 循环结束后就只需要插入新区间即可，参见代码如下：

解法二：

```
class Solution {
public:
    vector<vector<int>> insert(vector<vector<int>>& intervals,
vector<int>& newInterval) {
        vector<vector<int>> res;
        int n = intervals.size(), cur = 0;
        for (int i = 0; i < n; ++i) {
            if (intervals[i][1] < newInterval[0]) {
                res.push_back(intervals[i]);
                ++cur;
            } else if (intervals[i][0] > newInterval[1]) {
                res.push_back(intervals[i]);
            } else {
                newInterval[0] = min(newInterval[0], intervals[i][0]);
                newInterval[1] = max(newInterval[1], intervals[i][1]);
            }
        }
        res.insert(res.begin() + cur, newInterval);
        return res;
    }
};
```

下面这种解法就是把上面解法的 for 循环改为了 while 循环，其他的都没有变，代码如下：

解法三：

```
class Solution {
```

```
public:
    vector<vector<int>> insert(vector<vector<int>>& intervals,
vector<int>& newInterval) {
        vector<vector<int>> res;
        int n = intervals.size(), cur = 0, i = 0;
        while (i < n) {
            if (intervals[i][1] < newInterval[0]) {
                res.push_back(intervals[i]);
                ++cur;
            } else if (intervals[i][0] > newInterval[1]) {
                res.push_back(intervals[i]);
            } else {
                newInterval[0] = min(newInterval[0], intervals[i][0]);
                newInterval[1] = max(newInterval[1], intervals[i][1]);
            }
            ++i;
        }
        res.insert(res.begin() + cur, newInterval);
        return res;
    }
};
```

59. Spiral Matrix II

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 ]
]
```

此题跟之前那道 Spiral Matrix 本质上没什么区别，就相当于个类似逆运算的过程，这道题是要按螺旋的顺序来填数，由于给定矩形是个正方形，我们计算环数时用 n / 2 来计算，若n为奇数时，此时最中间的那个点没有被算在环数里，所以最后需要单独赋值，还是下标转换问题是难点，参考之前 Spiral Matrix 的讲解来转换下标吧，参见代码如下：

解法一：

```
class Solution {
public:
    vector<vector<int>> generateMatrix(int n) {
```

```
        vector<vector<int>> res(n, vector<int>(n));
        int val = 1, p = n;
        for (int i = 0; i < n / 2; ++i, p -= 2) {
            for (int col = i; col < i + p; ++col)
                res[i][col] = val++;
            for (int row = i + 1; row < i + p; ++row)
                res[row][i + p - 1] = val++;
            for (int col = i + p - 2; col >= i; --col)
                res[i + p - 1][col] = val++;
            for (int row = i + p - 2; row > i; --row)
                res[row][i] = val++;
        }
        if (n % 2 != 0) res[n / 2][n / 2] = val;
        return res;
    }
};
```

当然我们也可以使用下面这种简化了坐标转换的方法，博主个人还是比较推崇下面这种解法，不容易出错，而且好理解，参见代码如下：

解法二：

```
class Solution {
public:
    vector<vector<int>> generateMatrix(int n) {
        vector<vector<int>> res(n, vector<int>(n));
        int up = 0, down = n - 1, left = 0, right = n - 1, val = 1;
        while (true) {
            for (int j = left; j <= right; ++j) res[up][j] = val++;
            if (++up > down) break;
            for (int i = up; i <= down; ++i) res[i][right] = val++;
            if (--right < left) break;
            for (int j = right; j >= left; --j) res[down][j] = val++;
            if (--down < up) break;
            for (int i = down; i >= up; --i) res[i][left] = val++;
            if (++left > right) break;
        }
        return res;
    }
};
```

60. Permutation Sequence

The set `[1,2,3,..., _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"
```

这道题是让求出n个数字的第k个排列组合，由于其特殊性，我们不用将所有的排列组合的情况都求出来，然后返回其第k个，我们可以只求出第k个排列组合即可，那么难点就在于如何知道数字的排列顺序，可参见网友喜刷刷的博客，首先我们要知道当n = 3时，其排列组合共有3! = 6种，当n = 4时，其排列组合共有4! = 24种，我们就以n = 4, k = 17的情况来分析，所有排列组合情况如下：

1234
1243
1324
1342
1423
1432
2134
2143
2314
2341
2413
2431
3124
3142
3214
3241
3412 <--- k = 17
3421
4123
4132
4213

4231
4312
4321

我们可以发现，每一位上1,2,3,4分别都出现了6次，当最高位上的数字确定了，第二高位每个数字都出现了2次，当第二高位也确定了，第三高位上的数字都只出现了1次，当第三高位确定了，那么第四高位上的数字也只能出现一次，下面我们来看k = 17这种情况的每位数字如何确定，由于k = 17是转化为数组下标为16：

最高位可取1,2,3,4中的一个，每个数字出现3！= 6次（因为当最高位确定了，后面三位可以任意排列，所以是3！，那么最高位的数字就会重复3！次），所以k = 16的第一位数字的下标为16 / 6 = 2，在 "1234" 中即3被取出。这里我们的k是要求的坐标为k的全排列序列，我们定义 k' 为当最高位确定后，要求的全排序列在新范围中的位置，同理，k'' 为当第二高为确定后，所要求的全排列序列在新范围中的位置，以此类推，下面来具体看看：

第二位此时从1,2,4中取一个，k = 16，则此时的 k' = 16 % (3!) = 4，注意思考这里为何要取余，如果对这24个数以6个一组来分，那么k=16这个位置就是在第三组（k/6 = 2）中的第五个（k%6 = 4）数字。如下所示，而剩下的每个数字出现2！= 2次，所以第二数字的下标为4 / 2 = 2，在 "124" 中即4被取出。

3124
3142
3214
3241
3412 <--- k' = 4
3421

第三位此时从1,2中去一个，k' = 4，则此时的k'' = 4 % (2!) = 0，如下所示，而剩下的每个数字出现1！= 1次，所以第三个数字的下标为 0 / 1 = 0，在 "12" 中即1被取出。

3412 <--- k'' = 0
3421

第四位是从2中取一个，k'' = 0，则此时的k''' = 0 % (1!) = 0，如下所示，而剩下的每个数字出现0！= 1次，所以第四个数字的下标为0 / 1= 0，在 "2" 中即2被取出。

3412 <--- k''' = 0

那么我们就可以找出规律了
a1 = k / (n - 1)!
k1 = k

a2 = k1 / (n - 2)!
k2 = k1 % (n - 2)!
...

an-1 = kn-2 / 1!
kn-1 = kn-2 % 1!

an = kn-1 / 0!
kn = kn-1 % 0!

代码如下：

```
class Solution {
public:
    string getPermutation(int n, int k) {
        string res;
        string num = "123456789";
        vector<int> f(n, 1);
        for (int i = 1; i < n; ++i) f[i] = f[i - 1] * i;
        --k;
        for (int i = n; i >= 1; --i) {
            int j = k / f[i - 1];
            k %= f[i - 1];
            res.push_back(num[j]);
            num.erase(j, 1);
        }
        return res;
    }
};
```

61. Rotate List

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

For example:
Given `1->2->3->4->5->NULL` and *k* = `2`,
return `4->5->1->2->3->NULL`.

这道旋转链表的题和之前那道 Rotate Array 旋转数组 很类似，但是比那道要难一些，因为链表的值不能通过下表来访问，只能一个一个的走，我刚开始拿到这题首先想到的就是用快慢指针来解，快指针先走k步，然后两个指针一起走，当快指针走到末尾时，慢指针的下一个位置是新的顺序的头结点，这样就可以旋转链表了，自信满满的写完程序，放到OJ上跑，以为能一次通过，结果跪在了各种特殊情况，首先一个就是当原链表为空时，直接返回NULL，还有就是当k大于链表长度和k远远大于链表长度时该如何处理，我们需要首先遍历一遍原链表得到链表长度n，然后k对n取余，这样k肯定小于n，就可以用上面的算法了，代码如下:

解法一

```
class Solution {
public:
    ListNode *rotateRight(ListNode *head, int k) {
        if (!head) return NULL;
        int n = 0;
        ListNode *cur = head;
        while (cur) {
            ++n;
            cur = cur->next;
        }
        k %= n;
```

```
        ListNode *fast = head, *slow = head;
        for (int i = 0; i < k; ++i) {
            if (fast) fast = fast->next;
        }
        if (!fast) return head;
        while (fast->next) {
            fast = fast->next;
            slow = slow->next;
        }
        fast->next = head;
        fast = slow->next;
        slow->next = NULL;
        return fast;
    }
};
```

这道题还有一种解法，跟上面的方法类似，但是不用快慢指针，一个指针就够了，原理是先遍历整个链表获得链表长度n，然后此时把链表头和尾链接起来，在往后走n - k % n个节点就到达新链表的头结点前一个点，这时断开链表即可，代码如下:

```
class Solution {
public:
    ListNode *rotateRight(ListNode *head, int k) {
        if (!head) return NULL;
        int n = 1;
        ListNode *cur = head;
        while (cur->next) {
            ++n;
            cur = cur->next;
        }
        cur->next = head;
        int m = n - k % n;
        for (int i = 0; i < m; ++i) {
            cur = cur->next;
        }
        ListNode *newhead = cur->next;
        cur->next = NULL;
        return newhead;
    }
};
```

62. Unique Paths

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
```

这道题让求所有不同的路径的个数，一开始还真把博主难住了，因为之前好像没有遇到过这类的问题，所以感觉好像有种无从下手的感觉。在网上找攻略之后才恍然大悟，原来这跟之前那道 Climbing Stairs 很类似，那道题是说可以每次能爬一格或两格，问到达顶部的所有不同爬法的个数。而这道题是每次可以向下走或者向右走，求到达最右下角的所有不同走法的个数。那么跟爬梯子问题一样，需要用动态规划 Dynamic Programming 来解，可以维护一个二维数组 dp，其中 dp[i][j] 表示到当前位置不同的走法的个数，然后可以得到状态转移方程为: dp[i][j] = dp[i - 1][j] + dp[i][j - 1]，这里为了节省空间，使用一维数组 dp，一行一行的刷新也可以，代码如下：

解法一：

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<int> dp(n, 1);
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                dp[j] += dp[j - 1];
            }
        }
        return dp[n - 1];
    }
};
```

这道题其实还有另一种很数学的解法，参见网友 Code Ganker 的博客，实际相当于机器人总共走了 m + n - 2步，其中 m - 1 步向右走，n - 1 步向下走，那么总共不同的方法个数就相当于在步数里面 m - 1 和 n - 1 中较小的那个数的取法，实际上是一道组合数的问题，写出代码如下:

解法二：

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        double num = 1, denom = 1;
        int small = m > n ? n : m;
        for (int i = 1; i <= small - 1; ++i) {
            num *= m + n - 1 - i;
            denom *= i;
        }
        return (int)(num / denom);
    }
};
```

63. Unique Paths II

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
```

这道题是之前那道 Unique Paths 的延伸，在路径中加了一些障碍物，还是用动态规划 Dynamic Programming 来解，使用一个二维的 dp 数组，大小为 (m+1) x (n+1)，这里的 dp[i][j] 表示到达 (i-1, j-1) 位置的不同路径的数量，那么i和j需要更新的范围就是 [1, m] 和 [1, n]。状态转移方程跟之前那道题是一样的，因为每个位置只能由其上面和左面的位置移动而来，所以也是由其上面和左边的 dp 值相加来更新当前的 dp 值，如下所示：

dp[i][j] = dp[i-1][j] + dp[i][j-1]

这里就能看出来初始化 d p数组的大小为 (m+1) x (n+1)，是为了 handle 边缘情况，当i或j为0时，减1可能会出错。当某个位置是障碍物时，其 dp 值为0，直接跳过该位置即可。这里还需要初始化 dp 数组的某个值，使得其能正常累加。当起点不是障碍物时，其 dp 值应该为1，即dp[1][1] = 1，由于其是由 dp[0][1] + dp[1][0] 更新而来，所以二者中任意一个初始化为1即可。由于之后 LeetCode 更新了这道题的 test case，使得使用 int 型的 dp 数组会有溢出的错误，所以改为使用 long 型的数组来避免 overflow，代码如下：

解法一：

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        if (obstacleGrid.empty() || obstacleGrid[0].empty() || obstacleGrid[0][0] == 1) return 0;
        int m = obstacleGrid.size(), n = obstacleGrid[0].size();
        vector<vector<long>> dp(m + 1, vector<long>(n + 1, 0));
        dp[0][1] = 1;
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (obstacleGrid[i - 1][j - 1] != 0) continue;
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }
        return dp[m][n];
    }
};
```

或者我们也可以使用一维 dp 数组来解，省一些空间，参见代码如下：

解法二：

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        if (obstacleGrid.empty() || obstacleGrid[0].empty() || obstacleGrid[0][0] == 1) return 0;
        int m = obstacleGrid.size(), n = obstacleGrid[0].size();
        vector<long> dp(n, 0);
        dp[0] = 1;
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (obstacleGrid[i][j] == 1) dp[j] = 0;
```

```
                else if (j > 0) dp[j] += dp[j - 1];
            }
        }
        return dp[n - 1];
    }
};
```

64. Minimum Path Sum

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.
```

这道题给了我们一个只有非负数的二维数组，让找一条从左上到右下的路径，使得路径和最小，限定了每次只能向下或者向右移动。一个常见的错误解法就是每次走右边或下边数字中较小的那个，这样的贪婪算法获得的局部最优解不一定是全局最优解，因此是不行的。实际上这道题跟之前那道 Dungeon Game 没有什么太大的区别，都需要用动态规划 Dynamic Programming 来做，这应该算是 DP 问题中比较简单的一类，我们维护一个二维的 dp 数组，其中 dp[i][j] 表示到达当前位置的最小路径和。接下来找状态转移方程，因为到达当前位置 (i, j) 只有两种情况，要么从上方 (i-1, j) 过来，要么从左边 (i, j-1) 过来，我们选择 dp 值较小的那个路径，即比较 dp[i-1][j] 和 dp[i][j-1]，将其中的较小值加上当前的数字 grid[i][j]，就是当前位置的 dp 值了。但是有些特殊情况要提前赋值，比如起点位置，直接赋值为 grid[0][0]，还有就是第一行和第一列，其中第一行的位置只能从左边过来，第一列的位置从能从上面过来，所以这两行要提前初始化好，然后再从 (1, 1) 的位置开始更新到右下角即可，反正难度不算大，代码如下：

解法一：

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        if (grid.empty() || grid[0].empty()) return 0;
        int m = grid.size(), n = grid[0].size();
        vector<vector<int>> dp(m, vector<int>(n));
        dp[0][0] = grid[0][0];
        for (int i = 1; i < m; ++i) dp[i][0] = grid[i][0] + dp[i - 1][0];
```

```
        for (int j = 1; j < n; ++j) dp[0][j] = grid[0][j] + dp[0][j - 1];
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                dp[i][j] = grid[i][j] + min(dp[i - 1][j], dp[i][j - 1]);
            }
        }
        return dp[m - 1][n - 1];
    }
};
```

我们可以优化空间复杂度，可以使用一个一维的 dp 数组就可以了，初始化为整型最大值，但是 dp[0][0] 要初始化为0。之所以可以用一维数组代替之前的二维数组，是因为当前的 dp 值只跟左边和上面的 dp 值有关。这里我们并不提前更新第一行或是第一列，而是在遍历的时候判断，若j等于0时，说明是第一列，我们直接加上当前的数字，否则就要比较是左边的 dp[j-1] 小还是上面的 dp[j] 小，当是第一行的时候，dp[j] 是整型最大值，所以肯定会取到 dp[j-1] 的值，然后再加上当前位置的数字即可，参见代码如下：

解法二：

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        if (grid.empty() || grid[0].empty()) return 0;
        int m = grid.size(), n = grid[0].size();
        vector<int> dp(n, INT_MAX);
        dp[0] = 0;
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (j == 0) dp[j] += grid[i][j];
                else dp[j] = grid[i][j] + min(dp[j], dp[j - 1]);
            }
        }
        return dp[n - 1];
    }
};
```

我们还可以进一步的优化空间，连一维数组都不用新建，而是直接使用原数组 grid 进行累加，这里的累加方式跟解法一稍有不同，没有提前对第一行和第一列进行赋值，而是放在一起判断了，当i和j同时为0时，直接跳过。否则当i等于0时，只加上左边的值，当j等于0时，只加上面的值，否则就比较左边和上面的值，加上较小的那个即可，参见代码如下：

解法三：

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        if (grid.empty() || grid[0].empty()) return 0;
        for (int i = 0; i < grid.size(); ++i) {
            for (int j = 0; j < grid[i].size(); ++j) {
```

```
                if (i == 0 && j == 0) continue;
                if (i == 0) grid[0][j] += grid[0][j - 1];
                else if (j == 0) grid[i][0] += grid[i - 1][0];
                else grid[i][j] += min(grid[i - 1][j], grid[i][j - 1]);
            }
        }
        return grid.back().back();
    }
};
```

