418. 屏幕可显示句子的数量（中等）

1. 题目描述

   给你一个 rows x cols 的屏幕和一个用 非空 的单词列表组成的句子，请你计算出给定句子可以在屏幕上完整显示的次数。

   注意：

   1. 一个单词不能拆分成两行。
   2. 单词在句子中的顺序必须保持不变。
   3. 在一行中 的两个连续单词必须用一个空格符分隔。
   4. 句子中的单词总量不会超过 100。
   5. 每个单词的长度大于 0 且不会超过 10。
   6. 1 ≤ rows, cols ≤ 20,000.

```
示例 1:
输入:
rows = 2, cols = 8, 句子 sentence = ["hello", "world"]
输出:
1
解释:
hello---
world---
字符 '-' 表示屏幕上的一个空白位置。

示例 2:
输入:
rows = 3, cols = 6, 句子 sentence = ["a", "bcd", "e"]
输出:
2
解释:
a-bcd-
e-a---
bcd-e-
字符 '-' 表示屏幕上的一个空白位置。

示例 3:
输入:
rows = 4, cols = 5, 句子 sentence = ["I", "had", "apple", "pie"]
输出:
1
解释:
I-had
apple
pie-I
had--
字符 '-' 表示屏幕上的一个空白位置。
```

2. 正确解法——动态规划

动态规划解法 比较奇葩 关键是设计memory结构 抓住本题词数和词长都很小但rows cols很长的特点 见代码注释

时间复杂度为O(N) N与rows-cols同数量级

```
class Solution {
public:
    int wordsTyping(vector<string>& sentence, int rows, int cols) {
    // 从第 i 个词开始 这一行能放下几遍句子
    vector<int> dp(sentence.size(), 0);
    // 从第 i 个词开始 放下dp[i]遍句子后 变为第几个词
    vector<int> next(sentence.size(), 0);

    for (int i = 0; i < sentence.size(); ++i) {
        int count = 0;
        int ptr = i;
        int cur = cols;
        while (cur >= (int)sentence[ptr].size()) {
        cur -= sentence[ptr].size() + 1;
        ++ptr;
        if (ptr == sentence.size()) {
            ++count;
            ptr = 0;
        }
        }
        dp[i] = count;
        next[i] = ptr;
    }

    int count = 0;
    int cur = 0;
    for (int i = 0; i < rows; ++i) {
        count += dp[cur];
        cur = next[cur];
    }
    return count;
    }
};
```

85. 最大矩形（困难）

1. 题目描述

   给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

```
示例:
输入:
[
  ["1","0","1","0","0"],
  ["1","0","1","1","1"],
  ["1","1","1","1","1"],
  ["1","0","0","1","0"]
]
输出: 6
```

2. 简单实现

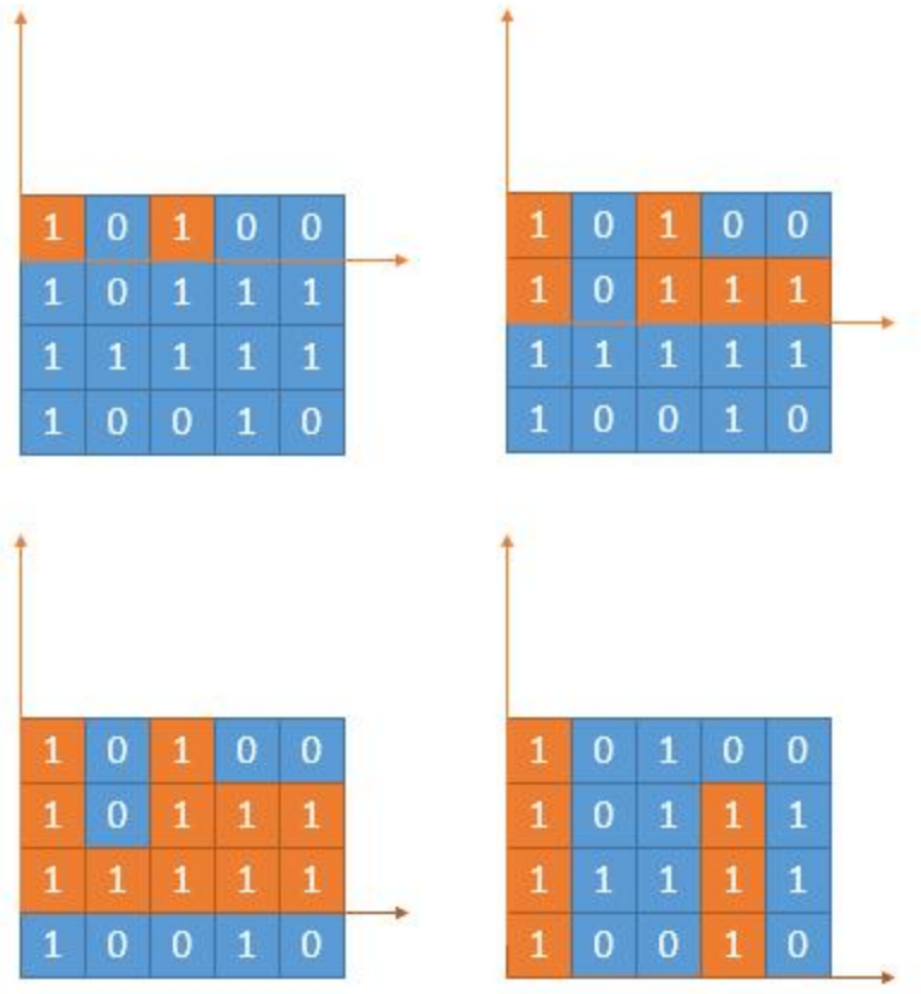
```
class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        int m = matrix.size();
        if(m <= 0) return 0;
        int n = matrix[0].size();
        if(n <= 0 ) return 0;
        vector<vector<vector<int>>> dp(m, vector<vector<int>>(n, vector<int>
(2)));//行、列
        int ans = 0;
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                if(matrix[i][j] == '0')
                    dp[i][j][0] = dp[i][j][1] = 0;
                else{
                    if(j == 0) dp[i][j][0] = 1;
                    else dp[i][j][0] = dp[i][j-1][0]+1;
                    if(i == 0) dp[i][j][1] = 1;
                    else dp[i][j][1] = dp[i-1][j][1]+1;
                    if(i == 0) ans = max(ans, dp[i][j][0]);
                    else if(j == 0) ans = max(ans, dp[i][j][1]);
                    if(i != 0 && j != 0){
                        int c = i-1;
                        int w = dp[i][j][0];
                        ans = max(ans, w*(i-c));
                        while(c>=0 && matrix[c][j]=='1'){//从下往上一条条遍历
                            w = min(w, dp[c][j][0]);
                            c--;
                            ans = max(ans, w*(i-c));
                        }
                    }
                }
            }
        }
        return ans;
    }
};
```