下面这种写法跟上面的基本相同，只不过用了 up 和 left 两个变量来计算上面和左边的值，看起来稍稍简洁一点，参见代码如下：

解法四：

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        if (grid.empty() || grid[0].empty()) return 0;
        for (int i = 0; i < grid.size(); ++i) {
            for (int j = 0; j < grid[i].size(); ++j) {
                if (i == 0 && j == 0) continue;
                int up = (i == 0) ? INT_MAX : grid[i - 1][j];
                int left = (j == 0) ? INT_MAX : grid[i][j - 1];
                grid[i][j] += min(up, left);
            }
        }
        return grid.back().back();
    }
};
```

65. Valid Number

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.

Update (2015-02-10):
The signature of the `C++` function had been updated. If you still see your function signature accepts a `const char *` argument, please click the reload button to reset your code definition.

这道验证数字的题比想象中的要复杂的多，有很多情况需要考虑，而OJ上给这道题的分类居然是Easy，Why? 而10.9% 的全场最低的Accept Rate正说明这道题的难度，网上有很多解法，有利用有限自动机Finite Automata Machine的程序写的简洁优雅 (http://blog.csdn.net/kenden23/article/details/18696083), 还有利用正则表达式，更是写的丧心病狂的简洁 (http://blog.csdn.net/fightforyourdream/article/details/12900751)。而我主要还是用最一般的写法，参考了网上另一篇博文 (http://yucoding.blogspot.com/2013/05/leetcode-question-118-valid-number.html)，处理各种情况。

首先，从题目中给的一些例子可以分析出来，我们所需要关注的除了数字以外的特殊字符有空格 ' '， 小数点 '.', 自然数 'e/E', 还要加上正负号 '+/-"， 除了这些字符需要考虑意外，出现了任何其他的字符，可以马上判定不是数字。下面我们来一一分析这些出现了也可能是数字的特殊字符：

\1. 空格 ' '： 空格分为两种情况需要考虑，一种是出现在开头和末尾的空格，一种是出现在中间的字符。出现在开头和末尾的空格不影响数字，而一旦中间出现了空格，则立马不是数字。解决方法：预处理时去掉字符的首位空格，中间再检测到空格，则判定不是数字。

\2. 小数点 '.'：小数点需要分的情况较多，首先的是小数点只能出现一次，但是小数点可以出现在任何位置，开头(".3"), 中间("1.e2"), 以及结尾("1."), 而且需要注意的是，小数点不能出现在自然数 'e/E' 之后，如 "1e.1" false, "1e1.1" false。还有，当小数点位于末尾时，前面必须是数字，如 "1." true，" -." false。解决方法：开头中间结尾三个位置分开讨论情况。

\3. 自然数 'e/E'：自然数的前后必须有数字，即自然数不能出现在开头和结尾，如 "e" false，".e1" false, "3.e" false, "3.e1" true。而且小数点只能出现在自然数之前，还有就是自然数前面不能是符号，如 "+e1" false, "1+e" false. 解决方法：开头中间结尾三个位置分开讨论情况。

\4. 正负号 '+/-"，正负号可以再开头出现，可以再自然数e之后出现，但不能是最后一个字符，后面得有数字，如 "+1.e+5" true。解决方法：开头中间结尾三个位置分开讨论情况。

下面我们开始正式分开头中间结尾三个位置来讨论情况：

\1. 在讨论三个位置之前做预处理，去掉字符串首尾的空格，可以采用两个指针分别指向开头和结尾，遇到空格则跳过，分别指向开头结尾非空格的字符。

\2. 对首字符处理，首字符只能为数字或者正负号 '+/-"，我们需要定义三个flag在标示我们是否之前检测到过小数点，自然数和正负号。首字符如为数字或正负号，则标记对应的flag，若不是，直接返回false。

\3. 对中间字符的处理，中间字符会出现五种情况，数字，小数点，自然数，正负号和其他字符。

若是数字，标记flag并通过。

若是自然数，则必须是第一次出现自然数，并且前一个字符不能是正负号，而且之前一定要出现过数字，才能标记flag通过。

若是正负号，则之前的字符必须是自然数e，才能标记flag通过。

若是小数点，则必须是第一次出现小数点并且自然数没有出现过，才能标记flag通过。

若是其他，返回false。

\4. 对尾字符处理，最后一个字符只能是数字或小数点，其他字符都返回false。

若是数字，返回true。

若是小数点，则必须是第一次出现小数点并且自然数没有出现过，还有前面必须是数字，才能返回true。

解法一：

```
class Solution {
public:
    bool isNumber(string s) {
        int len = s.size();
        int left = 0, right = len - 1;
        bool eExisted = false;
        bool dotExisted = false;
        bool digitExisited = false;
        // Delete spaces in the front and end of string
        while (s[left] == ' ') ++left;
        while (s[right] == ' ') --right;
        // If only have one char and not digit, return false
        if (left >= right && (s[left] < '0' || s[left] > '9')) return
false;
        //Process the first char
        if (s[left] == '.') dotExisted = true;
        else if (s[left] >= '0' && s[left] <= '9') digitExisited = true;
        else if (s[left] != '+' && s[left] != '-') return false;
        // Process the middle chars
        for (int i = left + 1; i <= right - 1; ++i) {
            if (s[i] >= '0' && s[i] <= '9') digitExisited = true;
            else if (s[i] == 'e' || s[i] == 'E') { // e/E cannot follow
+/-, must follow a digit
```

```
                if (!eExisted && s[i - 1] != '+' && s[i - 1] != '-' &&
digitExisited) eExisted = true;
                else return false;
            } else if (s[i] == '+' || s[i] == '-') { // +/- can only
follow e/E
                if (s[i - 1] != 'e' && s[i - 1] != 'E') return false;

            } else if (s[i] == '.') { // dot can only occur once and
cannot occur after e/E
                if (!dotExisted && !eExisted) dotExisted = true;
                else return false;
            } else return false;
        }
        // Process the last char, it can only be digit or dot, when is
dot, there should be no dot and e/E before and must follow a digit
        if (s[right] >= '0' && s[right] <= '9') return true;
        else if (s[right] == '.' && !dotExisted && !eExisted &&
digitExisited) return true;
        else return false;
    }
};
```

上面的写法略为复杂，我们尝试着来优化一下，根据上面的分析，所有的字符可以分为六大类，空格，符号，数字，小数点，自然底数和其他字符，我们需要五个标志变量，num, dot, exp, sign分别表示数字，小数点，自然底数和符号是否出现，numAfterE表示自然底数后面是否有数字，那么我们分别来看各种情况：

- 空格： 我们需要排除的情况是，当前位置是空格而后面一位不为空格，但是之前有数字，小数点，自然底数或者符号出现时返回false。

- 符号：符号前面如果有字符的话必须是空格或者是自然底数，标记sign为true。

- 数字：标记num和numAfterE为true。

- 小数点：如果之前出现过小数点或者自然底数，返回false，否则标记dot为true。

- 自然底数：如果之前出现过自然底数或者之前从未出现过数字，返回false，否则标记exp为true，numAfterE为false。

- 其他字符：返回false。

最后返回num && numAfterE即可。

解法二：

```
class Solution {
public:
    bool isNumber(string s) {
        bool num = false, numAfterE = true, dot = false, exp = false, sign
= false;
        int n = s.size();
        for (int i = 0; i < n; ++i) {
```

```
            if (s[i] == ' ') {
                if (i < n - 1 && s[i + 1] != ' ' && (num || dot || exp || sign)) return false;
            } else if (s[i] == '+' || s[i] == '-') {
                if (i > 0 && s[i - 1] != 'e' && s[i - 1] != ' ') return false;
                sign = true;
            } else if (s[i] >= '0' && s[i] <= '9') {
                num = true;
                numAfterE = true;
            } else if (s[i] == '.') {
                if (dot || exp) return false;
                dot = true;
            } else if (s[i] == 'e') {
                if (exp || !num) return false;
                exp = true;
                numAfterE = false;
            } else return false;
        }
        return num && numAfterE;
    }
};
```

这道题给了例子不够用，下面这些例子都是我在调试的过程中出现过的例子，用来参考：

```
string s1 = "0"; // True
string s2 = " 0.1 "; // True
string s3 = "abc"; // False
string s4 = "1 a"; // False
string s5 = "2e10"; // True

string s6 = "-e10"; // False
string s7 = " 2e-9 "; // True
string s8 = "+e1"; // False
string s9 = "1+e"; // False
string s10 = " "; // False

string s11 = "e9"; // False
string s12 = "4e+"; // False
string s13 = " -."; // False
string s14 = "+.8"; // True
string s15 = " 005047e+6"; // True

string s16 = ".e1"; // False
string s17 = "3.e"; // False
string s18 = "3.e1"; // True
string s19 = "+1.e+5"; // True
string s20 = " -54.53061"; // True

```

```
25 string s21 = ". 1"; // False
```

68. Text Justification

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 extraspace 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                  "
]
```

我将这道题翻译为文本的左右对齐是因为这道题像极了word软件里面的文本左右对齐功能，这道题我前前后后折腾了快四个小时终于通过了OJ，完成了之后想着去网上搜搜看有没有更简单的方法，搜了一圈发现都差不多，都挺复杂的，于是乎就按自己的思路来说吧，由于返回的结果是多行的，所以我们在处理的时候也要一行一行的来处理，首先要做的就是确定每一行能放下的单词数，这个不难，就是比较n个单词的长度和加上n - 1个空格的长度跟给定的长度L来比较即可，找到了一行能放下的单词个数，然后计算出这一行存在的空格的个数，是用给定的长度L减去这一行所有单词的长度和。得到了空格的个数之后，就要在每个单词后面插入这些空格，这里有两种情况，比如某一行有两个单词"to" 和 "a"，给定长度L为6，如果这行不是最后一行，那么应该输出"to a"，如果是最后一行，则应该输出 "to a "，所以这里需要分情况讨论，最后一行的处理方法和其他行之间略有不同。最后一个难点就是，如果一行有三个单词，这时候中间有两个空，如果空格数不是2的倍数，那么左边的空间里要比右边的空间里多加入一个空格，那么我们只需要用总的空格数除以空间个数，能除尽最好，说明能平均分配，除不尽的话就多加个空格放在左边的空间里，以此类推，具体实现过程还是看代码吧：

```
class Solution {
```

```
public:
    vector<string> fullJustify(vector<string> &words, int L) {
        vector<string> res;
        int i = 0;
        while (i < words.size()) {
            int j = i, len = 0;
            while (j < words.size() && len + words[j].size() + j - i <= L)
{
                len += words[j++].size();
            }
            string out;
            int space = L - len;
            for (int k = i; k < j; ++k) {
                out += words[k];
                if (space > 0) {
                    int tmp;
                    if (j == words.size()) {
                        if (j - k == 1) tmp = space;
                        else tmp = 1;
                    } else {
                        if (j - k - 1 > 0) {
                            if (space % (j - k - 1) == 0) tmp = space / (j
- k - 1);
                            else tmp = space / (j - k - 1) + 1;
                        } else tmp = space;
                    }
                    out.append(tmp, ' ');
                    space -= tmp;
                }
            }
            res.push_back(out);
            i = j;
        }
        return res;
    }
};
```

69. Sqrt(x)

Implement `int sqrt(int x)`.

Compute and return the square root of *x*.

这道题要求我们求平方根，我们能想到的方法就是算一个候选值的平方，然后和x比较大小，为了缩短查找时间，我们采用二分搜索法来找平方根，这里属于博主之前总结的LeetCode Binary Search Summary 二分搜索法小结中的第三类的变形，找最后一个不大于目标值的数，代码如下：

解法一：

```
class Solution {
public:
    int mySqrt(int x) {
        if (x <= 1) return x;
        int left = 0, right = x;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (x / mid >= mid) left = mid + 1;
            else right = mid;
        }
        return right - 1;
    }
};
```

这道题还有另一种解法，是利用牛顿迭代法，记得高数中好像讲到过这个方法，是用逼近法求方程根的神器，在这里也可以借用一下，可参见网友Annie Kim's Blog的博客，因为要求x2 = n的解，令f(x)=x2-n，相当于求解f(x)=0的解，可以求出递推式如下：

xi+1=xi - (xi2 - n) / (2xi) = xi - xi / 2 + n / (2xi) = xi / 2 + n / 2xi = (xi + n/xi) / 2

解法二：

```
class Solution {
public:
    int mySqrt(int x) {
        if (x == 0) return 0;
        double res = 1, pre = 0;
        while (abs(res - pre) > 1e-6) {
            pre = res;
            res = (res + x / res) / 2;
        }
        return int(res);
    }
};
```

也是牛顿迭代法，写法更加简洁一些，注意为了防止越界，声明为长整型，参见代码如下：

解法三：

```
class Solution {
public:
    int mySqrt(int x) {
        long res = x;
        while (res * res > x) {
            res = (res + x / res) / 2;
        }
        return res;
    }
};
```

71. Simplify Path

Given an absolute path for a file (Unix-style), simplify it.

For example,
path = `"/home/"`, => `"/home"`
path = `"/a/./b/../../c/"`, => `"/c"`

click to show corner cases.

Corner Cases:

- Did you consider the case where path = `"/../"`?
  In this case, you should return `"/"`.
- Another corner case is the path might contain multiple slashes `'/'` together, such as `"/home//foo/"`.
  In this case, you should ignore redundant slashes and return `"/home/foo"`.

这道题让简化给定的路径，光根据题目中给的那一个例子还真不太好总结出规律，应该再加上两个例子 path = `"/a/./b/../c/"`, => `"/a/c"`和path = "/a/./b/c/", => "/a/b/c", 这样我们就可以知道中间是"."的情况直接去掉，是".."时删掉它上面挨着的一个路径，而下面的边界条件给的一些情况中可以得知，如果是空的话返回"/"，如果有多个"/"只保留一个。那么我们可以把路径看做是由一个或多个"/"分割开的众多子字符串，把它们分别提取出来一一处理即可，代码如下：

C++ 解法一：

```
class Solution {
public:
    string simplifyPath(string path) {
        vector<string> v;
        int i = 0;
        while (i < path.size()) {
            while (path[i] == '/' && i < path.size()) ++i;
            if (i == path.size()) break;
            int start = i;
            while (path[i] != '/' && i < path.size()) ++i;
            int end = i - 1;
            string s = path.substr(start, end - start + 1);
            if (s == "..") {
                if (!v.empty()) v.pop_back();
            } else if (s != ".") {
                v.push_back(s);
            }
        }
        if (v.empty()) return "/";
        string res;
        for (int i = 0; i < v.size(); ++i) {
            res += '/' + v[i];
        }
```

```
        return res;
    }
};
```

还有一种解法是利用了C语言中的函数strtok来分隔字符串，但是需要把string和char*类型相互转换，转换方法请猛戳这里。除了这块不同，其余的思想和上面那种解法相同，代码如下：

C 解法一：

```
class Solution {
public:
    string simplifyPath(string path) {
        vector<string> v;
        char *cstr = new char[path.length() + 1];
        strcpy(cstr, path.c_str());
        char *pch = strtok(cstr, "/");
        while (pch != NULL) {
            string p = string(pch);
            if (p == "..") {
                if (!v.empty()) v.pop_back();
            } else if (p != ".") {
                v.push_back(p);
            }
            pch = strtok(NULL, "/");
        }
        if (v.empty()) return "/";
        string res;
        for (int i = 0; i < v.size(); ++i) {
            res += '/' + v[i];
        }
        return res;
    }
};
```

C++中也有专门处理字符串的机制，我们可以使用stringstream来分隔字符串，然后对每一段分别处理，思路和上面的方法相似，参见代码如下：

C++ 解法二：

```
class Solution {
public:
    string simplifyPath(string path) {
        string res, t;
        stringstream ss(path);
        vector<string> v;
        while (getline(ss, t, '/')) {
            if (t == "" || t == ".") continue;
            if (t == ".." && !v.empty()) v.pop_back();
            else if (t != "..") v.push_back(t);
```

```
        }
        for (string s : v) res += "/" + s;
        return res.empty() ? "/" : res;
    }
};
```

Java 解法二：

```
public class Solution {
    public String simplifyPath(String path) {
        Stack<String> s = new Stack<>();
        String[] p = path.split("/");
        for (String t : p) {
            if (!s.isEmpty() && t.equals("..")) {
                s.pop();
            } else if (!t.equals(".") && !t.equals("") && !t.equals("..")) {
                s.push(t);
            }
        }
        List<String> list = new ArrayList(s);
        return "/" + String.join("/", list);
    }
}
```

72. Edit Distance

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')
```

这道题让求从一个字符串转变到另一个字符串需要的变换步骤，共有三种变换方式，插入一个字符，删除一个字符，和替换一个字符。题目乍眼一看并不难，但是实际上却暗藏玄机，对于两个字符串的比较，一般都会考虑一下用 HashMap 统计字符出现的频率，但是在这道题却不可以这么做，因为字符串的顺序很重要。还有一种比较常见的错误，就是想当然的认为对于长度不同的两个字符串，长度的差值都是要用插入操作，然后再对应每位字符，不同的地方用修改操作，但是其实这样可能会多用操作，因为删除操作有时同时可以达到修改的效果。比如题目中的例子1，当把 horse 变为 rorse 之后，之后只要删除第二个r，跟最后一个e，就可以变为 ros。实际上只要三步就完成了，因为删除了某个字母后，原来左右不相连的字母现在就连一起了，有可能刚好组成了需要的字符串。所以在比较的时候，要尝试三种操作，因为谁也不知道当前的操作会对后面产生什么样的影响。对于当前比较的两个字符 word1[i] 和 word2[j]，若二者相同，一切好说，直接跳到下一个位置。若不相同，有三种处理方法，首先是直接插入一个 word2[j]，那么 word2[j] 位置的字符就跳过了，接着比较 word1[i] 和 word2[j+1] 即可。第二个种方法是删除，即将 word1[i] 字符直接删掉，接着比较 word1[i+1] 和 word2[j] 即可。第三种则是将 word1[i] 修改为 word2[j]，接着比较 word1[i+1] 和 word[j+1] 即可。分析到这里，就可以直接写出递归的代码，但是很可惜会 Time Limited Exceed，所以必须要优化时间复杂度，需要去掉大量的重复计算，这里使用记忆数组 memo 来保存计算过的状态，从而可以通过 OJ，注意这里的 insertCnt，deleteCnt，replaceCnt 仅仅是表示当前对应的位置分别采用了插入，删除，和替换操作，整体返回的最小距离，后面位置的还是会调用递归返回最小的，参见代码如下：

解法一：

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.size(), n = word2.size();
        vector<vector<int>> memo(m, vector<int>(n));
        return helper(word1, 0, word2, 0, memo);
    }
    int helper(string& word1, int i, string& word2, int j,
vector<vector<int>>& memo) {
        if (i == word1.size()) return (int)word2.size() - j;
        if (j == word2.size()) return (int)word1.size() - i;
        if (memo[i][j] > 0) return memo[i][j];
        int res = 0;
        if (word1[i] == word2[j]) {
            return helper(word1, i + 1, word2, j + 1, memo);
        } else {
            int insertCnt = helper(word1, i, word2, j + 1, memo);
```

```
            int deleteCnt = helper(word1, i + 1, word2, j, memo);
            int replaceCnt = helper(word1, i + 1, word2, j + 1, memo);
            res = min(insertCnt, min(deleteCnt, replaceCnt)) + 1;
        }
        return memo[i][j] = res;
    }
};
```