3. 最优解法

参考84.柱状图中最大的矩形，单调栈解法：

https://leetcode-cn.com/problems/largest-rectangle-in-histogram/solution/bao-li-jie-fa-zhan-by-liweiwei1419/

再想一下这个题，看下边的橙色的部分，这完全就是上一道题呀！




```
class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        if (matrix.size() == 0) return 0;
        int width = matrix[0].size(), height = matrix.size();
        vector<int>leftVec(width, 0), rightVec(width, width), heightVec(width, 0);
        int maxArea=0;
        for (int i = 0; i < height; i++) {
            int curRight = width, curLeft = 0;
            //right
            for (int j = width-1; j >= 0; j--){
                if (matrix[i][j] == '1')
                    rightVec[j] = min(rightVec[j], curRight);//至少与j等高的右界
                else {
                    rightVec[j] = width;//更新为width，之后再出现1时才能重新记录
                    curRight = j;
```

```
                }
            }
            //left-height-area
            for (int j = 0; j < width; j++) {
                if (matrix[i][j] == '1'){
                    leftVec[j] = max(leftVec[j],curLeft);
                    heightVec[j]++;
                }
                else {
                    leftVec[j] = 0;
                    curLeft = j+1;
                    heightVec[j] = 0;
                }
                maxArea = max(maxArea,(rightVec[j]-leftVec[j])*heightVec[j]);
            }
        }
        return maxArea;
    }
};
```

939. 最小面积矩形（中等）

1. 题目描述

   给定在 xy 平面上的一组点，确定由这些点组成的矩形的最小面积，其中矩形的边平行于 x 轴和 y 轴。

   如果没有任何矩形，就返回 0。

```
示例 1:
输入: [[1,1],[1,3],[3,1],[3,3],[2,2]]
输出: 4

示例 2:
输入: [[1,1],[1,3],[3,1],[3,3],[4,1],[4,3]]
输出: 2
```

提示:

- 1 <= points.length <= 500
- 0 <= points[i][0] <= 40000
- 0 <= points[i][1] <= 40000
- 所有的点都是不同的。

2. 简单实现

   直接用两个map表示每行/列上的点的列/行号，然后依次找以每个点为左上角点能构成的矩形

```
class Solution {
public:
    int minAreaRect(vector<vector<int>>& points) {
        int size = points.size();
        if(size < 4) return 0;
        map<int, set<int>> rc;
```

```
        map<int, set<int>> cr;
        for(int i = 0; i < size; i++){
            rc[points[i][0]].insert(points[i][1]);
            cr[points[i][1]].insert(points[i][0]);
        }
        int ans = INT_MAX;
        for(int i = 0; i < size; i++){//某点i作为左上角点
            if(rc[points[i][0]].size() > 1 && cr[points[i][1]].size() > 1){//存在同
行、同列点
                auto it = rc[points[i][0]].find(points[i][1]);
                if(it == rc[points[i][0]].end()) continue;
                it++;
                while(it != rc[points[i][0]].end() && (*it) - points[i][1] < ans)
{//右上角点2
                    int w = (*it) - points[i][1];
                    auto it2 = cr[points[i][1]].find(points[i][0]);
                    if(it2 == cr[points[i][1]].end()) continue;
                    it2++;
                    while(it2 != cr[points[i][1]].end() && ((*it2)-points[i][0])*w
< ans){//左下角点3
                        if(rc[*it2].find(*it) != rc[*it2].end()){
                            //存在能与点123构成矩形的点
                            ans = ((*it2)-points[i][0]) * w;
                        }
                        it2++;
                    }
                    it++;
                }
            }
        }
        if(ans == INT_MAX)
            ans = 0;
        return ans;
    }
};
```