根据以往的经验，对于字符串相关的题目且求极值的问题，十有八九都是用动态规划 Dynamic Programming 来解，这道题也不例外。其实解法一的递归加记忆数组的方法也可以看作是 DP 的递归写法。这里需要维护一个二维的数组 dp，其大小为 mxn，m和n分别为 word1 和 word2 的长度。dp[i][j] 表示从 word1 的前i个字符转换到 word2 的前j个字符所需要的步骤。先给这个二维数组 dp 的第一行第一列赋值，这个很简单，因为第一行和第一列对应的总有一个字符串是空串，于是转换步骤完全是另一个字符串的长度。跟以往的 DP 题目类似，难点还是在于找出状态转移方程，可以举个例子来看，比如 word1 是 "bbc"，word2 是 "abcd"，可以得到 dp 数组如下：

```
  Ø a b c d
Ø 0 1 2 3 4
b 1 1 1 2 3
b 2 2 1 2 3
c 3 3 2 1 2
```

通过观察可以发现，当 word1[i] == word2[j] 时，dp[i][j] = dp[i - 1][j - 1]，其他情况时，dp[i][j] 是其左，左上，上的三个值中的最小值加1，其实这里的左，上，和左上，分别对应的增加，删除，修改操作，具体可以参见解法一种的讲解部分，那么可以得到状态转移方程为：

dp[i][j] = dp[i - 1][j - 1] if word1[i - 1] == word2[j - 1]

= min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1])) + 1 else

解法二：

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.size(), n = word2.size();
        vector<vector<int>> dp(m + 1, vector<int>(n + 1));
        for (int i = 0; i <= m; ++i) dp[i][0] = i;
        for (int i = 0; i <= n; ++i) dp[0][i] = i;
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (word1[i - 1] == word2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    dp[i][j] = min(dp[i - 1][j - 1], min(dp[i - 1][j],
dp[i][j - 1])) + 1;
                }
            }
        }
```

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

73. Set Matrix Zeroes

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

click to show follow up.

Follow up:

Did you use extra space?
A straight forward solution using O(*m n*) space is probably a bad idea.
A simple improvement uses O(*m* + *n*) space, but still not the best solution.
Could you devise a constant space solution?

据说这题是CareerCup上的原题，我还没有刷CareerCup，所以不知道啦，不过这题也不算难，虽然我也是看了网上的解法照着写的，但是下次遇到绝对想的起来。这道题中说的空间复杂度为O(mn)的解法自不用多说，直接新建一个和matrix等大小的矩阵，然后一行一行的扫，只要有0，就将新建的矩阵的对应行全赋0，行扫完再扫列，然后把更新完的矩阵赋给matrix即可，这个算法的空间复杂度太高。将其优化到O(m+n)的方法是，用一个长度为m的一维数组记录各行中是否有0，用一个长度为n的一维数组记录各列中是否有0，最后直接更新matrix数组即可。这道题的要求是用O(1)的空间，那么我们就不能新建数组，我们考虑就用原数组的第一行第一列来记录各行各列是否有0.

- 先扫描第一行第一列，如果有0，则将各自的flag设置为true
- 然后扫描除去第一行第一列的整个数组，如果有0，则将对应的第一行和第一列的数字赋0
- 再次遍历除去第一行第一列的整个数组，如果对应的第一行和第一列的数字有一个为0，则将当前值赋0
- 最后根据第一行第一列的flag来更新第一行第一列

代码如下：

```
class Solution {
public:
    void setZeroes(vector<vector<int> > &matrix) {
        if (matrix.empty() || matrix[0].empty()) return;
        int m = matrix.size(), n = matrix[0].size();
        bool rowZero = false, colZero = false;
        for (int i = 0; i < m; ++i) {
            if (matrix[i][0] == 0) colZero = true;
        }
        for (int i = 0; i < n; ++i) {
            if (matrix[0][i] == 0) rowZero = true;
        }
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
```

```
                if (matrix[i][j] == 0) {
                    matrix[0][j] = 0;
                    matrix[i][0] = 0;
                }
            }
        }
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                if (matrix[0][j] == 0 || matrix[i][0] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }
        if (rowZero) {
            for (int i = 0; i < n; ++i) matrix[0][i] = 0;
        }
        if (colZero) {
            for (int i = 0; i < m; ++i) matrix[i][0] = 0;
        }
    }
};
```

74. Search a 2D Matrix

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
```

这道题要求搜索一个二维矩阵，由于给的矩阵是有序的，所以很自然的想到要用二分查找法，可以在第一列上先用一次二分查找法找到目标值所在的行的位置，然后在该行上再用一次二分查找法来找是否存在目标值。对于第一个二分查找，由于第一列的数中可能没有 target 值，该如何查找呢，是博主之前的总结帖 LeetCode Binary Search Summary 二分搜索法小结 中的哪一类呢？如果是查找第一个不小于目标值的数，当 target 在第一列时，会返回 target 所在的行，但若 target 不在的话，有可能会返回下一行，不好统一。所以可以查找第一个大于目标值的数，也就是总结帖中的第三类，这样只要回退一个，就一定是 target 所在的行。但需要注意的一点是，如果返回的是0，就不能回退了，以免越界，记得要判断一下。找到了 target 所在的行数，就可以再次使用二分搜索，此时就是总结帖中的第一类了，查找和 target 值相同的数，也是最简单的一类，分分钟搞定即可，参见代码如下：

解法一：

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        if (matrix.empty() || matrix[0].empty()) return false;
        int left = 0, right = matrix.size();
        while (left < right) {
            int mid = (left + right) / 2;
            if (matrix[mid][0] == target) return true;
            if (matrix[mid][0] <= target) left = mid + 1;
            else right = mid;
        }
        int tmp = (right > 0) ? (right - 1) : right;
        left = 0;
        right = matrix[tmp].size();
        while (left < right) {
            int mid = (left + right) / 2;
            if (matrix[tmp][mid] == target) return true;
            if (matrix[tmp][mid] < target) left = mid + 1;
            else right = mid;
        }
        return false;
    }
};
```

当然这道题也可以使用一次二分查找法，如果我们按S型遍历该二维数组，可以得到一个有序的一维数组，只需要用一次二分查找法，而关键就在于坐标的转换，如何把二维坐标和一维坐标转换是关键点，把一个长度为n的一维数组转化为 m*n 的二维数组* (*m*n = n)后，那么原一维数组中下标为i的元素将出现在二维数组中的 [i/n][i%n] 的位置，有了这一点，代码很好写出来了：

解法二：

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        if (matrix.empty() || matrix[0].empty()) return false;
        int m = matrix.size(), n = matrix[0].size();
        int left = 0, right = m * n;
        while (left < right) {
            int mid = (left + right) / 2;
            if (matrix[mid / n][mid % n] == target) return true;
            if (matrix[mid / n][mid % n] < target) left = mid + 1;
            else right = mid;
        }
        return false;
    }
};
```

这道题其实也可以不用二分搜索法，直接使用双指针也是可以的，i指向0，j指向列数，这样第一个被验证的数就是二维数组右上角的数字，假如这个数字等于 target，直接返回 true；若大于 target，说明要减小数字，则列数j自减1；若小于 target，说明要增加数字，行数i自增1。若 while 循环退出了还是没找到 target，直接返回 false 即可，参见代码如下：

解法三：

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        if (matrix.empty() || matrix[0].empty()) return false;
        int i = 0, j = (int)matrix[0].size() - 1;
        while (i < matrix.size() && j >= 0) {
            if (matrix[i][j] == target) return true;
            else if (matrix[i][j] > target) --j;
            else ++i;
        }
        return false;
    }
};
```

75. Sort Colors

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?

这道题的本质还是一道排序的题，题目中给出提示说可以用计数排序，需要遍历数组两遍，那么先来看这种方法，因为数组中只有三个不同的元素，所以实现起来很容易。

- 首先遍历一遍原数组，分别记录 0，1，2 的个数。
- 然后更新原数组，按个数分别赋上 0，1，2。

解法一：

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        vector<int> colors(3);
        for (int num : nums) ++colors[num];
        for (int i = 0, cur = 0; i < 3; ++i) {
            for (int j = 0; j < colors[i]; ++j) {
                nums[cur++] = i;
            }
        }
    }
};
```

题目中还要让只遍历一次数组来求解，那么我需要用双指针来做，分别从原数组的首尾往中心移动。

- 定义red指针指向开头位置，blue指针指向末尾位置。

- 从头开始遍历原数组，如果遇到0，则交换该值和red指针指向的值，并将red指针后移一位。若遇到2，则交换该值和blue指针指向的值，并将blue指针前移一位。若遇到1，则继续遍历。

解法二：

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        int red = 0, blue = (int)nums.size() - 1;
        for (int i = 0; i <= blue; ++i) {
            if (nums[i] == 0) {
                swap(nums[i], nums[red++]);
            } else if (nums[i] == 2) {
                swap(nums[i--], nums[blue--]);
            }
        }
    }
};
```

当然我们也可以使用 while 循环的方式来写，那么就需要一个变量 cur 来记录当前遍历到的位置，参见代码如下：

解法三：

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        int left = 0, right = (int)nums.size() - 1, cur = 0;
        while (cur <= right) {
            if (nums[cur] == 0) {
                swap(nums[cur++], nums[left++]);
            } else if (nums[cur] == 2) {
                swap(nums[cur], nums[right--]);
            } else {
                ++cur;
            }
        }
    }
};
```

76. Minimum Window Substring

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.

这道题给了我们一个原字符串S，还有一个目标字符串T，让在S中找到一个最短的子串，使得其包含了T中的所有的字母，并且限制了时间复杂度为 O(n)。这道题的要求是要在 O(n) 的时间度里实现找到这个最小窗口字串，暴力搜索 Brute Force 肯定是不能用的，因为遍历所有的子串的时间复杂度是平方级的。那么来想一下，时间复杂度卡的这么严，说明必须在一次遍历中完成任务，当然遍历若干次也是O(n)，但不一定有这个必要，尝试就一次遍历拿下！那么再来想，既然要包含T中所有的字母，那么对于T中的每个字母，肯定要快速查找是否在子串中，既然总时间都卡在了 O(n)，肯定不想在这里还浪费时间，就用空间换时间（也就算法题中可以这么干了，七老八十的富翁就算用大别野也换不来时间啊。依依东望，望的就是时间呐 T.T），使用 HashMap，建立T中每个字母与其出现次数之间的映射，那么你可能会有疑问，为啥不用 HashSet 呢，别急，讲到后面你就知道用 HashMap 有多妙，简直妙不可言～

目前在脑子一片浆糊的情况下，我们还是从简单的例子来分析吧，题目例子中的S有点长，换个短的 S = "ADBANC"，T = "ABC"，那么肉眼遍历一遍S呗，首先第一个是A，嗯很好，T中有，第二个是D，T中没有，不理它，第三个是B，嗯很好，T中有，第四个又是A，多了一个，礼多人不怪嘛，收下啦，第五个是N，一边凉快去，第六个终于是C了，那么貌似好像需要整个S串，其实不然，注意之前有多一个A，就算去掉第一个A，也没事，因为第四个A可以代替之，第二个D也可以去掉，因为不在T串中，第三个B就不能再去掉了，不然就没有B了。所以最终的答案就"BANC"了。通过上面的描述，你有没有发现一个有趣的现象，先扩展，再收缩，就好像一个窗口一样，先扩大右边界，然后再收缩左边界，上面的例子中右边界无法扩大了后才开始收缩左边界，实际上对于复杂的例子，有可能是扩大右边界，然后缩小一下左边界，然后再扩大右边界等等。这就很像一个不停滑动的窗口了，这就是大名鼎鼎的滑动窗口 Sliding Window 了，简直是神器啊，能解很多子串，子数组，子序列等等的问题，是必须要熟练掌握的啊！

下面来考虑用代码来实现，先来回答一下前面埋下的伏笔，为啥要用 HashMap，而不是 HashSet，现在应该很显而易见了吧，因为要统计T串中字母的个数，而不是仅仅看某个字母是否在T串中出现。统计好T串中字母的个数了之后，开始遍历S串，对于S中的每个遍历到的字母，都在 HashMap 中的映射值减1，如果减1后的映射值仍大于等于0，说明当前遍历到的字母是T串中的字母，使用一个计数器 cnt，使其自增1。当 cnt 和T串字母个数相等时，说明此时的窗口已经包含了T串中的所有字母，此时更新一个 minLen 和结果 res，这里的 minLen 是一个全局变量，用来记录出现过的包含T串所有字母的最短的子串的长度，结果 res 就是这个最短的子串。然后开始收缩左边界，由于遍历的时候，对映射值减了1，所以此时去除字母的时候，就要把减去的1加回来，此时如果加1后的值大于0了，说明此时少了一个T中的字母，那么 cnt 值就要减1了，然后移动左边界 left。你可能会疑问，对于不在T串中的字母的映射值也这么加呀减呀的，真的大丈夫（带胶布）吗？其实没啥事，因为对于不在T串中的字母，减1后，变-1，cnt 不会增加，之后收缩左边界的时候，映射值加1后为0，cnt 也不会减少，所以并没有什么影响啦，下面是具体的步骤啦：

- 先扫描一遍T，把对应的字符及其出现的次数存到 HashMap 中。

- 然后开始遍历S，就把遍历到的字母对应的 HashMap 中的 value 减一，如果减1后仍大于等于0，cnt 自增1。

- 如果 cnt 等于T串长度时，开始循环，纪录一个字串并更新最小字串值。然后将子窗口的左边界向右移，如果某个移除掉的字母是T串中不可缺少的字母，那么 cnt 自减1，表示此时T串并没有完全匹配。

解法一：

```
class Solution {
public:
    string minWindow(string s, string t) {
        string res = "";
        unordered_map<char, int> letterCnt;
        int left = 0, cnt = 0, minLen = INT_MAX;
        for (char c : t) ++letterCnt[c];
        for (int i = 0; i < s.size(); ++i) {
            if (--letterCnt[s[i]] >= 0) ++cnt;
            while (cnt == t.size()) {
                if (minLen > i - left + 1) {
                    minLen = i - left + 1;
                    res = s.substr(left, minLen);
                }
                if (++letterCnt[s[left]] > 0) --cnt;
                ++left;
            }
        }
        return res;
    }
};
```

这道题也可以不用 HashMap，直接用个 int 的数组来代替，因为 ASCII 只有256个字符，所以用个大小为 256 的 int 数组即可代替 HashMap，但由于一般输入字母串的字符只有 128 个，所以也可以只用 128，其余部分的思路完全相同，虽然只改了一个数据结构，但是运行速度提高了一倍，说明数组还是比 HashMap 快啊。在热心网友 chAngelts 的提醒下，还可以进一步的优化，没有必要每次都计算子串，只要有了起始位置和长度，就能唯一的确定一个子串。这里使用一个全局变量 minLeft 来记录最终结果子串的起始位置，初始化为 -1，最终配合上 minLen，就可以得到最终结果了。注意在返回的时候要检测一下若 minLeft 仍为初始值 -1，需返回空串，参见代码如下：

解法二：

```
class Solution {
public:
    string minWindow(string s, string t) {
        vector<int> letterCnt(128, 0);
        int left = 0, cnt = 0, minLeft = -1, minLen = INT_MAX;
        for (char c : t) ++letterCnt[c];
        for (int i = 0; i < s.size(); ++i) {
            if (--letterCnt[s[i]] >= 0) ++cnt;
            while (cnt == t.size()) {
                if (minLen > i - left + 1) {
                    minLen = i - left + 1;
                    minLeft = left;
                }
                if (++letterCnt[s[left]] > 0) --cnt;
                ++left;
            }
```

```
        }
        return minLeft == -1 ? "" : s.substr(minLeft, minLen);
    }
};
```

77. Combinations

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

For example,
If *n* = 4 and *k* = 2, a solution is:

```
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

这道题让求1到n共n个数字里k个数的组合数的所有情况，还是要用深度优先搜索DFS来解，根据以往的经验，像这种要求出所有结果的集合，一般都是用DFS调用递归来解。那么我们建立一个保存最终结果的大集合res，还要定义一个保存每一个组合的小集合out，每次放一个数到out里，如果out里数个数到了k个，则把out保存到最终结果中，否则在下一层中继续调用递归。网友u010500263的博客里有一张图很好的说明了递归调用的顺序，请点击这里。根据上面分析，可写出代码如下：

解法一：

```
class Solution {
public:
    vector<vector<int>> combine(int n, int k) {
        vector<vector<int>> res;
        vector<int> out;
        helper(n, k, 1, out, res);
        return res;
    }
    void helper(int n, int k, int level, vector<int>& out,
vector<vector<int>>& res) {
        if (out.size() == k) {res.push_back(out); return;}
        for (int i = level; i <= n; ++i) {
            out.push_back(i);
            helper(n, k, i + 1, out, res);
            out.pop_back();
        }
    }
};
```

对于n = 5, k = 3, 处理的结果如下：

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

我们再来看一种递归的写法，此解法没用helper当递归函数，而是把本身就当作了递归函数，写起来十分的简洁，也是非常有趣的一种解法。这个解法用到了一个重要的性质 C(n, k) = C(n-1, k-1) + C(n-1, k)，这应该在我们高中时候学排列组合的时候学过吧，博主也记不清了。总之，翻译一下就是，在n个数中取k个数的组合项个数，等于在n-1个数中取k-1个数的组合项个数再加上在n-1个数中取k个数的组合项个数之和。这里博主就不证明了，因为我也不会，就直接举题目中的例子来说明吧：

C(4, 2) = C(3, 1) + C(3, 2)

我们不难写出 C(3, 1) 的所有情况：[1], [2], [3]，还有 C(3, 2) 的所有情况：[1, 2], [1, 3], [2, 3]。我们发现二者加起来为6，正好是 C(4, 2) 的个数之和。但是我们仔细看会发现，C(3, 2)的所有情况包含在 C(4, 2) 之中，但是 C(3, 1) 的每种情况只有一个数字，而我们需要的结果k=2，其实很好办，每种情况后面都加上4，于是变成了：[1, 4], [2, 4], [3, 4]，加上C(3, 2) 的所有情况：[1, 2], [1, 3], [2, 3]，正好就得到了 n=4, k=2 的所有情况了。参见代码如下：

解法二：

```
class Solution {
public:
    vector<vector<int>> combine(int n, int k) {
        if (k > n || k < 0) return {};
        if (k == 0) return {{}};
        vector<vector<int>> res = combine(n - 1, k - 1);
        for (auto &a : res) a.push_back(n);
        for (auto &a : combine(n - 1, k)) res.push_back(a);
        return res;
    }
};
```

我们再来看一种迭代的写法，也是一种比较巧妙的方法。这里每次先递增最右边的数字，存入结果res中，当右边的数字超过了n，则增加其左边的数字，然后将当前数组赋值为左边的数字，再逐个递增，直到最左边的数字也超过了n，停止循环。对于n=4, k=2时，遍历的顺序如下所示：

```
0 0 #initialization
1 0
1 1
1 2 #push_back
```

```
1 3 #push_back
1 4 #push_back
1 5
2 5
2 2
2 3 #push_back
2 4 #push_back
...
3 4 #push_back
3 5
4 5
4 4
4 5
5 5 #stop
```

解法三：

```
class Solution {
public:
    vector<vector<int>> combine(int n, int k) {
        vector<vector<int>> res;
        vector<int> out(k, 0);
        int i = 0;
        while (i >= 0) {
            ++out[i];
            if (out[i] > n) --i;
            else if (i == k - 1) res.push_back(out);
            else {
                ++i;
                out[i] = out[i - 1];
            }
        }
        return res;
    }
};
```

78. Subsets

Given a set of distinct integers, *S* , return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example,
If *S* = `[1,2,3]`, a solution is:

```
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

这道求子集合的问题，由于其要列出所有结果，按照以往的经验，肯定要是要用递归来做。这道题其实它的非递归解法相对来说更简单一点，下面我们先来看非递归的解法，由于题目要求子集合中数字的顺序是非降序排列的，所有我们需要预处理，先给输入数组排序，然后再进一步处理，最开始我在想的时候，是想按照子集的长度由少到多全部写出来，比如子集长度为0的就是空集，空集是任何集合的子集，满足条件，直接加入。下面长度为1的子集，直接一个循环加入所有数字，子集长度为2的话可以用两个循环，但是这种想法到后面就行不通了，因为循环的个数不能无限的增长，所以我们必须换一种思路。我们可以一位一位的网上叠加，比如对于题目中给的例子 [1,2,3] 来说，最开始是空集，那么我们现在要处理1，就在空集上加1，为 [1]，现在我们有两个自己 [] 和 [1]，下面我们来处理2，我们在之前的子集基础上，每个都加个2，可以分别得到 [2]，[1, 2]，那么现在所有的子集合为 [], [1], [2], [1, 2]，同理处理3的情况可得 [3], [1, 3], [2, 3], [1, 2, 3], 再加上之前的子集就是所有的子集合了，代码如下：

解法一：

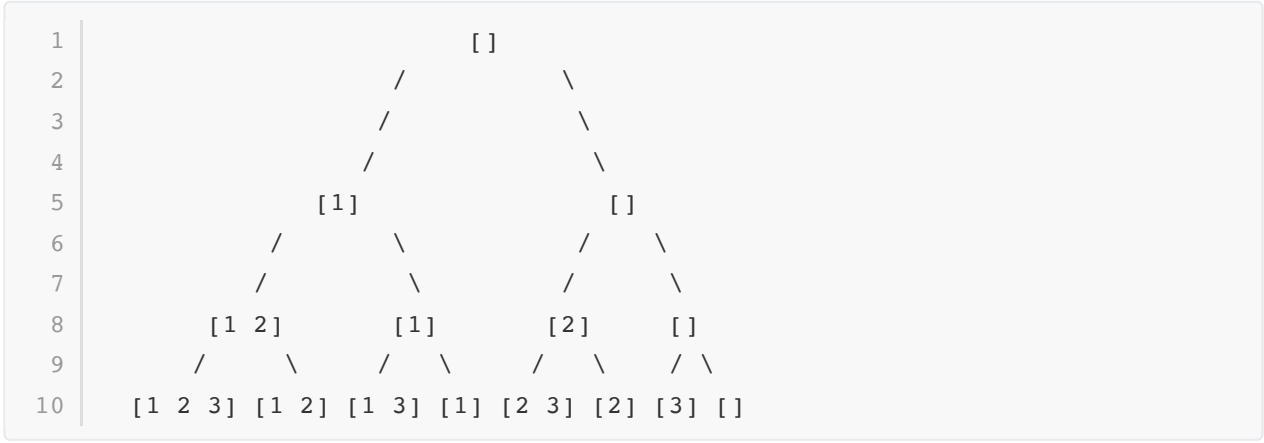
```
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        vector<vector<int> > res(1);
        sort(S.begin(), S.end());
        for (int i = 0; i < S.size(); ++i) {
            int size = res.size();
            for (int j = 0; j < size; ++j) {
                res.push_back(res[j]);
                res.back().push_back(S[i]);
            }
        }
        return res;
    }
};
```

整个添加的顺序为：

[]
[1]
[2]
[1 2]
[3]
[1 3]

[2 3]
[1 2 3]

下面来看递归的解法，相当于一种深度优先搜索，参见网友 JustDoIt的博客，由于原集合每一个数字只有两种状态，要么存在，要么不存在，那么在构造子集时就有选择和不选择两种情况，所以可以构造一棵二叉树，左子树表示选择该层处理的节点，右子树表示不选择，最终的叶节点就是所有子集合，树的结构如下：



解法二：

```
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        vector<vector<int> > res;
        vector<int> out;
        sort(S.begin(), S.end());
        getSubsets(S, 0, out, res);
        return res;
    }
    void getSubsets(vector<int> &S, int pos, vector<int> &out,
vector<vector<int> > &res) {
        res.push_back(out);
        for (int i = pos; i < S.size(); ++i) {
            out.push_back(S[i]);
            getSubsets(S, i + 1, out, res);
            out.pop_back();
        }
    }
};
```

整个添加的顺序为：

[]
[1]
[1 2]
[1 2 3]
[1 3]
[2]

[2 3]
[3]

最后我们再来看一种解法，这种解法是 CareerCup 书上给的一种解法，想法也比较巧妙，把数组中所有的数分配一个状态，true 表示这个数在子集中出现，false 表示在子集中不出现，那么对于一个长度为n的数组，每个数字都有出现与不出现两种情况，所以共有 2n 中情况，那么我们把每种情况都转换出来就是子集了，我们还是用题目中的例子, [1 2 3] 这个数组共有8个子集，每个子集的序号的二进制表示，把是1的位对应原数组中的数字取出来就是一个子集，八种情况都取出来就是所有的子集了，参见代码如下：

	1	2	3	Subset
0	F	F	F	[]
1	F	F	T	3
2	F	T	F	2
3	F	T	T	23
4	T	F	F	1
5	T	F	T	13
6	T	T	F	12
7	T	T	T	123

解法三：

```
class Solution {
public:
    vector<vector<int> > subsets(vector<int> &S) {
        vector<vector<int> > res;
        sort(S.begin(), S.end());
        int max = 1 << S.size();
        for (int k = 0; k < max; ++k) {
            vector<int> out = convertIntToSet(S, k);
            res.push_back(out);
        }
        return res;
    }
    vector<int> convertIntToSet(vector<int> &S, int k) {
        vector<int> sub;
        int idx = 0;
        for (int i = k; i > 0; i >>= 1) {
            if ((i & 1) == 1) {
                sub.push_back(S[idx]);
            }
            ++idx;
        }
```

```
        return sub;
    }
};
```

79. Word Search

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.

For example,
Given board =

```
[
  ["ABCE"],
  ["SFCS"],
  ["ADEE"]
]
```

word = `"ABCCED"`, -> returns `true`,
word = `"SEE"`, -> returns `true`,
word = `"ABCB"`, -> returns `false`.

这道题是典型的深度优先遍历 DFS 的应用，原二维数组就像是一个迷宫，可以上下左右四个方向行走，我们以二维数组中每一个数都作为起点和给定字符串做匹配，我们还需要一个和原数组等大小的 visited 数组，是 bool 型的，用来记录当前位置是否已经被访问过，因为题目要求一个 cell 只能被访问一次。如果二维数组 board 的当前字符和目标字符串 word 对应的字符相等，则对其上下左右四个邻字符分别调用 DFS 的递归函数，只要有一个返回 true，那么就表示可以找到对应的字符串，否则就不能找到，具体看代码实现如下：

解法一：

```
class Solution {
public:
    bool exist(vector<vector<char>>& board, string word) {
        if (board.empty() || board[0].empty()) return false;
        int m = board.size(), n = board[0].size();
        vector<vector<bool>> visited(m, vector<bool>(n));
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (search(board, word, 0, i, j, visited)) return true;
            }
        }
        return false;
    }
```

```
    bool search(vector<vector<char>>& board, string word, int idx, int i, int j, vector<vector<bool>>& visited) {
        if (idx == word.size()) return true;
        int m = board.size(), n = board[0].size();
        if (i < 0 || j < 0 || i >= m || j >= n || visited[i][j] || board[i][j] != word[idx]) return false;
        visited[i][j] = true;
        bool res = search(board, word, idx + 1, i - 1, j, visited)
                || search(board, word, idx + 1, i + 1, j, visited)
                || search(board, word, idx + 1, i, j - 1, visited)
                || search(board, word, idx + 1, i, j + 1, visited);
        visited[i][j] = false;
        return res;
    }
};
```

我们还可以不用 visited 数组，直接对 board 数组进行修改，将其遍历过的位置改为井号，记得递归调用完后需要恢复之前的状态，参见代码如下：

解法二：

```
class Solution {
public:
    bool exist(vector<vector<char>>& board, string word) {
        if (board.empty() || board[0].empty()) return false;
        int m = board.size(), n = board[0].size();
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (search(board, word, 0, i, j)) return true;
            }
        }
        return false;
    }
    bool search(vector<vector<char>>& board, string word, int idx, int i, int j) {
        if (idx == word.size()) return true;
        int m = board.size(), n = board[0].size();
        if (i < 0 || j < 0 || i >= m || j >= n || board[i][j] != word[idx]) return false;
        char c = board[i][j];
        board[i][j] = '#';
        bool res = search(board, word, idx + 1, i - 1, j)
                || search(board, word, idx + 1, i + 1, j)
                || search(board, word, idx + 1, i, j - 1)
                || search(board, word, idx + 1, i, j + 1);
        board[i][j] = c;
        return res;
    }
};
```

80. Remove Duplicates from Sorted Array II

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.
```

Clarification:

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by reference, which means modification to the input array will be known to the caller as well.

Internally you can think of this:

```
// nums is passed in by reference. (i.e., without making a copy)
int len = removeDuplicates(nums);

// any modification to nums in your function would be known by the caller.
// using the length returned by your function, it prints the first len elements.
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

这道题是之前那道 Remove Duplicates from Sorted Array 的拓展，这里允许最多重复的次数是两次，那么可以用一个变量 cnt 来记录还允许有几次重复，cnt 初始化为1，如果出现过一次重复，则 cnt 递减1，那么下次再出现重复，快指针直接前进一步，如果这时候不是重复的，则 cnt 恢复1，由于整个数组是有序的，所以一旦出现不重复的数，则一定比这个数大，此数之后不会再有重复项。理清了上面的思路，则代码很好写了：

解法一：

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int pre = 0, cur = 1, cnt = 1, n = nums.size();
        while (cur < n) {
            if (nums[pre] == nums[cur] && cnt == 0) ++cur;
            else {
                if (nums[pre] == nums[cur]) --cnt;
                else cnt = 1;
                nums[++pre] = nums[cur++];
            }
        }
        return nums.empty() ? 0 : pre + 1;
    }
};
```

这里其实也可以用类似于 Remove Duplicates from Sorted Array 中的解法三的模版，由于这里最多允许两次重复，那么当前的数字 num 只要跟上上个覆盖位置的数字 nusm[i-2] 比较，若 num 较大，则绝不会出现第三个重复数字（前提是数组是有序的），这样的话根本不需要管 nums[i-1] 是否重复，只要将重复个数控制在2个以内就可以了，参见代码如下：

解法二：

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int i = 0;
        for (int num : nums) {
            if (i < 2 || num > nums[i - 2]) {
                nums[i++] = num;
            }
        }
        return i;
    }
};
```

81. Search in Rotated Sorted Array II

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
```

Follow up:

- This is a follow up problem to Search in Rotated Sorted Array, where `nums` may contain duplicates.
- Would this affect the run-time complexity? How and why?

这道是之前那道 Search in Rotated Sorted Array 的延伸，现在数组中允许出现重复数字，这个也会影响我们选择哪半边继续搜索，由于之前那道题不存在相同值，我们在比较中间值和最右值时就完全符合之前所说的规律： **如果中间的数小于最右边的数，则右半段是有序的，若中间数大于最右边数，则左半段是有序的** 。而如果可以有重复值，就会出现来面两种情况，[3 1 1] 和 [1 1 3 1]，对于这两种情况中间值等于最右值时，目标值3既可以在左边又可以在右边，那怎么办么，对于这种情况其实处理非常简单，只要把最右值向左一位即可继续循环，如果还相同则继续移，直到移到不同值为止，然后其他部分还采用 Search in Rotated Sorted Array 中的方法，可以得到代码如下：

```
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        int n = nums.size(), left = 0, right = n - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] == target) return true;
            if (nums[mid] < nums[right]) {
                if (nums[mid] < target && nums[right] >= target) left = mid + 1;
                else right = mid - 1;
            } else if (nums[mid] > nums[right]){
                if (nums[left] <= target && nums[mid] > target) right = mid - 1;
                else left = mid + 1;
            } else --right;
        }
        return false;
```

```
    }
};
```

82. Remove Duplicates from Sorted List II

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
```

和之前那道 Remove Duplicates from Sorted List 不同的地方是这里要删掉所有的重复项，由于链表开头可能会有重复项，被删掉的话头指针会改变，而最终却还需要返回链表的头指针。所以需要定义一个新的节点，然后链上原链表，然后定义一个前驱指针和一个现指针，每当前驱指针指向新建的节点，现指针从下一个位置开始往下遍历，遇到相同的则继续往下，直到遇到不同项时，把前驱指针的next指向下面那个不同的元素。如果现指针遍历的第一个元素就不相同，则把前驱指针向下移一位。代码如下：

解法一：

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head || !head->next) return head;
        ListNode *dummy = new ListNode(-1), *pre = dummy;
        dummy->next = head;
        while (pre->next) {
            ListNode *cur = pre->next;
            while (cur->next && cur->next->val == cur->val) {
                cur = cur->next;
            }
            if (cur != pre->next) pre->next = cur->next;
            else pre = pre->next;
        }
        return dummy->next;
    }
};
```

同样，我们也可以使用递归来做，首先判空，如果 head 为空，直接返回。然后判断，若 head 之后的结点存在，且值相等，那么先进行一个 while 循环，跳过后面所有值相等的结点，到最后一个值相等的点停下。比如对于例子2来说，head 停在第三个结点1处，然后对后面一个结点调用递归函数，即结点2，这样做的好处是，返回的值就完全把所有的结点1都删掉了。若 head 之后的结点值不同，那么还是对 head 之后的结点调用递归函数，将返回值连到 head 的后面，这样 head 结点还是保留下来了，因为值不同嘛，最后返回 head 即可，参见代码如下：

解法二：

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head) return head;
        if (head->next && head->val == head->next->val) {
            while (head->next && head->val == head->next->val) {
                head = head->next;
            }
            return deleteDuplicates(head->next);
        }
        head->next = deleteDuplicates(head->next);
        return head;
    }
};
```

84. Largest Rectangle in Histogram

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.

For example,
Given height = `[2,1,5,6,2,3]`,
return `10`.