3. 对角线法

枚举每两个点为对角线的情况，思路不错，效果没我的好，我的有优化

```
class Solution {
public:
    int minAreaRect(vector<vector<int>>& points) {
        int size = points.size();
        if(size < 4) return 0;
        unordered_set<int> pointSet;
        for (int i = 0; i < size; i++)
            pointSet.insert(40001 * points[i][0] + points[i][1]);

        int ans = INT_MAX;
        for (int i = 0; i < size; ++i)
            for (int j = i+1; j < size; ++j) {
                if (points[i][0] != points[j][0] && points[i][1] != points[j][1]) {
```

```
                    if (pointSet.find(40001 * points[i][0] + points[j][1]) !=
pointSet.end() && pointSet.find(40001 * points[j][0] + points[i][1]) !=
pointSet.end()) {
                        ans = min(ans, abs((points[j][0] - points[i][0]) *
                                           (points[j][1] - points[i][1]));
                    }
                }
            }

        return ans < INT_MAX ? ans : 0;
    }
};
```

49. 字母异位词分组（中等）

1. 题目描述

   给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

   ```
   示例:
   输入: ["eat", "tea", "tan", "ate", "nat", "bat"]
   输出:
   [
     ["ate","eat","tea"],
     ["nat","tan"],
     ["bat"]
   ]
   ```

   说明:

   - 所有输入均为小写字母。
   - 不考虑答案输出的顺序。

2. 简单实现

```
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> m;
        for(int i = 0; i < strs.size(); i++){
            string cur = strs[i];
            sort(cur.begin(), cur.end());
            m[cur].push_back(strs[i]);
        }
        vector<vector<string>> ans;
        for(auto it = m.begin(); it != m.end(); it++){
            ans.push_back(it->second);
        }
        return ans;
    }
};
```

3. 改进

   节省一些空间复杂度，注意代码中的sub

```
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        vector<vector<string>> res;
        int sub=0;  //结果vector的下标值
        string tmp; //临时string
        unordered_map<string, int> work; //判断排序后单词是否存在，即字母组成是否一致
        for(auto str:strs) {
            tmp=str;
            sort(tmp.begin(), tmp.end());
            if(work.count(tmp))
               res[work[tmp]].push_back(str);//直接按下标push到答案里
            else {
                vector<string> vec(1,str);
                res.push_back(vec);
                work[tmp]=sub++;
            }
        }
        return res;
    }
};
```

239. 滑动窗口最大值（困难）

1. 题目描述

   给定一个数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

   返回滑动窗口中的最大值。

   进阶：

   你能在线性时间复杂度内解决此题吗？

```
示例：
输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出: [3,3,5,5,6,7]
解释:
滑动窗口的位置                最大值
---------------------------------------
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

提示：

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4

2. 简单实现

   位类似于最小栈的思想，维持一个单调递减队列

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> ans;
        if(nums.size() == 0) return ans;
        deque<int> q;//队列内存储下标，从而判断窗口情况
        for(int i = 0; i < nums.size(); i++){
            while(!q.empty() && nums[q.back()] <= nums[i])
                q.pop_back();//在num[i]滑出窗口前，pop出的这些数在窗口内都不可能与num[i]争
取最大值
            q.push_back(i);
            if(i-q.front() > k-1)//队列头超出窗口范围
                q.pop_front();
            if(i >= k-1)
                ans.push_back(nums[q.front()]);
        }
        return ans;
    }
};
```

3. 动态规划法

   https://leetcode-cn.com/problems/sliding-window-maximum/solution/hua-dong-chuang-kou-zui-da-zhi-by-leetcode-3/

   十分巧妙的分块dp了，想出来的都是什么神人，服了服了

158. 用Read4读取N个字符II（困难）

1. 题目描述

   给你一个文件，并且该文件只能通过给定的 read4 方法来读取，请实现一个方法使其能够读取 n 个字符。注意：你的 read 方法可能会被调用多次。

   read4 的定义：

```
参数类型: char[] buf
返回类型: int
注意: buf[] 是目标缓存区不是源缓存区，read4 的返回结果将会复制到 buf[] 当中。
```

   下列是一些使用 read4 的例子：

```
File file("abcdefghijk"); // 文件名为 "abcdefghijk"， 初始文件指针 (fp) 指向 'a'
char[] buf = new char[4]; // 创建一个缓存区使其能容纳足够的字符
read4(buf); // read4 返回 4。现在 buf = "abcd", fp 指向 'e'
read4(buf); // read4 返回 4。现在 buf = "efgh", fp 指向 'i'
read4(buf); // read4 返回 3。现在 buf = "ijk", fp 指向文件末尾
```

read 方法：

通过使用 read4 方法，实现 read 方法。该方法可以从文件中读取 n 个字符并将其存储到缓存数组 buf 中。您 不能 直接操作文件。

返回值为实际读取的字符。

read 的定义：

```
参数:   char[] buf, int n
返回值: int
注意: buf[] 是目标缓存区不是源缓存区，你需要将结果写入 buf[] 中。
```

```
示例 1:
File file("abc");
Solution sol;
// 假定 buf 已经被分配了内存，并且有足够的空间来存储文件中的所有字符。
sol.read(buf, 1); // 当调用了您的 read 方法后，buf 需要包含 "a"。 一共读取 1 个字符，因此返回 1。
sol.read(buf, 2); // 现在 buf 需要包含 "bc"。一共读取 2 个字符，因此返回 2。
sol.read(buf, 1); // 由于已经到达了文件末尾，没有更多的字符可以读取，因此返回 0。

示例 2:
File file("abc");
Solution sol;
sol.read(buf, 4); // 当调用了您的 read 方法后，buf 需要包含 "abc"。 一共只能读取 3 个字符，因此返回 3。
sol.read(buf, 1); // 由于已经到达了文件末尾，没有更多的字符可以读取，因此返回 0。
```

注意：

1. 你 不能 直接操作该文件，文件只能通过 read4 获取而 不能 通过 read。
2. read 函数可以被调用 多次。
3. 请记得 重置 在 Solution 中声明的类变量（静态变量），因为类变量会 在多个测试用例中保持不变，影响判题准确。请 查阅 这里。
4. 你可以假定目标缓存数组 buf 保证有足够的空间存下 n 个字符。
5. 保证在一个给定测试用例中，read 函数使用的是同一个 buf。

2. 简单实现

熟悉一下char*，算法不难，一步一步来就好了

```
// Forward declaration of the read4 API.
int read4(char *buf);

class Solution {
public:
    /**
```

```
     * @param buf Destination buffer
     * @param n   Number of characters to read
     * @return    The number of actual characters read
     */
    char local_buf[4] = {0};//存储被read4出来但是还未被read出去的数据
    int l = 0;//local_buf有效数据左界
    int r = 0;//local_buf有效数据右界
    int read(char *buf, int n) {
        int cnt = 0;
        while (l < r && n > 0) {//先把local_buf中的读出来
            buf[cnt++] = local_buf[l++];
            --n;
        }
        if (n == 0)
            return  cnt;
        int k = n / 4;
        for (int i = 0; i < k; ++i) {//4个4个读
            l = 0;
            r = read4(local_buf);
            while (l < r && n > 0) {
                buf[cnt++] = local_buf[l++];
                --n;
            }
            if (r < 4)//文件尾了
                return cnt;
        }
        if (n == 0)
            return cnt;
        //还要读1~3个
        l = 0;
        r = read4(local_buf);
        while (l < r && n > 0) {
            buf[cnt++] = local_buf[l++];
            --n;
        }
        return cnt;
    }
};
```

329. 矩阵中的最长递增路径（困难）

1. 题目描述

   给定一个整数矩阵，找出最长递增路径的长度。

   对于每个单元格，你可以往上，下，左，右四个方向移动。 你不能在对角线方向上移动或移动到边界外（即不允许环绕）。

```
示例 1:
输入: nums =
[
  [9,9,4],
```

```
  [6,6,8],
  [2,1,1]
]
输出: 4
解释: 最长递增路径为 [1, 2, 6, 9]。

示例 2:
输入: nums =
[
  [3,4,5],
  [3,2,6],
  [2,2,1]
]
输出: 4
解释: 最长递增路径是 [3, 4, 5, 6]。注意不允许在对角线方向上移动。
```

2. 简单实现

带记忆化存储的dfs

```
class Solution {
public:
    int m,n;
    vector<vector<int>> path;//记录以各个位置为起点的最长递增路径
    vector<vector<int>> dirs = {{0,1}, {0,-1}, {1,0}, {-1,0}};
    int getPath(vector<vector<int>>& matrix, int x , int y){
        if(path[x][y] != -1)//存储了答案，无需遍历
            return path[x][y];
        int re = 1;
        for(int i = 0; i < 4; i++){//四个方向
            int xx = x + dirs[i][0];
            int yy = y + dirs[i][1];
            if(xx >= 0 && xx < m && yy >= 0 && yy < n && matrix[xx][yy] > matrix[x]
[y])
                re = max(re, getPath(matrix,xx,yy)+1);
        }
        path[x][y] = re;
        return re;
    }
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        m = matrix.size();
        if(m <= 0) return 0;
        n = matrix[0].size();
        path = vector<vector<int>>(m, vector<int>(n, -1));
        int ans = 0;
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                getPath(matrix, i, j);
                ans = max(ans, path[i][j]);
            }
        }
        return ans;
    }
```

```
};
```

3. 最优解法——”剥洋葱“动态规划

啊啊啊

直觉

每个细胞的结果只与相邻的结果相关，能否使用动态规划?