这道题让求直方图中最大的矩形，刚开始看到求极值问题以为要用DP来做，可是想不出递推式，只得作罢。这道题如果用暴力搜索法估计肯定没法通过OJ，但是我也没想出好的优化方法，在网上搜到了网友水中的鱼的博客，发现他想出了一种很好的优化方法，就是遍历数组，每找到一个局部峰值（只要当前的数字大于后面的一个数字，那么当前数字就看作一个局部峰值，跟前面的数字大小无关），然后向前

遍历所有的值，算出共同的矩形面积，每次对比保留最大值。这里再说下为啥要从局部峰值处理，看题目中的例子，局部峰值为 2，6，3，我们只需在这些局部峰值出进行处理，为啥不用在非局部峰值处统计呢，这是因为非局部峰值处的情况，后面的局部峰值都可以包括，比如1和5，由于局部峰值6是高于1和5的，所有1和5能组成的矩形，到6这里都能组成，并且还可以加上6本身的一部分组成更大的矩形，那么就不用费力气去再统计一个1和5处能组成的矩形了。代码如下：

解法一：

```
// Pruning optimize
class Solution {
public:
    int largestRectangleArea(vector<int> &height) {
        int res = 0;
        for (int i = 0; i < height.size(); ++i) {
            if (i + 1 < height.size() && height[i] <= height[i + 1]) {
                continue;
            }
            int minH = height[i];
            for (int j = i; j >= 0; --j) {
                minH = min(minH, height[j]);
                int area = minH * (i - j + 1);
                res = max(res, area);
            }
        }
        return res;
    }
};
```

后来又在网上发现一种比较流行的解法，是利用栈来解，可参见网友实验室小纸贴校外版的博客，但是经过仔细研究，其核心思想跟上面那种剪枝的方法有异曲同工之妙，这里维护一个栈，用来保存递增序列，相当于上面那种方法的找局部峰值。我们可以看到，直方图矩形面积要最大的话，需要尽可能的使得连续的矩形多，并且最低一块的高度要高。有点像木桶原理一样，总是最低的那块板子决定桶的装水量。那么既然需要用单调栈来做，首先要考虑到底用递增栈，还是用递减栈来做。我们想啊，递增栈是维护递增的顺序，当遇到小于栈顶元素的数就开始处理，而递减栈正好相反，维护递减的顺序，当遇到大于栈顶元素的数开始处理。那么根据这道题的特点，我们需要按从高板子到低板子的顺序处理，先处理最高的板子，宽度为1，然后再处理旁边矮一些的板子，此时长度为2，因为之前的高板子可组成矮板子的矩形，因此我们需要一个递增栈，当遇到大的数字直接进栈，而当遇到小于栈顶元素的数字时，就要取出栈顶元素进行处理了，那取出的顺序就是从高板子到矮板子了，于是乎遇到的较小的数字只是一个触发，表示现在需要开始计算矩形面积了，为了使得最后一块板子也被处理，这里用了个小 trick，在高度数组最后面加上一个0，这样原先的最后一个板子也可以被处理了。由于栈顶元素是矩形的高度，那么关键就是求出来宽度，那么跟之前那道 Trapping Rain Water 一样，单调栈中不能放高度，而是需要放坐标。由于我们先取出栈中最高的板子，那么就可以先算出长度为1的矩形面积了，然后再取下一个板子，此时根据矮板子的高度算长度为2的矩形面积，以此类推，知道数字大于栈顶元素为止，再次进栈，巧妙的一比！关于单调栈问题可以参见博主的一篇总结帖 LeetCode Monotonous Stack Summary 单调栈小结，代码如下：

解法二：

```
class Solution {
public:
    int largestRectangleArea(vector<int> &height) {
        int res = 0;
        stack<int> st;
        height.push_back(0);
        for (int i = 0; i < height.size(); ++i) {
            if (st.empty() || height[st.top()] < height[i]) {
                st.push(i);
            } else {
                int cur = st.top(); st.pop();
                res = max(res, height[cur] * (st.empty() ? i : (i - st.top() - 1)));
                --i;
            }
        }
        return res;
    }
};
```

我们可以将上面的方法稍作修改，使其更加简洁一些：

解法三：

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        int res = 0;
        stack<int> st;
        heights.push_back(0);
        for (int i = 0; i < heights.size(); ++i) {
            while (!st.empty() && heights[st.top()] >= heights[i]) {
                int cur = st.top(); st.pop();
                res = max(res, heights[cur] * (st.empty() ? i : (i - st.top() - 1)));
            }
            st.push(i);
        }
        return res;
    }
};
```

85. Maximal Rectangle

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
```

此题是之前那道的 Largest Rectangle in Histogram 的扩展，这道题的二维矩阵每一层向上都可以看做一个直方图，输入矩阵有多少行，就可以形成多少个直方图，对每个直方图都调用 Largest Rectangle in Histogram 中的方法，就可以得到最大的矩形面积。那么这道题唯一要做的就是将每一层都当作直方图的底层，并向上构造整个直方图，由于题目限定了输入矩阵的字符只有 '0' 和 '1' 两种，所以处理起来也相对简单。方法是，对于每一个点，如果是 '0'，则赋0，如果是 '1'，就赋之前的 height 值加上1。具体参见代码如下：

解法一：

```
class Solution {
public:
    int maximalRectangle(vector<vector<char> > &matrix) {
        int res = 0;
        vector<int> height;
        for (int i = 0; i < matrix.size(); ++i) {
            height.resize(matrix[i].size());
            for (int j = 0; j < matrix[i].size(); ++j) {
                height[j] = matrix[i][j] == '0' ? 0 : (1 + height[j]);
            }
            res = max(res, largestRectangleArea(height));
        }
        return res;
    }
    int largestRectangleArea(vector<int>& height) {
        int res = 0;
        stack<int> s;
        height.push_back(0);
        for (int i = 0; i < height.size(); ++i) {
            if (s.empty() || height[s.top()] <= height[i]) s.push(i);
            else {
                int tmp = s.top(); s.pop();
                res = max(res, height[tmp] * (s.empty() ? i : (i - s.top() - 1)));
                --i;
            }
        }
        return res;
    }
```

```
};
```

我们也可以在一个函数内完成，这样代码看起来更加简洁一些，注意这里的 height 初始化的大小为 n+1，为什么要多一个呢？这是因为我们只有在当前位置小于等于前一个位置的高度的时候，才会去计算矩形的面积，假如最后一个位置的高度是最高的，那么我们就没法去计算并更新结果 res 了，所以要在最后再加一个高度0，这样就一定可以计算前面的矩形面积了，这跟上面解法子函数中给 height 末尾加一个0是一样的效果，参见代码如下：

解法二：

```
class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) return 0;
        int res = 0, m = matrix.size(), n = matrix[0].size();
        vector<int> height(n + 1);
        for (int i = 0; i < m; ++i) {
            stack<int> s;
            for (int j = 0; j < n + 1; ++j) {
                if (j < n) {
                    height[j] = matrix[i][j] == '1' ? height[j] + 1 : 0;
                }
                while (!s.empty() && height[s.top()] >= height[j]) {
                    int cur = s.top(); s.pop();
                    res = max(res, height[cur] * (s.empty() ? j : (j - s.top() - 1)));
                }
                s.push(j);
            }
        }
        return res;
    }
};
```

下面这种方法的思路很巧妙，height 数组和上面一样，这里的 left 数组表示若当前位置是1且与其相连都是1的左边界的位置（若当前 height 是0，则当前 left 一定是0），right 数组表示若当前位置是1且与其相连都是1的右边界的位置再加1（加1是为了计算长度方便，直接减去左边界位置就是长度），初始化为n（若当前 height 是0，则当前 right 一定是n），那么对于任意一行的第j个位置，矩形为 (right[j] - left[j]) * height[j]，我们举个例子来说明，比如给定矩阵为：

```
[
  [1, 1, 0, 0, 1],
  [0, 1, 0, 0, 1],
  [0, 0, 1, 1, 1],
  [0, 0, 1, 1, 1],
  [0, 0, 0, 0, 1]
]
```

第0行：

```
h: 1 1 0 0 1
l: 0 0 0 0 4
r: 2 2 5 5 5
```

第1行：

```
h: 0 2 0 0 2
l: 0 1 0 0 4
r: 5 2 5 5 5
```

第2行：

```
h: 0 0 1 1 3
l: 0 0 2 2 4
r: 5 5 5 5 5
```

第3行：

```
h: 0 0 2 2 4
l: 0 0 2 2 4
r: 5 5 5 5 5
```

第4行：

```
h: 0 0 0 0 5
l: 0 0 0 0 4
r: 5 5 5 5 5
```

解法三：

```
class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) return 0;
        int res = 0, m = matrix.size(), n = matrix[0].size();
        vector<int> height(n, 0), left(n, 0), right(n, n);
        for (int i = 0; i < m; ++i) {
            int cur_left = 0, cur_right = n;
            for (int j = 0; j < n; ++j) {
                if (matrix[i][j] == '1') {
                    ++height[j];
                    left[j] = max(left[j], cur_left);
                } else {
                    height[j] = 0;
                    left[j] = 0;
```

```
                    cur_left = j + 1;
                }
            }
            for (int j = n - 1; j >= 0; --j) {
                if (matrix[i][j] == '1') {
                    right[j] = min(right[j], cur_right);
                } else {
                    right[j] = n;
                    cur_right = j;
                }
                res = max(res, (right[j] - left[j]) * height[j]);
            }
        }
        return res;
    }
};
```

再来看一种解法，这里我们统计每一行的连续1的个数，使用一个数组 h_max, 其中 h_max[i][j] 表示第i行，第j个位置水平方向连续1的个数，若 matrix[i][j] 为0，那对应的 h_max[i][j] 也一定为0。统计的过程跟建立累加和数组很类似，唯一不同的是遇到0了要将 h_max 置0。这个统计好了之后，只需要再次遍历每个位置，首先每个位置的 h_max 值都先用来更新结果 res，因为高度为1也可以看作是矩形，然后我们向上方遍历，上方 (i, j-1) 位置也会有 h_max 值，但是用二者之间的较小值才能构成矩形，用新的矩形面积来更新结果 res，这样一直向上遍历，直到遇到0，或者是越界的时候停止，这样就可以找出所有的矩形了，参见代码如下：

解法四：

```
class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) return 0;
        int res = 0, m = matrix.size(), n = matrix[0].size();
        vector<vector<int>> h_max(m, vector<int>(n));
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (matrix[i][j] == '0') continue;
                if (j > 0) h_max[i][j] = h_max[i][j - 1] + 1;
                else h_max[i][0] = 1;
            }
        }
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (h_max[i][j] == 0) continue;
                int mn = h_max[i][j];
                res = max(res, mn);
                for (int k = i - 1; k >= 0 && h_max[k][j] != 0; --k) {
                    mn = min(mn, h_max[k][j]);
                    res = max(res, mn * (i - k + 1));
                }
```

```
            }
        }
        return res;
    }
};
```

86. Partition List

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.

For example,
Given `1->4->3->2->5->2` and *x* = 3,
return `1->2->2->4->3->5`.

这道题要求我们划分链表，把所有小于给定值的节点都移到前面，大于该值的节点顺序不变，相当于一个局部排序的问题。那么可以想到的一种解法是首先找到第一个大于或等于给定值的节点，用题目中给的例子来说就是先找到4，然后再找小于3的值，每找到一个就将其取出置于4之前即可，代码如下：

解法一

```
class Solution {
public:
    ListNode *partition(ListNode *head, int x) {
        ListNode *dummy = new ListNode(-1);
        dummy->next = head;
        ListNode *pre = dummy, *cur = head;;
        while (pre->next && pre->next->val < x) pre = pre->next;
        cur = pre;
        while (cur->next) {
            if (cur->next->val < x) {
                ListNode *tmp = cur->next;
                cur->next = tmp->next;
                tmp->next = pre->next;
                pre->next = tmp;
                pre = pre->next;
            } else {
                cur = cur->next;
            }
        }
        return dummy->next;
    }
};
```

这种解法的链表变化顺序为：

1 -> 4 -> 3 -> 2 -> 5 -> 2

1 -> 2 -> 4 -> 3 -> 5 -> 2

1 -> 2 -> 2 -> 4 -> 3 -> 5

此题还有一种解法，就是将所有小于给定值的节点取出组成一个新的链表，此时原链表中剩余的节点的值都大于或等于给定值，只要将原链表直接接在新链表后即可，代码如下：

解法二

```
class Solution {
public:
    ListNode *partition(ListNode *head, int x) {
        if (!head) return head;
        ListNode *dummy = new ListNode(-1);
        ListNode *newDummy = new ListNode(-1);
        dummy->next = head;
        ListNode *cur = dummy, *p = newDummy;
        while (cur->next) {
            if (cur->next->val < x) {
                p->next = cur->next;
                p = p->next;
                cur->next = cur->next->next;
                p->next = NULL;
            } else {
                cur = cur->next;
            }
        }
        p->next = dummy->next;
        return newDummy->next;
    }
};
```

此种解法链表变化顺序为：

Original: 1 -> 4 -> 3 -> 2 -> 5 -> 2

New:

Original: 4 -> 3 -> 2 -> 5 -> 2

New: 1

Original: 4 -> 3 -> 5 -> 2

New: 1 -> 2

Original: 4 -> 3 -> 5

New: 1 -> 2 -> 2

Original:

New: 1 -> 2 -> 2 -> 4 -> 3 -> 5

87. Scramble String

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
```

这道题定义了一种搅乱字符串，就是说假如把一个字符串当做一个二叉树的根，然后它的非空子字符串是它的子节点，然后交换某个子字符串的两个子节点，重新爬行回去形成一个新的字符串，这个新字符串和原来的字符串互为搅乱字符串。这道题可以用递归 Recursion 或是动态规划 Dynamic Programming 来做，我们先来看递归的解法，参见网友 uniEagle 的博客，**简单的说，就是 s1 和 s2 是 scramble 的话，那么必然存在一个在 s1 上的长度 l1，将 s1 分成 s11 和 s12 两段，同样有 s21 和 s22，那么要么 s11 和 s21 是 scramble 的并且 s12 和 s22 是 scramble 的；要么 s11 和 s22 是 scramble 的并且 s12 和 s21 是 scramble 的。** 就拿题目中的例子 rgeat 和 great 来说，rgeat 可分成 rg 和 eat 两段，great 可分成 gr 和 eat 两段，rg 和 gr 是 scrambled 的，eat 和 eat 当然是 scrambled。根据这点，我们可以写出代码如下：

解法一：

```
// Recursion
class Solution {
public:
    bool isScramble(string s1, string s2) {
        if (s1.size() != s2.size()) return false;
        if (s1 == s2) return true;
        string str1 = s1, str2 = s2;
        sort(str1.begin(), str1.end());
        sort(str2.begin(), str2.end());
        if (str1 != str2) return false;
        for (int i = 1; i < s1.size(); ++i) {
            string s11 = s1.substr(0, i);
            string s12 = s1.substr(i);
            string s21 = s2.substr(0, i);
            string s22 = s2.substr(i);
            if (isScramble(s11, s21) && isScramble(s12, s22)) return true;
            s21 = s2.substr(s1.size() - i);
            s22 = s2.substr(0, s1.size() - i);
            if (isScramble(s11, s21) && isScramble(s12, s22)) return true;
        }
        return false;
    }
};
```

当然，这道题也可以用动态规划 Dynamic Programming，根据以往的经验来说，根字符串有关的题十有八九可以用 DP 来做，那么难点就在于如何找出状态转移方程。参见网友 Code Ganker 的博客，这其实是一道三维动态规划的题目，使用一个三维数组 dp[i][j][n]，其中i是 s1 的起始字符，j是 s2 的起始字符，而n是当前的字符串长度，dp[i][j][len] 表示的是以i和j分别为 s1 和 s2 起点的长度为 len 的字符串是不是互为 scramble。有了 dp 数组接下来看看状态转移方程，也就是怎么根据历史信息来得到 dp[i][j][len]。判断这个是不是满足，首先是把当前 s1[i...i+len-1] 字符串劈一刀分成两部分，然后分两种情况：第一种是左边和 s2[j...j+len-1] 左边部分是不是 scramble，以及右边和 s2[j...j+len-1] 右边部分是不是 scramble；第二种情况是左边和 s2[j...j+len-1] 右边部分是不是 scramble，以及右边和 s2[j...j+len-1] 左边部分是不是 scramble。如果以上两种情况有一种成立，说明 s1[i...i+len-1] 和 s2[j...j+len-1] 是 scramble 的。而对于判断这些左右部分是不是 scramble 是有历史信息的，因为长度小于n的所有情况都在前面求解过了（也就是长度是最外层循环）。上面说的是劈一刀的情况，对于 s1[i...i+len-1] 有 len-1 种劈法，在这些劈法中只要有一种成立，那么两个串就是 scramble 的。总结起来状态转移方程是：

dp[i][j][len] = || (dp[i][j][k] && dp[i+k][j+k][len-k] || dp[i][j+len-k][k] && dp[i+k][j][len-k])

对于所有 1<=k<len，也就是对于所有 len-1 种劈法的结果求或运算。因为信息都是计算过的，对于每种劈法只需要常量操作即可完成，因此求解递推式是需要 O(len)（因为 len-1 种劈法）。如此总时间复杂度因为是三维动态规划，需要三层循环，加上每一步需要线行时间求解递推式，所以是 O(n^4)。虽然已经比较高了，但是至少不是指数量级的，动态规划还是有很大优势的，空间复杂度是 O(n^3)。代码如下：

解法二：

```
// DP
class Solution {
public:
    bool isScramble(string s1, string s2) {
        if (s1.size() != s2.size()) return false;
        if (s1 == s2) return true;
        int n = s1.size();
        vector<vector<vector<bool>>> dp (n, vector<vector<bool>>(n,
vector<bool>(n + 1)));
        for (int len = 1; len <= n; ++len) {
            for (int i = 0; i <= n - len; ++i) {
                for (int j = 0; j <= n - len; ++j) {
                    if (len == 1) {
                        dp[i][j][1] = s1[i] == s2[j];
                    } else {
                        for (int k = 1; k < len; ++k) {
                            if ((dp[i][j][k] && dp[i + k][j + k][len - k])
|| (dp[i + k][j][len - k] && dp[i][j + len - k][k])) {
                                dp[i][j][len] = true;
                            }
                        }
                    }
                }
            }
        }
```

```
        return dp[0][0][n];
    }
};
```

上面的代码的实现过程如下，首先按单个字符比较，判断它们之间是否是 scrambled 的。在更新第二个表中第一个值 (gr 和 rg 是否为 scrambled 的)时，比较了第一个表中的两种构成，一种是 g与r, r与g，另一种是 g与g, r与r，其中后者是真，只要其中一个为真，则将该值赋真。其实这个原理和之前递归的原理很像，在判断某两个字符串是否为 scrambled 时，比较它们所有可能的拆分方法的子字符串是否是 scrambled 的，只要有一个种拆分方法为真，则比较的两个字符串一定是 scrambled 的。比较 rge 和 gre 的实现过程如下所示：

```
     r    g    e
g    x    √    x
r    √    x    x
e    x    x    √


     rg    ge
gr    √    x
re    x    x


     rge
gre   √
```

DP 的另一种写法，参考网友加载中..的博客，思路都一样，代码如下：

解法三：

```
// Still DP
class Solution {
public:
    bool isScramble(string s1, string s2) {
        if (s1.size() != s2.size()) return false;
        if (s1 == s2) return true;
        int n = s1.size();
        vector<vector<vector<bool>>> dp (n, vector<vector<bool>>(n,
vector<bool>(n + 1)));
        for (int i = n - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) {
                for (int k = 1; k <= n - max(i, j); ++k) {
                    if (s1.substr(i, k) == s2.substr(j, k)) {
                        dp[i][j][k] = true;
                    } else {
                        for (int t = 1; t < k; ++t) {
                            if ((dp[i][j][t] && dp[i + t][j + t][k - t])
|| (dp[i][j + k - t][t] && dp[i + t][j][k - t])) {
                                dp[i][j][k] = true;
```

```
                                break;
                            }
                        }
                    }
                }
            }
        }
        return dp[0][0][n];
    }
};
```

下面这种解法和第一个解法思路相同，只不过没有用排序算法，而是采用了类似于求异构词的方法，用一个数组来保存每个字母出现的次数，后面判断 Scramble 字符串的方法和之前的没有区别：

解法四：

```
class Solution {
public:
    bool isScramble(string s1, string s2) {
        if (s1 == s2) return true;
        if (s1.size() != s2.size()) return false;
        int n = s1.size(), m[26] = {0};
        for (int i = 0; i < n; ++i) {
            ++m[s1[i] - 'a'];
            --m[s2[i] - 'a'];
        }
        for (int i = 0; i < 26; ++i) {
            if (m[i] != 0) return false;
        }
        for (int i = 1; i < n; ++i) {
            if ((isScramble(s1.substr(0, i), s2.substr(0, i)) && isScramble(s1.substr(i), s2.substr(i))) || (isScramble(s1.substr(0, i), s2.substr(n - i)) && isScramble(s1.substr(i), s2.substr(0, n - i)))) {
                return true;
            }
        }
        return false;
    }
};
```

下面这种解法实际上是解法二的递归形式，我们用了 memo 数组来减少了大量的运算，注意这里的 memo 数组一定要有三种状态，初始化为 -1，区域内为 scramble 是1，不是 scramble 是0，这样就避免了已经算过了某个区间，但由于不是 scramble，从而又进行一次计算，从而会 TLE，感谢网友 bambu 的提供的思路，参见代码如下：

解法五：

```
class Solution {
public:
```

```
    bool isScramble(string s1, string s2) {
        if (s1 == s2) return true;
        if (s1.size() != s2.size()) return false;
        int n = s1.size();
        vector<vector<vector<int>>> memo(n, vector<vector<int>>(n, vector<int>(n + 1, -1)));
        return helper(s1, s2, 0, 0, n, memo);
    }
    bool helper(string& s1, string& s2, int idx1, int idx2, int len, vector<vector<vector<int>>>& memo) {
        if (len == 0) return true;
        if (len == 1) memo[idx1][idx2][len] = s1[idx1] == s2[idx2];
        if (memo[idx1][idx2][len] != -1) return memo[idx1][idx2][len];
        for (int k = 1; k < len; ++k) {
            if ((helper(s1, s2, idx1, idx2, k, memo) && helper(s1, s2, idx1 + k, idx2 + k, len - k, memo)) || (helper(s1, s2, idx1, idx2 + len - k, k, memo) && helper(s1, s2, idx1 + k, idx2, len - k, memo))) {
                return memo[idx1][idx2][len] = 1;
            }
        }
        return memo[idx1][idx2][len] = 0;
    }
};
```

89. Gray Code

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.

For example, given *n* = 2, return `[0,1,3,2]`. Its gray code sequence is:

```
00 - 0
01 - 1
11 - 3
10 - 2
```

Note:
For a given *n* , a gray code sequence is not uniquely defined.

For example, `[0,2,3,1]` is also a valid gray code sequence according to the above definition.

For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.

这道题是关于格雷码的，猛地一看感觉完全没接触过格雷码，但是看了维基百科后，隐约的感觉原来好像哪门可提到过，哎全还给老师了。这道题如果不了解格雷码，还真不太好做，幸亏脑补了维基百科，上面说格雷码是一种循环二进制单位距离码，主要特点是两个相邻数的代码只有一位二进制数不同的编码，格雷码的处理主要是位操作 Bit Operation，LeetCode中关于位操作的题也挺常见，比如 Repeated DNA Sequences 求重复的DNA序列， Single Number 单独的数字, 和 Single Number II 单独的数字之二 等等。三位的格雷码和二进制数如下：

```
Int    Grey Code    Binary
 0        000         000
 1        001         001
 2        011         010
 3        010         011
 4        110         100
 5        111         101
 6        101         110
 7        100         111
```

其实这道题还有多种解法。首先来看一种最简单的，是用到格雷码和二进制数之间的相互转化，可参见我之前的博客 Convertion of grey code and binary 格雷码和二进制数之间的转换 ，明白了转换方法后，这道题完全没有难度，代码如下：

解法一：

```
// Binary to grey code
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> res;
        for (int i = 0; i < pow(2,n); ++i) {
            res.push_back((i >> 1) ^ i);
        }
        return res;
    }
};
```

然后我们来看看其他的解法，参考维基百科上关于格雷码的性质，有一条是说镜面排列的，n位元的格雷码可以从n-1位元的格雷码以上下镜射后加上新位元的方式快速的得到，如下图所示一般。

有了这条性质，我们很容易的写出代码如下：

解法二：

```
// Mirror arrangement
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> res{0};
```

```
        for (int i = 0; i < n; ++i) {
            int size = res.size();
            for (int j = size - 1; j >= 0; --j) {
                res.push_back(res[j] | (1 << i));
            }
        }
        return res;
    }
};
```

维基百科上还有一条格雷码的性质是直接排列，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。根据这条性质也可以写出代码，不过相比前面的略微复杂，代码如下：

0 0 0
0 0 1
0 1 1
0 1 0
1 1 0
1 1 1
1 0 1
1 0 0

解法三：

```
// Direct arrangement
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> res{0};
        int len = pow(2, n);
        for (int i = 1; i < len; ++i) {
            int pre = res.back();
            if (i % 2 == 1) {
                pre = (pre & (len - 2)) | ((~pre) & 1);
            } else {
                int cnt = 1, t = pre;
                while ((t & 1) != 1) {
                    ++cnt;
                    t >>= 1;
                }
                if ((pre & (1 << cnt)) == 0) pre |= (1 << cnt);
                else pre &= ~(1 << cnt);
            }
            res.push_back(pre);
        }
        return res;
    }
};
```

上面三种解法都需要事先了解格雷码及其性质，假如我们之前并没有接触过格雷码，那么我们其实也可以用比较笨的方法来找出结果，比如下面这种方法用到了一个set来保存已经产生的结果，我们从0开始，遍历其二进制每一位，对其取反，然后看其是否在set中出现过，如果没有，我们将其加入set和结果res中，然后再对这个数的每一位进行遍历，以此类推就可以找出所有的格雷码了，参见代码如下：

解法四：

```
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> res;
        unordered_set<int> s;
        helper(n, s, 0, res);
        return res;
    }
    void helper(int n, unordered_set<int>& s, int out, vector<int>& res) {
        if (!s.count(out)) {
            s.insert(out);
            res.push_back(out);
        }
        for (int i = 0; i < n; ++i) {
            int t = out;
            if ((t & (1 << i)) == 0) t |= (1 << i);
            else t &= ~(1 << i);
            if (s.count(t)) continue;
            helper(n, s, t, res);
            break;
        }
    }
};
```

既然递归方法可以实现，那么就有对应的迭代的写法，当然需要用stack来辅助，参见代码如下：

解法五：

```
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> res{0};
        unordered_set<int> s;
        stack<int> st;
        st.push(0);
        s.insert(0);
        while (!st.empty()) {
            int t = st.top(); st.pop();
            for (int i = 0; i < n; ++i) {
                int k = t;
                if ((k & (1 << i)) == 0) k |= (1 << i);
                else k &= ~(1 << i);
```

```
                if (s.count(k)) continue;
                s.insert(k);
                st.push(k);
                res.push_back(k);
                break;
            }
        }
        return res;
    }
};
```

90. Subsets II

Given a collection of integers that might contain duplicates, *S* , return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example,
If *S* = `[1,2,2]`, a solution is:

```
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

这道子集合之二是之前那道 Subsets 的延伸，这次输入数组允许有重复项，其他条件都不变，只需要在之前那道题解法的基础上稍加改动便可以做出来，我们先来看非递归解法，拿题目中的例子 [1 2 2] 来分析，根据之前 Subsets 里的分析可知，当处理到第一个2时，此时的子集合为 [], [1], [2], [1, 2]，而这时再处理第二个2时，如果在 [] 和 [1] 后直接加2会产生重复，所以只能在上一个循环生成的后两个子集合后面加2，发现了这一点，题目就可以做了，我们用 last 来记录上一个处理的数字，然后判定当前的数字和上面的是否相同，若不同，则循环还是从0到当前子集的个数，若相同，则新子集个数减去之前循环时子集的个数当做起点来循环，这样就不会产生重复了，代码如下：

解法一：

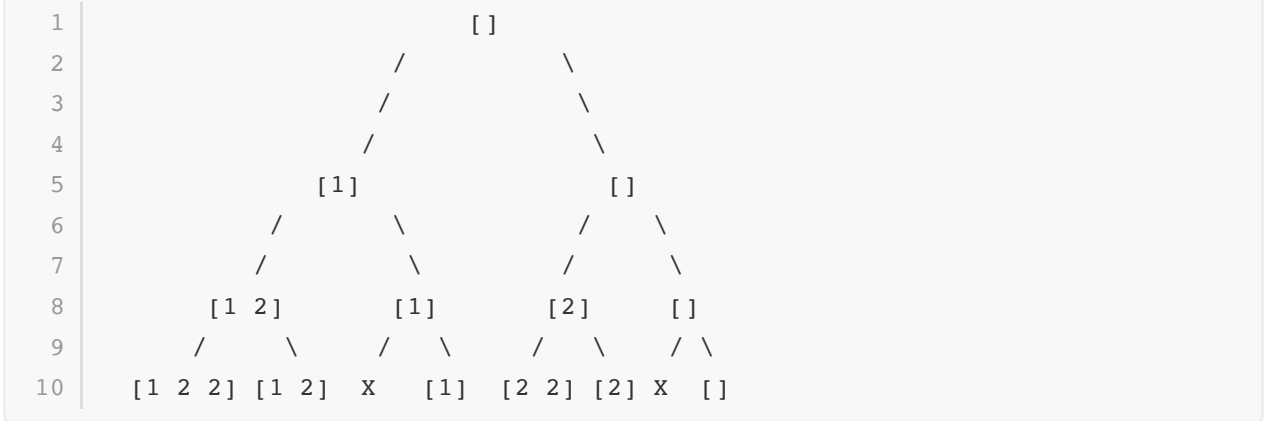
```
class Solution {
public:
    vector<vector<int>> subsetsWithDup(vector<int> &S) {
        if (S.empty()) return {};
        vector<vector<int>> res(1);
        sort(S.begin(), S.end());
```

```
        int size = 1, last = S[0];
        for (int i = 0; i < S.size(); ++i) {
            if (last != S[i]) {
                last = S[i];
                size = res.size();
            }
            int newSize = res.size();
            for (int j = newSize - size; j < newSize; ++j) {
                res.push_back(res[j]);
                res.back().push_back(S[i]);
            }
        }
        return res;
    }
};
```

整个添加的顺序为：

[]
[1]
[2]
[1 2]
[2 2]
[1 2 2]

对于递归的解法，根据之前 Subsets 里的构建树的方法，在处理到第二个2时，由于前面已经处理了一次2，这次我们只在添加过2的 [2] 和 [1 2] 后面添加2，其他的都不添加，那么这样构成的二叉树如下图所示：



代码只需在原有的基础上增加一句话，while (S[i] == S[i + 1]) ++i; 这句话的作用是跳过树中为X的叶节点，因为它们是重复的子集，应被抛弃。代码如下：

解法二：

```
class Solution {
public:
    vector<vector<int>> subsetsWithDup(vector<int> &S) {
        if (S.empty()) return {};
```

```
        vector<vector<int>> res;
        vector<int> out;
        sort(S.begin(), S.end());
        getSubsets(S, 0, out, res);
        return res;
    }
    void getSubsets(vector<int> &S, int pos, vector<int> &out,
vector<vector<int>> &res) {
        res.push_back(out);
        for (int i = pos; i < S.size(); ++i) {
            out.push_back(S[i]);
            getSubsets(S, i + 1, out, res);
            out.pop_back();
            while (i + 1 < S.size() && S[i] == S[i + 1]) ++i;
        }
    }
};
```

整个添加的顺序为：

[]
[1]
[1 2]
[1 2 2]
[2]
[2 2]

91. Decode Ways

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).
```

这道题要求解码方法，跟之前那道 Climbing Stairs 非常的相似，但是还有一些其他的限制条件，比如说一位数时不能为0，两位数不能大于 26，其十位上的数也不能为0，除去这些限制条件，跟爬梯子基本没啥区别，也勉强算特殊的斐波那契数列，当然需要用动态规划 Dynamci Programming 来解。建立一维 dp 数组，其中 dp[i] 表示s中前i个字符组成的子串的解码方法的个数，长度比输入数组长多多1，并将 dp[0] 初始化为1。现在来找状态转移方程，dp[i] 的值跟之前的状态有着千丝万缕的联系，就拿题目中的例子2来分析吧，当 i=1 时，对应s中的字符是 s[0]='2'，只有一种拆分方法，就是2，注意 s[0] 一定不能为0，这样的话无法拆分。当 i=2 时，对应s中的字符是 s[1]='2'，由于 s[1] 不为0，那么其可以被单独拆分出来，就可以在之前 dp[i-1] 的每种情况下都加上一个单独的2，这样 dp[i] 至少可以有跟 dp[i-1] 一样多的拆分情况，接下来还要看其能否跟前一个数字拼起来，若拼起来的两位数小于等于26，并且大于等于 10（因为两位数的高位不能是0），那么就可以在之前 dp[i-2] 的每种情况下都加上这个二位数，所以最终 dp[i] = dp[i-1] + dp[i-2]，是不是发现跟斐波那契数列的性质吻合了。所以0是个很特殊的存在，若当前位置是0，则一定无法单独拆分出来，即不能加上 dp[i-1]，就只能看否跟前一个数字组成大于等于 10 且小于等于 26 的数，能的话可以加上 dp[i-2]，否则就只能保持为0了。具体的操作步骤是，在遍历的过程中，对每个数字首先判断其是否为0，若是则将 dp[i] 赋为0，若不是，赋上 dp[i-1] 的值，然后看数组前一位是否存在，如果存在且满足前一位是1，或者和当前位一起组成的两位数不大于 26，则当前 dp[i] 值加上 dp[i - 2]。最终返回 dp 数组的最后一个值即可，代码如下：

C++ 解法一：

```
class Solution {
public:
    int numDecodings(string s) {
        if (s.empty() || s[0] == '0') return 0;
        vector<int> dp(s.size() + 1, 0);
        dp[0] = 1;
        for (int i = 1; i < dp.size(); ++i) {
            dp[i] = (s[i - 1] == '0') ? 0 : dp[i - 1];
            if (i > 1 && (s[i - 2] == '1' || (s[i - 2] == '2' && s[i - 1] <= '6'))) {
                dp[i] += dp[i - 2];
            }
        }
        return dp.back();
    }
};
```

Java 解法一：

```
class Solution {
    public int numDecodings(String s) {
        if (s.isEmpty() || s.charAt(0) == '0') return 0;
        int[] dp = new int[s.length() + 1];

```

```
        dp[0] = 1;
        for (int i = 1; i < dp.length; ++i) {
            dp[i] = (s.charAt(i - 1) == '0') ? 0 : dp[i - 1];
            if (i > 1 && (s.charAt(i - 2) == '1' || (s.charAt(i - 2) == '2' && s.charAt(i - 1) <= '6'))) {
                dp[i] += dp[i - 2];
            }
        }
        return dp[s.length()];
    }
}
```

下面这种方法跟上面的方法的思路一样，只是写法略有不同：

C++ 解法二：

```
class Solution {
public:
    int numDecodings(string s) {
        if (s.empty() || s[0] == '0') return 0;
        vector<int> dp(s.size() + 1, 0);
        dp[0] = 1;
        for (int i = 1; i < dp.size(); ++i) {
            if (s[i - 1] != '0') dp[i] += dp[i - 1];
            if (i >= 2 && s.substr(i - 2, 2) <= "26" && s.substr(i - 2, 2) >= "10") {
                dp[i] += dp[i - 2];
            }
        }
        return dp.back();
    }
};
```

Java 解法二：

```
class Solution {
    public int numDecodings(String s) {
        if (s.isEmpty() || s.charAt(0) == '0') return 0;
        int[] dp = new int[s.length() + 1];
        dp[0] = 1;
        for (int i = 1; i < dp.length; ++i) {
            if (s.charAt(i - 1) != '0') dp[i] += dp[i - 1];
            if (i >= 2 && (s.substring(i - 2, i).compareTo("10") >= 0 && s.substring(i - 2, i).compareTo("26") <= 0)) {
                dp[i] += dp[i - 2];
            }
        }
        return dp[s.length()];
    }
```