算法

如果我们定义从单元格 (i, j) 开始的最长递增路径为函数

$$f(i, j)$$

则可以写出状态转移函数

$$f(i,j) = max\{f(x,y) | (x,y) \text{ is a nei} * * * \text{or of} (i,j) \text{ and matrix}[x][y] > \text{matrix}[i][j]\} + 1$$

此公式与以前方法中使用的公式相同。有了状态转移函数，你可能会觉得可以使用动态规划来推导出所有结果，去他的深度优先搜索!

这听起来很美好，可惜你忽略了一件事：我们没有依赖列表。

想要让动态规划有效，如果问题 B 依赖于问题 A 的结果，就必须确保问题 A 比问题 B先计算。这样的依赖顺序对许多问题十分简单自然。如著名的斐波那契数列：

$$F(0) = 1, F(1) = 1, F(n) = F(n-1) + F(n-2)$$

子问题 $F(n)$ 依赖于 $F(n-1)$ 和 $F(n-2)$。因此，自然顺序就是正确的计算顺序。被依赖者总会先被计算。

这种依赖顺序的术语是“拓扑顺序”或“拓扑排序”：

> 对有向无环图的拓扑排序是顶点的一个线性排序，使得对于任何有向边 (u, v)，顶点 u 都在 顶点 v 的前面。

在本问题中，拓扑顺序并不简单自然。没有矩阵的值，我们无法知道两个邻居 A 和 B 的依赖关系。作为预处理，我们必须显式执行拓扑排序。之后，我们可以按照存储的拓扑顺序使用状态转移函数动态地解决问题。

有多种实现拓扑排序的方法。这里我们使用的是一种被称为“剥洋葱”的方法。其思路是在一个有向无环图中，会有一些不依赖于其他顶点的顶点，称为“叶子”。我们将这些叶子放在一个列表中（他们的内部排序不重要），然后将他们从图中移除。移除之后，会产生新的“叶子”。重复以上过程，就像一层一层一层地拨开洋葱的心。最后，列表中就会存储有效的拓扑排序。

在本问题中，因为我们想要求出在整个图中最长的路径，也就是“洋葱”的层总数。因此，我们可以在“剥离”的期间计算层数，在不调用动态规划的情况下返回计数。

```
// Topological Sort Based Solution
// An Alternative Solution
public class Solution {
```

```java
    private static final int[][] dir = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    private int m, n;
    public int longestIncreasingPath(int[][] grid) {
        int m = grid.length;
        if (m == 0) return 0;
        int n = grid[0].length;
        // padding the matrix with zero as boundaries
        // assuming all positive integer, otherwise use INT_MIN as boundaries
        int[][] matrix = new int[m + 2][n + 2];
        for (int i = 0; i < m; ++i)
            System.arraycopy(grid[i], 0, matrix[i + 1], 1, n);

        // calculate outdegrees
        int[][] outdegree = new int[m + 2][n + 2];
        for (int i = 1; i <= m; ++i)
            for (int j = 1; j <= n; ++j)
                for (int[] d: dir)
                    if (matrix[i][j] < matrix[i + d[0]][j + d[1]])
                        outdegree[i][j]++;

        // find leaves who have zero out degree as the initial level
        n += 2;
        m += 2;
        List<int[]> leaves = new ArrayList<>();
        for (int i = 1; i < m - 1; ++i)
            for (int j = 1; j < n - 1; ++j)
                if (outdegree[i][j] == 0) leaves.add(new int[]{i, j});

        // remove leaves level by level in topological order
        int height = 0;
        while (!leaves.isEmpty()) {
            height++;
            List<int[]> newLeaves = new ArrayList<>();
            for (int[] node : leaves) {
                for (int[] d:dir) {
                    int x = node[0] + d[0], y = node[1] + d[1];
                    if (matrix[node[0]][node[1]] > matrix[x][y])
                        if (--outdegree[x][y] == 0)
                            newLeaves.add(new int[]{x, y});
                }
            }
            leaves = newLeaves;
        }
        return height;
    }
}
```

809. 情感丰富的文字（中等）

1. 题目描述

有时候人们会用重复写一些字母来表示额外的感受，比如 "hello" -> "heeellooo", "hi" -> "hiii"。我们将相邻字母都相同的一串字符定义为相同字母组，例如："h", "eee", "ll", "ooo"。

对于一个给定的字符串 S ，如果另一个单词能够通过将一些字母组扩张从而使其和 S 相同，我们将这个单词定义为可扩张的（stretchy）。扩张操作定义如下：选择一个字母组（包含字母 c ），然后往其中添加相同的字母 c 使其长度达到 3 或以上。

例如，以 "hello" 为例，我们可以对字母组 "o" 扩张得到 "hellooo"，但是无法以同样的方法得到 "helloo" 因为字母组 "oo" 长度小于 3。此外，我们可以进行另一种扩张 "ll" -> "llll" 以获得 "hellllooo"。如果 S = "hellllooo"，那么查询词 "hello" 是可扩张的，因为可以对它执行这两种扩张操作使得 query = "hello" -> "hellooo" -> "hellllooo" = S。