```
}
```

我们再来看一种空间复杂度为 O(1) 的解法，用两个变量 a, b 来分别表示 s[i-1] 和 s[i-2] 的解码方法，然后从 i=1 开始遍历，也就是字符串的第二个字符，判断如果当前字符为 '0'，说明当前字符不能单独拆分出来，只能和前一个字符一起，先将 a 赋为0，然后看前面的字符，如果前面的字符是1或者2时，就可以更新 a = a + b，然后 b = a - b，其实 b 赋值为之前的 a，如果不满足这些条件的话，那么 b = a，参见代码如下：

C++ 解法三：

```
class Solution {
public:
    int numDecodings(string s) {
        if (s.empty() || s[0] == '0') return 0;
        int a = 1, b = 1, n = s.size();
        for (int i = 1; i < n; ++i) {
            if (s[i] == '0') a = 0;
            if (s[i - 1] == '1' || (s[i - 1] == '2' && s[i] <= '6')) {
                a = a + b;
                b = a - b;
            } else {
                b = a;
            }
        }
        return a;
    }
};
```

Java 解法三：

```
class Solution {
    public int numDecodings(String s) {
        if (s.isEmpty() || s.charAt(0) == '0') return 0;
        int a = 1, b = 1, n = s.length();
        for (int i = 1; i < n; ++i) {
            if (s.charAt(i) == '0') a = 0;
            if (s.charAt(i - 1) == '1' || (s.charAt(i - 1) == '2' &&
s.charAt(i) <= '6')) {
                a = a + b;
                b = a - b;
            } else {
                b = a;
            }
        }
        return a;
    }
}
```

92. Reverse Linked List II

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
```

很奇怪为何没有倒置链表之一，就来了这个倒置链表之二，不过猜也能猜得到之一就是单纯的倒置整个链表，而这道作为延伸的地方就是倒置其中的某一小段。对于链表的问题，根据以往的经验一般都是要建一个dummy node，连上原链表的头结点，这样的话就算头结点变动了，我们还可以通过dummy->next来获得新链表的头结点。这道题的要求是只通过一次遍历完成，就拿题目中的例子来说，变换的是2,3,4这三个点，我们需要找到第一个开始变换结点的前一个结点，只要让pre向后走m-1步即可，为啥要减1呢，因为题目中是从1开始计数的，这里只走了1步，就是结点1，用pre指向它。万一是结点1开始变换的怎么办，这就是我们为啥要用dummy结点了，pre也可以指向dummy结点。然后就要开始交换了，由于一次只能交换两个结点，所以我们按如下的交换顺序：

1 -> 2 -> 3 -> 4 -> 5 -> NULL

1 -> 3 -> 2 -> 4 -> 5 -> NULL

1 -> 4 -> 3 -> 2 -> 5 -> NULL

我们可以看出来，总共需要n-m步即可，第一步是将结点3放到结点1的后面，第二步将结点4放到结点1的后面。这是很有规律的操作，那么我们就说一个就行了，比如刚开始，pre指向结点1，cur指向结点2，然后我们建立一个临时的结点t，指向结点3（注意我们用临时变量保存某个结点就是为了首先断开该结点和前面结点之间的联系，这可以当作一个规律记下来），然后我们断开结点2和结点3，将结点2的next连到结点4上，也就是 cur->next = t->next，再把结点3连到结点1的后面结点（即结点2）的前面，即 t->next = pre->next，最后再将原来的结点1和结点2的连接断开，将结点1连到结点3，即 pre->next = t。这样我们就完成了将结点3取出，加入结点1的后方。第二步将结点4取出，加入结点1的后方，也是同样的操作，这里就不多说了，请大家自己尝试下吧，参见代码如下：

```
class Solution {
public:
    ListNode *reverseBetween(ListNode *head, int m, int n) {
        ListNode *dummy = new ListNode(-1), *pre = dummy;
        dummy->next = head;
        for (int i = 0; i < m - 1; ++i) pre = pre->next;
        ListNode *cur = pre->next;
        for (int i = m; i < n; ++i) {
            ListNode *t = cur->next;
            cur->next = t->next;
            t->next = pre->next;
            pre->next = t;
        }
        return dummy->next;
```

```
    }
};
```

93. Restore IP Addresses

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"]
```

这道题要求是复原IP地址，IP地址对我们并不陌生，就算我们不是学CS的，只要我们是广大网友之一，就应该对其并不陌生。IP地址由32位二进制数组成，为便于使用，常以XXX.XXX.XXX.XXX形式表现，每组XXX代表小于或等于255的10进制数。所以说IP地址总共有四段，每一段可能有一位，两位或者三位，范围是[0, 255]，题目明确指出输入字符串只含有数字，所以当某段是三位时，我们要判断其是否越界（>255），还有一点很重要的是，当只有一位时，0可以成某一段，如果有两位或三位时，像 00，01， 001， 011， 000等都是不合法的，所以我们还是需要有一个判定函数来判断某个字符串是否合法。这道题其实也可以看做是字符串的分段问题，在输入字符串中加入三个点，将字符串分为四段，每一段必须合法，求所有可能的情况。根据目前刷了这么多题，得出了两个经验， **一是只要遇到字符串的子序列或配准问题首先考虑动态规划DP，二是只要遇到需要求出所有可能情况首先考虑用递归** 。这道题并非是求字符串的子序列或配准问题，更符合第二种情况，所以我们要用递归来解。我们用k来表示当前还需要分的段数，如果k = 0，则表示三个点已经加入完成，四段已经形成，若这时字符串刚好为空，则将当前分好的结果保存。若k != 0, 则对于每一段，我们分别用一位，两位，三位来尝试，分别判断其合不合法，如果合法，则调用递归继续分剩下的字符串，最终和求出所有合法组合，代码如下：

C++ 解法一：

```
class Solution {
public:
    vector<string> restoreIpAddresses(string s) {
        vector<string> res;
        restore(s, 4, "", res);
        return res;
    }
    void restore(string s, int k, string out, vector<string> &res) {
        if (k == 0) {
            if (s.empty()) res.push_back(out);
        }
        else {
            for (int i = 1; i <= 3; ++i) {
                if (s.size() >= i && isValid(s.substr(0, i))) {
                    if (k == 1) restore(s.substr(i), k - 1, out +
s.substr(0, i), res);
                    else restore(s.substr(i), k - 1, out + s.substr(0, i)
+ ".", res);
```

```
                }
            }
        }
    }
    bool isValid(string s) {
        if (s.empty() || s.size() > 3 || (s.size() > 1 && s[0] == '0')) return false;
        int res = atoi(s.c_str());
        return res <= 255 && res >= 0;
    }
};
```

我们也可以省掉isValid函数，直接在调用递归之前用if语句来滤掉不符合题意的情况，这里面用了k != std::to_string(val).size()，其实并不难理解，比如当k=3时，说明应该是个三位数，但如果字符是"010"，那么转为整型val=10，再转回字符串就是"10"，此时的长度和k值不同了，这样就可以找出不合要求的情况了，参见代码如下；

C++ 解法二：

```
class Solution {
public:
    vector<string> restoreIpAddresses(string s) {
        vector<string> res;
        helper(s, 0, "", res);
        return res;
    }
    void helper(string s, int n, string out, vector<string>& res) {
        if (n == 4) {
            if (s.empty()) res.push_back(out);
        } else {
            for (int k = 1; k < 4; ++k) {
                if (s.size() < k) break;
                int val = atoi(s.substr(0, k).c_str());
                if (val > 255 || k != std::to_string(val).size()) continue;
                helper(s.substr(k), n + 1, out + s.substr(0, k) + (n == 3 ? "" : "."), res);
            }
        }
    }
};
```

Java 解法二：

```
public class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<String> res = new ArrayList<String>();
        helper(s, 0, "", res);
        return res;
```

```
    }
    public void helper(String s, int n, String out, List<String> res) {
        if (n == 4) {
            if (s.isEmpty()) res.add(out);
            return;
        }
        for (int k = 1; k < 4; ++k) {
            if (s.length() < k) break;
            int val = Integer.parseInt(s.substring(0, k));
            if (val > 255 || k != String.valueOf(val).length()) continue;
            helper(s.substring(k), n + 1, out + s.substring(0, k) + (n == 3 ? "" : "."), res);
        }
    }
}
```

由于每段数字最多只能有三位，而且只能分为四段，所以情况并不是很多，我们可以使用暴力搜索的方法，每一段都循环1到3，然后当4段位数之和等于原字符串长度时，我们进一步判断每段数字是否不大于255，然后滤去不合要求的数字，加入结果中即可，参见代码如下：

C++ 解法三：

```
class Solution {
public:
    vector<string> restoreIpAddresses(string s) {
        vector<string> res;
        for (int a = 1; a < 4; ++a)
        for (int b = 1; b < 4; ++b)
        for (int c = 1; c < 4; ++c)
        for (int d = 1; d < 4; ++d)
            if (a + b + c + d == s.size()) {
                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) {
                    string t = to_string(A) + "." + to_string(B) + "." + to_string(C) + "." + to_string(D);
                    if (t.size() == s.size() + 3) res.push_back(t);
                }
            }
        return res;
    }
};
```

Java 解法三：

```
public class Solution {
    public List<String> restoreIpAddresses(String s) {
```

```
        List<String> res = new ArrayList<String>();
        for (int a = 1; a < 4; ++a)
        for (int b = 1; b < 4; ++b)
        for (int c = 1; c < 4; ++c)
        for (int d = 1; d < 4; ++d)
            if (a + b + c + d == s.length()) {
                int A = Integer.parseInt(s.substring(0, a));
                int B = Integer.parseInt(s.substring(a, a + b));
                int C = Integer.parseInt(s.substring(a + b, a + b + c));
                int D = Integer.parseInt(s.substring(a + b + c));
                if (A <= 255 && B <= 255 && C <= 255 && D <= 255) {
                    String t = String.valueOf(A) + "." + String.valueOf(B)
+ "." + String.valueOf(C) + "." + String.valueOf(D);
                    if (t.length() == s.length() + 3) res.add(t);
                }
            }
        return res;
    }
}
```

95. Unique Binary Search Trees II

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
```

这道题是之前的 Unique Binary Search Trees 的延伸，之前那个只要求算出所有不同的二叉搜索树的个数，这道题让把那些二叉树都建立出来。这种建树问题一般来说都是用递归来解，这道题也不例外，划分左右子树，递归构造。这个其实是用到了大名鼎鼎的分治法 Divide and Conquer，类似的题目还有之前的那道 Different Ways to Add Parentheses 用的方法一样，用递归来解，划分左右两个子数组，递归构造。刚开始时，将区间 [1, n] 当作一个整体，然后需要将其中的每个数字都当作根结点，其划分开了左右两个子区间，然后分别调用递归函数，会得到两个结点数组，接下来要做的就是从这两个数组中每次各取一个结点，当作当前根结点的左右子结点，然后将根结点加入结果 res 数组中即可，参见代码如下：

解法一：

```
class Solution {
public:
    vector<TreeNode*> generateTrees(int n) {
        if (n == 0) return {};
        return helper(1, n);
    }
    vector<TreeNode*> helper(int start, int end) {
        if (start > end) return {nullptr};
        vector<TreeNode*> res;
        for (int i = start; i <= end; ++i) {
            auto left = helper(start, i - 1), right = helper(i + 1, end);
            for (auto a : left) {
                for (auto b : right) {
                    TreeNode *node = new TreeNode(i);
                    node->left = a;
                    node->right = b;
                    res.push_back(node);
                }
            }
        }
        return res;
    }
};
```

我们可以使用记忆数组来优化，保存计算过的中间结果，从而避免重复计算。注意这道题的标签有一个是动态规划 Dynamic Programming，其实带记忆数组的递归形式就是 DP 的一种，memo[i][j] 表示在区间 [i, j] 范围内可以生成的所有 BST 的根结点，所以 memo 必须是一个三维数组，这样在递归函数中，就可以去 memo 中查找当前的区间是否已经计算过了，是的话，直接返回 memo 中的数组，否则就按之前的方法去计算，最后计算好了之后要更新 memo 数组，参见代码如下：

解法二：

```
class Solution {
public:
    vector<TreeNode*> generateTrees(int n) {
        if (n == 0) return {};
        vector<vector<vector<TreeNode*>>> memo(n,
vector<vector<TreeNode*>>(n));
```

```
        return helper(1, n, memo);
    }
    vector<TreeNode*> helper(int start, int end, vector<vector<vector<TreeNode*>>>& memo) {
        if (start > end) return {nullptr};
        if (!memo[start - 1][end - 1].empty()) return memo[start - 1][end - 1];
        vector<TreeNode*> res;
        for (int i = start; i <= end; ++i) {
            auto left = helper(start, i - 1, memo), right = helper(i + 1, end, memo);
            for (auto a : left) {
                for (auto b : right) {
                    TreeNode *node = new TreeNode(i);
                    node->left = a;
                    node->right = b;
                    res.push_back(node);
                }
            }
        }
        return memo[start - 1][end - 1] = res;
    }
};
```

96. Unique Binary Search Trees

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
```

这道题实际上是 卡塔兰数 Catalan Numbe 的一个例子，如果对卡塔兰数不熟悉的童鞋可能真不太好做。话说其实我也是今天才知道的好嘛 -.-|||，为啥我以前都不知道捏？！为啥卡塔兰数不像斐波那契数那样人尽皆知呢，是我太孤陋寡闻么？！不过今天知道也不晚，不断的学习新的东西，这才是刷题的意义所在嘛! 好了，废话不多说了，赶紧回到题目上来吧。我们先来看当 n = 1 的情况，只能形成唯一的一棵二叉搜索树，n分别为 1,2,3 的情况如下所示：

```
    1                       n = 1

    2         1             n = 2
   /           \
  1             2

  1     3     3     2     1       n = 3
   \   /     /     / \     \
    3 2     1     1   3     2
   / /       \             \
  2 1         2             3
```

就跟斐波那契数列一样，我们把 n = 0 时赋为1，因为空树也算一种二叉搜索树，那么 n = 1 时的情况可以看做是其左子树个数乘以右子树的个数，左右子树都是空树，所以1乘1还是1。那么 n = 2 时，由于1和2都可以为根，分别算出来，再把它们加起来即可。n = 2 的情况可由下面式子算出（这里的 dp[i] 表示当有i个数字能组成的 BST 的个数）：

dp[2] = dp[0] * dp[1] (1为根的情况，则左子树一定不存在，右子树可以有一个数字)

+ dp[1] * dp[0] (2为根的情况，则左子树可以有一个数字，右子树一定不存在)

同理可写出 n = 3 的计算方法：

dp[3] = dp[0] * dp[2] (1为根的情况，则左子树一定不存在，右子树可以有两个数字)

+ dp[1] * dp[1] (2为根的情况，则左右子树都可以各有一个数字)

+ dp[2] * dp[0] (3为根的情况，则左子树可以有两个数字，右子树一定不存在)

由此可以得出卡塔兰数列的递推式为：

我们根据以上的分析，可以写出代码如下：

解法一：

```
class Solution {
public:
    int numTrees(int n) {
        vector<int> dp(n + 1);
        dp[0] = dp[1] = 1;
        for (int i = 2; i <= n; ++i) {
            for (int j = 0; j < i; ++j) {
                dp[i] += dp[j] * dp[i - j - 1];
            }
        }
        return dp[n];
    }
};
```

由卡特兰数的递推式还可以推导出其通项公式，即 C(2n,n)/(n+1)，表示在 2n 个数字中任取n个数的方法再除以 n+1，只要你还没有忘记高中的排列组合的知识，就不难写出下面的代码，注意在相乘的时候为了防止整型数溢出，要将结果 res 定义为长整型，参见代码如下：

解法二：

```
class Solution {
public:
    int numTrees(int n) {
        long res = 1;
        for (int i = n + 1; i <= 2 * n; ++i) {
            res = res * i / (i - n);
        }
        return res / (n + 1);
    }
};
```

97. Interleaving String

Given *s1* , *s2* , *s3* , find whether *s3* is formed by the interleaving of *s1* and *s2*.

Example 1:

```
Input: s1 = "aabcc", s2 = "dbbca", _s3_ = "aadbbcbcac"
Output: true
```

Example 2:

```
Input: s1 = "aabcc", s2 = "dbbca", _s3_ = "aadbbbaccc"
Output: false
```

这道求交织相错的字符串和之前那道 Word Break 的题很类似，就像我之前说的只要是遇到字符串的子序列或是匹配问题直接就上动态规划 Dynamic Programming，其他的都不要考虑，什么递归呀的都是浮云（当然带记忆数组的递归写法除外，因为这也可以算是 DP 的一种），千辛万苦的写了递归结果拿到 OJ 上妥妥 Time Limit Exceeded，能把人气昏了，所以还是直接就考虑 DP 解法省事些。一般来说字符串匹配问题都是更新一个二维 dp 数组，核心就在于找出状态转移方程。那么我们还是从题目中给的例子出发吧，手动写出二维数组 dp 如下：

```
  Ø d b b c a
Ø T F F F F F
a T F F F F F
a T T T T T F
b F T T F T F
c F F T T T T
c F F F T F T
```

首先，这道题的大前提是字符串 s1 和 s2 的长度和必须等于 s3 的长度，如果不等于，肯定返回 false。那么当 s1 和 s2 是空串的时候，s3 必然是空串，则返回 true。所以直接给 dp[0][0] 赋值 true，然后若 s1 和 s2 其中的一个为空串的话，那么另一个肯定和 s3 的长度相等，则按位比较，若相同且上一个位置为 True，赋 True，其余情况都赋 False，这样的二维数组 dp 的边缘就初始化好了。下面只需要找出状态转移方程来更新整个数组即可，我们发现，在任意非边缘位置 dp[i][j] 时，它的左边或上边有可能为 True 或是 False，两边都可以更新过来，只要有一条路通着，那么这个点就可以为 True。那么我们得分别来看，如果左边的为 True，那么我们去除当前对应的 s2 中的字符串 s2[j - 1] 和 s3 中对应的位置的字符相比（计算对应位置时还要考虑已匹配的s1中的字符），为 s3[j - 1 + i], 如果相等，则赋 True，反之赋 False。 而上边为 True 的情况也类似，所以可以求出状态转移方程为：

dp[i][j] = (dp[i - 1][j] && s1[i - 1] == s3[i - 1 + j]) || (dp[i][j - 1] && s2[j - 1] == s3[j - 1 + i]);

其中 dp[i][j] 表示的是 s2 的前 i 个字符和 s1 的前 j 个字符是否匹配 s3 的前 i+j 个字符，根据以上分析，可写出代码如下：

解法一：

```
class Solution {
public:
    bool isInterleave(string s1, string s2, string s3) {
        if (s1.size() + s2.size() != s3.size()) return false;
        int n1 = s1.size(), n2 = s2.size();
        vector<vector<bool>> dp(n1 + 1, vector<bool> (n2 + 1));
        dp[0][0] = true;
        for (int i = 1; i <= n1; ++i) {
            dp[i][0] = dp[i - 1][0] && (s1[i - 1] == s3[i - 1]);
        }
        for (int i = 1; i <= n2; ++i) {
            dp[0][i] = dp[0][i - 1] && (s2[i - 1] == s3[i - 1]);
        }
        for (int i = 1; i <= n1; ++i) {
            for (int j = 1; j <= n2; ++j) {
                dp[i][j] = (dp[i - 1][j] && s1[i - 1] == s3[i - 1 + j]) ||
(dp[i][j - 1] && s2[j - 1] == s3[j - 1 + i]);
            }
        }
        return dp[n1][n2];
    }
};
```

我们也可以把for循环合并到一起，用if条件来处理边界情况，整体思路和上面的解法没有太大的区别，参见代码如下：

解法二：

```
class Solution {
public:
    bool isInterleave(string s1, string s2, string s3) {
        if (s1.size() + s2.size() != s3.size()) return false;
        int n1 = s1.size(), n2 = s2.size();
```

```
        vector<vector<bool>> dp(n1 + 1, vector<bool> (n2 + 1, false));
        for (int i = 0; i <= n1; ++i) {
            for (int j = 0; j <= n2; ++j) {
                if (i == 0 && j == 0) {
                    dp[i][j] = true;
                } else if (i == 0) {
                    dp[i][j] = dp[i][j - 1] && s2[j - 1] == s3[i + j - 1];
                } else if (j == 0) {
                    dp[i][j] = dp[i - 1][j] && s1[i - 1] == s3[i + j - 1];
                } else {
                    dp[i][j] = (dp[i - 1][j] && s1[i - 1] == s3[i + j - 1]) || (dp[i][j - 1] && s2[j - 1] == s3[i + j - 1]);
                }
            }
        }
        return dp[n1][n2];
    }
};
```

这道题也可以使用带优化的 DFS 来做，我们使用一个 HashSet，用来保存匹配失败的情况，我们分别用变量i，j，和k来记录字符串 s1，s2，和 s3 匹配到的位置，初始化的时候都传入0。在递归函数中，首先根据i和j，算出 key 值，由于我们的 HashSet 中只能放一个数字，而我们要 encode 两个数字i和j，所以通过用i乘以 s3 的长度再加上j来得到 key，此时我们看，如果 key 已经在集合中，直接返回 false，因为集合中存的是无法匹配的情况。然后先来处理 corner case 的情况，如果i等于 s1 的长度了，说明 s1 的字符都匹配完了，此时 s2 剩下的字符和 s3 剩下的字符可以直接进行匹配了，所以我们直接返回两者是否能匹配的 bool 值。同理，如果j等于 s2 的长度了，说明 s2 的字符都匹配完了，此时 s1 剩下的字符和 s3 剩下的字符可以直接进行匹配了，所以我们直接返回两者是否能匹配的 bool 值。如果 s1 和 s2 都有剩余字符，那么当 s1 的当前字符等于 s3 的当前字符，那么调用递归函数，注意i和k都加上1，如果递归函数返回 true，则当前函数也返回 true；还有一种情况是，当 s2 的当前字符等于 s3 的当前字符，那么调用递归函数，注意j和k都加上1，如果递归函数返回 true，那么当前函数也返回 true。如果匹配失败了，则将 key 加入集合中，并返回 false 即可，参见代码如下：

解法三：

```
class Solution {
public:
    bool isInterleave(string s1, string s2, string s3) {
        if (s1.size() + s2.size() != s3.size()) return false;
        unordered_set<int> s;
        return helper(s1, 0, s2, 0, s3, 0, s);
    }
    bool helper(string& s1, int i, string& s2, int j, string& s3, int k, unordered_set<int>& s) {
        int key = i * s3.size() + j;
        if (s.count(key)) return false;
        if (i == s1.size()) return s2.substr(j) == s3.substr(k);
        if (j == s2.size()) return s1.substr(i) == s3.substr(k);
        if ((s1[i] == s3[k] && helper(s1, i + 1, s2, j, s3, k + 1, s)) ||
```

```
            (s2[j] == s3[k] && helper(s1, i, s2, j + 1, s3, k + 1, s)))
return true;
        s.insert(key);
        return false;
    }
};
```

既然 DFS 可以，那么 BFS 也就坐不住了，也要出来浪一波。这里我们需要用队列 queue 来辅助运算，如果将解法一讲解中的那个二维 dp 数组列出来的 TF 图当作一个迷宫的话，那么 BFS 的目的就是要从 (0, 0) 位置找一条都是T的路径通到 (n1, n2) 位置，这里我们还要使用 HashSet，不过此时保存到是已经遍历过的位置，队列中还是存 key 值，key 值的 encode 方法跟上面 DFS 解法的相同，初始时放个0进去。然后我们进行 while 循环，循环条件除了q不为空，还有一个是k小于 n3，因为匹配完 s3 中所有的字符就结束了。然后由于是一层层的遍历，所以要直接循环 queue 中元素个数的次数，在 for 循环中，对队首元素进行解码，得到i和j值，如果i小于 n1，说明 s1 还有剩余字符，如果 s1 当前字符等于 s3 当前字符，那么把 s1 的下一个位置 i+1 跟j一起加码算出 key 值，如果该 key 值不在于集合中，则加入集合，同时加入队列 queue 中；同理，如果j小于 n2，说明 s2 还有剩余字符，如果 s2 当前字符等于 s3 当前字符，那么把 s2 的下一个位置 j+1 跟i一起加码算出 key 值，如果该 key 值不在于集合中，则加入集合，同时加入队列 queue 中。for 循环结束后，k自增1。最后如果匹配成功的话，那么 queue 中应该只有一个 (n1, n2) 的 key 值，且k此时等于 n3，所以当 queue 为空或者k不等于 n3 的时候都要返回 false，参见代码如下：

解法四：

```
class Solution {
public:
    bool isInterleave(string s1, string s2, string s3) {
        if (s1.size() + s2.size() != s3.size()) return false;
        int n1 = s1.size(), n2 = s2.size(), n3 = s3.size(), k = 0;
        unordered_set<int> s;
        queue<int> q{{0}};
        while (!q.empty() && k < n3) {
            int len = q.size();
            for (int t = 0; t < len; ++t) {
                int i = q.front() / n3, j = q.front() % n3; q.pop();
                if (i < n1 && s1[i] == s3[k]) {
                    int key = (i + 1) * n3 + j;
                    if (!s.count(key)) {
                        s.insert(key);
                        q.push(key);
                    }
                }
                if (j < n2 && s2[j] == s3[k]) {
                    int key = i * n3 + j + 1;
                    if (!s.count(key)) {
                        s.insert(key);
                        q.push(key);
                    }
                }
            }
```

```
            ++k;
        }
        return !q.empty() && k == n3;
    }
};
```

98. Validate Binary Search Tree

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:

```
Input:
    2
   / \
  1   3
Output: true
```

Example 2:

```
    5
   / \
  1   4
     / \
    3   6
Output: false
Explanation: The input is: [5,1,4,null,null,3,6]. The root node's value
             is 5 but its right child's value is 4.
```

这道验证二叉搜索树有很多种解法，可以利用它本身的性质来做，即左<根<右，也可以通过利用中序遍历结果为有序数列来做，下面我们先来看最简单的一种，就是利用其本身性质来做，初始化时带入系统最大值和最小值，在递归过程中换成它们自己的节点值，用long代替int就是为了包括int的边界条件，代码如下：

C++ 解法一：

```
// Recursion without inorder traversal
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return isValidBST(root, LONG_MIN, LONG_MAX);
    }
    bool isValidBST(TreeNode* root, long mn, long mx) {
        if (!root) return true;
        if (root->val <= mn || root->val >= mx) return false;
        return isValidBST(root->left, mn, root->val) && isValidBST(root->right, root->val, mx);
    }
};
```

Java 解法一：

```
public class Solution {
    public boolean isValidBST(TreeNode root) {
        if (root == null) return true;
        return valid(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }
    public boolean valid(TreeNode root, long low, long high) {
        if (root == null) return true;
        if (root.val <= low || root.val >= high) return false;
        return valid(root.left, low, root.val) && valid(root.right, root.val, high);
    }
}
```

这题实际上简化了难度，因为有的时候题目中的二叉搜索树会定义为左<=根<右，而这道题设定为一般情况左<根<右，那么就可以用中序遍历来做。因为如果不去掉左=根这个条件的话，那么下边两个数用中序遍历无法区分：

```
 20   20
 /    \
20    20
```

它们的中序遍历结果都一样，但是左边的是 BST，右边的不是 BST。去掉等号的条件则相当于去掉了这种限制条件。下面来看使用中序遍历来做，这种方法思路很直接，通过中序遍历将所有的节点值存到一个数组里，然后再来判断这个数组是不是有序的，代码如下：

C++ 解法二：

```
// Recursion
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        if (!root) return true;
        vector<int> vals;
```

```
        inorder(root, vals);
        for (int i = 0; i < vals.size() - 1; ++i) {
            if (vals[i] >= vals[i + 1]) return false;
        }
        return true;
    }
    void inorder(TreeNode* root, vector<int>& vals) {
        if (!root) return;
        inorder(root->left, vals);
        vals.push_back(root->val);
        inorder(root->right, vals);
    }
};
```

Java 解法二：

```
public class Solution {
    public boolean isValidBST(TreeNode root) {
        List<Integer> list = new ArrayList<Integer>();
        inorder(root, list);
        for (int i = 0; i < list.size() - 1; ++i) {
            if (list.get(i) >= list.get(i + 1)) return false;
        }
        return true;
    }
    public void inorder(TreeNode node, List<Integer> list) {
        if (node == null) return;
        inorder(node.left, list);
        list.add(node.val);
        inorder(node.right, list);
    }
}
```

下面这种解法跟上面那个很类似，都是用递归的中序遍历，但不同之处是不将遍历结果存入一个数组遍历完成再比较，而是每当遍历到一个新节点时和其上一个节点比较，如果不大于上一个节点那么则返回 false，全部遍历完成后返回 true。代码如下：

C++ 解法三：

```
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        TreeNode *pre = NULL;
        return inorder(root, pre);
    }
    bool inorder(TreeNode* node, TreeNode*& pre) {
        if (!node) return true;
        bool res = inorder(node->left, pre);
        if (!res) return false;
```

```
        if (pre) {
            if (node->val <= pre->val) return false;
        }
        pre = node;
        return inorder(node->right, pre);
    }
};
```

当然这道题也可以用非递归来做，需要用到栈，因为中序遍历可以非递归来实现，所以只要在其上面稍加改动便可，代码如下：

C++ 解法四：

```
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        stack<TreeNode*> s;
        TreeNode *p = root, *pre = NULL;
        while (p || !s.empty()) {
            while (p) {
                s.push(p);
                p = p->left;
            }
            p = s.top(); s.pop();
            if (pre && p->val <= pre->val) return false;
            pre = p;
            p = p->right;
        }
        return true;
    }
};
```

Java 解法四：

```
public class Solution {
    public boolean isValidBST(TreeNode root) {
        Stack<TreeNode> s = new Stack<TreeNode>();
        TreeNode p = root, pre = null;
        while (p != null || !s.empty()) {
            while (p != null) {
                s.push(p);
                p = p.left;
            }
            p = s.pop();
            if (pre != null && p.val <= pre.val) return false;
            pre = p;
            p = p.right;
        }
        return true;
```

```
    }
}
```

最后还有一种方法，由于中序遍历还有非递归且无栈的实现方法，称之为 Morris 遍历，可以参考博主之前的博客 Binary Tree Inorder Traversal，这种实现方法虽然写起来比递归版本要复杂的多，但是好处在于是 O(1) 空间复杂度，参见代码如下：

C++ 解法五:

```
class Solution {
public:
    bool isValidBST(TreeNode *root) {
        if (!root) return true;
        TreeNode *cur = root, *pre, *parent = NULL;
        bool res = true;
        while (cur) {
            if (!cur->left) {
                if (parent && parent->val >= cur->val) res = false;
                parent = cur;
                cur = cur->right;
            } else {
                pre = cur->left;
                while (pre->right && pre->right != cur) pre = pre->right;
                if (!pre->right) {
                    pre->right = cur;
                    cur = cur->left;
                } else {
                    pre->right = NULL;
                    if (parent->val >= cur->val) res = false;
                    parent = cur;
                    cur = cur->right;
                }
            }
        }
        return res;
    }
};
```

99. Recover Binary Search Tree

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?

这道题要求我们复原一个二叉搜索树，说是其中有两个的顺序被调换了，题目要求上说 O(n) 的解法很直观，这种解法需要用到递归，用中序遍历树，并将所有节点存到一个一维向量中，把所有节点值存到另一个一维向量中，然后对存节点值的一维向量排序，在将排好的数组按顺序赋给节点。这种最一般的解法可针对任意个数目的节点错乱的情况，这里先贴上此种解法：

解法一：

```
// O(n) space complexity
class Solution {
public:
    void recoverTree(TreeNode* root) {
        vector<TreeNode*> list;
```

```
        vector<int> vals;
        inorder(root, list, vals);
        sort(vals.begin(), vals.end());
        for (int i = 0; i < list.size(); ++i) {
            list[i]->val = vals[i];
        }
    }
    void inorder(TreeNode* root, vector<TreeNode*>& list, vector<int>&
vals) {
        if (!root) return;
        inorder(root->left, list, vals);
        list.push_back(root);
        vals.push_back(root->val);
        inorder(root->right, list, vals);
    }
};
```

然后博主上网搜了许多其他解法，看到另一种是用双指针来代替一维向量的，但是这种方法用到了递归，也不是 O(1) 空间复杂度的解法，这里需要三个指针，first，second 分别表示第一个和第二个错乱位置的节点，pre 指向当前节点的中序遍历的前一个节点。这里用传统的中序遍历递归来做，不过再应该输出节点值的地方，换成了判断 pre 和当前节点值的大小，如果 pre 的大，若 first 为空，则将 first 指向 pre 指的节点，把 second 指向当前节点。这样中序遍历完整个树，若 first 和 second 都存在，则交换它们的节点值即可。这个算法的空间复杂度仍为 O(n)，n为树的高度，参见代码如下：

解法二：

```
// Still O(n) space complexity
class Solution {
public:
    TreeNode *pre = NULL, *first = NULL, *second = NULL;
    void recoverTree(TreeNode* root) {
        inorder(root);
        swap(first->val, second->val);
    }
    void inorder(TreeNode* root) {
        if (!root) return;
        inorder(root->left);
        if (!pre) pre = root;
        else {
            if (pre->val > root->val) {
                if (!first) first = pre;
                second = root;
            }
            pre = root;
        }
        inorder(root->right);
    }
};
```

我们其实也可以使用迭代的写法，因为中序遍历 Binary Tree Inorder Traversal 也可以借助栈来实现，原理还是跟前面的相同，记录前一个结点，并和当前结点相比，如果前一个结点值大，那么更新 first 和 second，最后交换 first 和 second 的结点值即可，参见代码如下：

解法三：

```
// Always O(n) space complexity
class Solution {
public:
    void recoverTree(TreeNode* root) {
        TreeNode *pre = NULL, *first = NULL, *second = NULL, *p = root;
        stack<TreeNode*> st;
        while (p || !st.empty()) {
            while (p) {
                st.push(p);
                p = p->left;
            }
            p = st.top(); st.pop();
            if (pre) {
                if (pre->val > p->val) {
                    if (!first) first = pre;
                    second = p;
                }
            }
            pre = p;
            p = p->right;
        }
        swap(first->val, second->val);
    }
};
```

这道题的真正符合要求的解法应该用的 Morris 遍历，这是一种非递归且不使用栈，空间复杂度为 O(1) 的遍历方法，可参见博主之前的博客 Binary Tree Inorder Traversal，在其基础上做些修改，加入 first, second 和 parent 指针，来比较当前节点值和中序遍历的前一节点值的大小，跟上面递归算法的思路相似，代码如下：

解法四：

```
// Now O(1) space complexity
class Solution {
public:
    void recoverTree(TreeNode* root) {
        TreeNode *first = nullptr, *second = nullptr, *cur = root, *pre =
nullptr ;
        while (cur) {
            if (cur->left){
                TreeNode *p = cur->left;
                while (p->right && p->right != cur) p = p->right;
                if (!p->right) {
```

```
                    p->right = cur;
                    cur = cur->left;
                    continue;
                } else {
                    p->right = NULL;
                }
            }
            if (pre && cur->val < pre->val){
              if (!first) first = pre;
              second = cur;
            }
            pre = cur;
            cur = cur->right;
        }
        swap(first->val, second->val);
    }
};
```