输入一组查询单词，输出其中可扩张的单词数量。

```
示例:
输入:
S = "heeellooo"
words = ["hello", "hi", "helo"]
输出: 1
解释:
我们能通过扩张 "hello" 的 "e" 和 "o" 来得到 "heeellooo"。
我们不能通过扩张 "helo" 来得到 "heeellooo" 因为 "ll" 的长度小于 3 。
```

说明:

- 0 <= len(S) <= 100。
- 0 <= len(words) <= 100。
- 0 <= len(words[i]) <= 100。

2. 简单实现

统计S每个阶段字符和数量，再遍历所有word依次统计和判断

```
class Solution {
public:
    int expressiveWords(string S, vector<string>& words) {
        vector<pair<char, int>> pattern;//S的各个字母组的字母和数量
        int len = S.size();
        int size = words.size();
        if(len <= 0 || size <= 0) return 0;
        char c = S[0];
        int cnt = 1;
        for(int i = 1; i < len; i++){
            if(S[i] == c)
                cnt++;
            else{
                pattern.push_back(make_pair(c, cnt));
                c = S[i];
                cnt = 1;
            }
        }
        pattern.push_back(make_pair(c, cnt));

        int ans = 0;
```

```
        for(int i = 0; i < size; i++){//依次查看各个单词word
            int idx = 0;//pattern中待匹配的字母组下标
            len = words[i].size();
            if(len <= 0) continue;//空字符串
            if(words[i][0] != pattern[0].first) continue;//字母不符
            c = words[i][0];
            cnt = 1;
            for(int j = 1; j <= len; j++){
                if(j == len && idx == pattern.size()-1){//结尾&&正好匹配到最后一个阶段
                    if((pattern[idx].second<3 && pattern[idx].second==cnt) //小于3要
求数量匹配
                       || pattern[idx].second>=3){//大于3只要1~pattern[idx].second,
在遍历时限制
                        ans++;
                    }
                }
                else if(words[i][j] == c){
                    cnt++;
                    if(cnt > pattern[idx].second)//超出数量限制
                        break;
                }
                else{
                    if((pattern[idx].second<3 && pattern[idx].second==cnt)
                       || pattern[idx].second>=3){//当前阶段符合要求
                        idx++;//新阶段
                        if(idx>=pattern.size() || words[i][j] !=
pattern[idx].first)//不符
                            break;
                        c = words[i][j];
                        cnt = 1;
                    }
                    else
                        break;
                }
            }
        }
        return ans;
    }
};
```

38. 外观数列（简单）

1. 题目描述

「外观数列」是一个整数序列，从数字 1 开始，序列中的每一项都是对前一项的描述。前五项如下：

```
1. 1
2. 11
3. 21
4. 1211
5. 111221
1 被读作  "one 1"  ("一个一") ，即 11。
11 被读作 "two 1s" ("两个一") ，即 21。
21 被读作 "one 2",  "one 1"  ("一个二" ,  "一个一") ，即 1211。
```

给定一个正整数 n（1 ≤ n ≤ 30），输出外观数列的第 n 项。

注意：整数序列中的每一项将表示为一个字符串。

```
示例 1:
输入: 1
输出: "1"
解释: 这是一个基本样例。

示例 2:
输入: 4
输出: "1211"
解释: 当 n = 3 时，序列是 "21"，其中我们有 "2" 和 "1" 两组，"2" 可以读作 "12"，也就是出现频次 = 1 而 值 = 2；类似 "1" 可以读作 "11"。所以答案是 "12" 和 "11" 组合在一起，也就是 "1211"。
```

2. 简单实现

其实最快的方法是：提前算出来n为1~30的结果，直接判断n返回，但是练习代码的话还是把正的写了出来

```cpp
class Solution {
public:
    string countAndSay(int n) {
        if(n == 1) return "1";
        string pre = "1";
        while(--n){
            string cur = "";
            char c = pre[0];
            int cnt = 1;
            for(int i = 1; i < pre.size(); i++){
                if(pre[i] == c)
                    cnt++;
                else{
                    cur += to_string(cnt) + c;
                    c = pre[i];
                    cnt = 1;
                }
            }
            cur += to_string(cnt) + c;
            pre = cur;
        }
        return pre;
    }
};
```

208.实现Trie（前缀树）（中等）

1. 题目描述

   实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三个操作。

   ```
   示例:
   Trie trie = new Trie();
   trie.insert("apple");
   trie.search("apple");   // 返回 true
   trie.search("app");     // 返回 false
   trie.startsWith("app"); // 返回 true
   trie.insert("app");
   trie.search("app");     // 返回 true
   ```

   说明:

   - 你可以假设所有的输入都是由小写字母 a-z 构成的。
   - 保证所有输入均为非空字符串。

2. 简单实现

```
class Trie {
public:
    unordered_map<char, Trie*> children;
    bool isWord;
    /** Initialize your data structure here. */
    Trie() {
        isWord = false;
    }

    /** Inserts a word into the trie. */
    void insert(string word) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            if(cur->children.count(word[i]) <= 0)
                cur->children[word[i]] = new Trie();
            cur = cur->children[word[i]];
        }
        cur->isWord = true;
    }

    /** Returns if the word is in the trie. */
    bool search(string word) {
        Trie* cur = this;
        for(int i = 0; i < word.size(); i++){
            // cout << word[i];
            if(cur->children.count(word[i]) <= 0)
                return false;
            cur = cur->children[word[i]];
        }
        return cur->isWord;
    }
```

```
    /** Returns if there is any word in the trie that starts with the given prefix.
*/
    bool startsWith(string prefix) {
        Trie* cur = this;
        for(int i = 0; i < prefix.size(); i++){
            if(cur->children.count(prefix[i]) <= 0)
                return false;
            cur = cur->children[prefix[i]];
        }
        return true;
    }
};
```

1110. 删点成林（中等）

1. 题目描述

   给出二叉树的根节点 root，树上每个节点都有一个不同的值。

   如果节点值在 to_delete 中出现，我们就把该节点从树上删去，最后得到一个森林（一些不相交的树构成的集合）。

   返回森林中的每棵树。你可以按任意顺序组织答案。

```
示例：
输入：root = [1,2,3,4,5,6,7], to_delete = [3,5]
输出：[[1,2,null,4],[6],[7]]
```

提示：

- 树中的节点数最大为 1000。
- 每个节点都有一个介于 1 到 1000 之间的值，且各不相同。
- to_delete.length <= 1000
- to_delete 包含一些从 1 到 1000、各不相同的值。

2. 简单实现

   每次删除节点要做的操作有：

   - 将其父节点对应的该子节点置为NULL
   - 以其非空左右子节点为跟，产生新的树

```
class Solution {
public:
    //获得树root的每个节点的<本节点值， <本节点指针， 父节点指针>>
    void getDic(TreeNode* root, unordered_map<int, pair<TreeNode*, TreeNode*>>& m){
        if(!root) return;
        if(root->left){
            m[root->left->val] = make_pair(root->left, root);
            getDic(root->left, m);
        }
        if(root->right){
```

```
            m[root->right->val] = make_pair(root->right, root);
            getDic(root->right, m);
        }
    }
    vector<TreeNode*> delNodes(TreeNode* root, vector<int>& to_delete) {
        if(!root || to_delete.size() == 0)
            return {};
        unordered_map<int, pair<TreeNode*, TreeNode*>> m;
        m[root->val] = make_pair(root, new TreeNode(-1));//根节点无父节点
        getDic(root, m);
        unordered_set<TreeNode*> s;//存储当前的森林内所有树的根节点指针
        s.insert(root);
        for(int i = 0; i < to_delete.size(); i++){
            TreeNode* cur = m[to_delete[i]].first;
            TreeNode* pre = m[to_delete[i]].second;
            if(pre->val != -1){//有父节点
                if(pre->left == cur)//是左子
                    pre->left = NULL;
                if(pre->right == cur)//是右子
                    pre->right = NULL;
            }
            if(cur->left)//左子树
                s.insert(cur->left);
            if(cur->right)//右子树
                s.insert(cur->right);
            if(s.find(cur) != s.end())//如果是当前某棵树的根节点，要
                s.erase(cur);
        }
        vector<TreeNode*> ans(s.begin(), s.end());
        return ans;
    }
};
```

3. 优化解法

首先把to_delete变成set，此处不多说； 节点进入结果当中，有2个条件：

- 不被删除
- 父节点不存在

因此在遍历过程中，将parentExists标志传递给子节点，子递归就可以选择是否加入到结果。另外，如果子节点被删除，父节点的left、right字段需要更新。

```
class Solution {
    Set<Integer> toDelete;
    public List<TreeNode> delNodes(TreeNode root, int[] to_delete) {
        toDelete = Arrays.stream(to_delete).boxed().collect(Collectors.toSet());
        List<TreeNode> ans = new ArrayList<>();
        helper(root, ans, false);
        return ans;
    }
    boolean helper(TreeNode root, List<TreeNode> ans, boolean parentExists) {
        boolean del = false;
        if (root == null)
```

```
            return del;
        del = toDelete.contains(root.val);
        if (helper(root.left, ans, !del))
            root.left = null;
        if (helper(root.right, ans, !del))
            root.right = null;
        if (!del && !parentExists)
            ans.add(root);
        return del;
    }
}
```

283. 移动零（简单）

1. 题目描述 给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

   示例:

   ```
   输入: [0,1,0,3,12]
   输出: [1,3,12,0,0]
   ```

   说明:

   1. 必须在原数组上操作，不能拷贝额外的数组。
   2. 尽量减少操作次数
2. 简单实现

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int l = 0, r = 0;
        int len = nums.size();
        while(r < len){
            if(nums[r] != 0)
                nums[l++] = nums[r];
            r++;
        }
        while(l < len)
            nums[l++] = 0;
    }
};
```

68. 文本左右对齐（困难）

1. 题目描述

   给定一个单词数组和一个长度 maxWidth，重新排版单词，使其成为每行恰好有 maxWidth 个字符，且左右两端对齐的文本。

你应该使用“贪心算法”来放置给定的单词；也就是说，尽可能多地往每行中放置单词。必要时可用空格 ' ' 填充，使得每行恰好有 maxWidth 个字符。

要求尽可能均匀分配单词间的空格数量。如果某一行单词间的空格不能均匀分配，则左侧放置的空格数要多于右侧的空格数。

文本的最后一行应为左对齐，且单词之间不插入额外的空格。

说明:

- 单词是指由非空格字符组成的字符序列。
- 每个单词的长度大于 0，小于等于 maxWidth。
- 输入单词数组 words 至少包含一个

```
示例:
输入:
words = ["This", "is", "an", "example", "of", "text", "justification."]
maxWidth = 16
输出:
[
   "This    is    an",
   "example  of text",
   "justification.  "
]

示例 2:
输入:
words = ["What","must","be","acknowledgment","shall","be"]
maxWidth = 16
输出:
[
  "What   must   be",
  "acknowledgment  ",
  "shall be        "
]
解释: 注意最后一行的格式应为 "shall be    " 而不是 "shall     be",
      因为最后一行应为左对齐，而不是左右两端对齐。
      第二行同样为左对齐，这是因为这行只包含一个单词。

示例 3:
输入:
words = ["Science","is","what","we","understand","well","enough","to","explain",
         "to","a","computer.","Art","is","everything","else","we","do"]
maxWidth = 20
输出:
[
  "Science  is  what we",
  "understand      well",
  "enough to explain to",
  "a  computer.  Art is",
  "everything  else  we",
  "do                  "
]
```

2. 简单实现

按要求一行一行弄就行

```
class Solution {
public:
    vector<string> fullJustify(vector<string>& words, int maxWidth) {
        int l = 0, r = 0;//当前行包括的单词words[l...r)
        int size = words.size();
        int cur_len = 0;//words[l...r)的长度和，注意这里未包含所需的最少空格数r-l-1
        vector<string> ans;
        while(r < size){
            if(cur_len + r-l + words[r].size() > maxWidth){//加上word[r]后该行容纳不下
                int spaces = maxWidth - cur_len;//除字母意外的空格数
                int cnt = r-l-1;//需要填充空格的单词分割区间数
                string cur = "";
                if(cnt == 0) //只有一个单词，无需用空格分隔
                    cur += words[l++];
                else{
                    int tab_len = spaces / cnt;//每个分隔区间的最短空格长度
                    string tab = "";//分隔空格串
                    while(tab_len--)
                        tab += ' ';
                    int mod = spaces % cnt;//平均后多出来的空格数，依次分给最左边的mod个分隔空格串
                    while(l < r){
                        cur += words[l++];//填入单词
                        if(l != r){//不是最后一个单词，后面要填充分隔空格串
                            cur += tab;
                            if(mod-- > 0)
                                cur += ' ';
                        }
                    }
                }
                int len = cur.size();
                while(len++ < maxWidth)//剩余不足maxWidth的部分用0补足
                    cur += ' ';
                ans.push_back(cur);
                cur_len = 0;
            }
            cur_len += words[r++].size();
        }
        //处理最后一行左对齐，所有单词间只有一个空格
        string cur = words[l++];
        while(l < r)
            cur += ' ' + words[l++];
        int len = cur.size();
        while(len++ < maxWidth)
            cur += ' ';
        ans.push_back(cur);
        return ans;
    }
};
```

127. 单词接龙（中等）

1. 题目描述

   给定两个单词（beginWord 和 endWord）和一个字典，找到从 beginWord 到 endWord 的最短转换序列的长度。转换需遵循如下规则：

   1. 每次转换只能改变一个字母。
   2. 转换过程中的中间单词必须是字典中的单词。

   说明:

   - 如果不存在这样的转换序列，返回 0。
   - 所有单词具有相同的长度。
   - 所有单词只由小写字母组成。
   - 字典中不存在重复的单词。
   - 你可以假设 beginWord 和 endWord 是非空的，且二者不相同。

```
示例 1:
输入:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]
输出: 5
解释: 一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog",
     返回它的长度 5。

示例 2:
输入:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]
输出: 0
解释: endWord "cog" 不在字典中，所以无法进行转换。
```

2. 简单实现

   双向同时BFS比单向BFS效率高

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
        if(wordList.size() <= 0) return 0;
        unordered_map<string, vector<string>> m;//用map存储图，<节点,该节点的所有相邻节
点>
        int len = wordList[0].size();
        for(int i = 0; i < wordList.size()-1; i++)
            for(int j = i + 1; j < wordList.size(); j++){//遍历构造图
                int cnt = 0;//不同字符数
                for(int k = 0; k < len && cnt <= 1; k++){
                    if(wordList[i][k] != wordList[j][k])
                        cnt++;
                }
```

```
                if(cnt == 1){//相邻节点
                    if(m.count(wordList[i]) <= 0)
                        m[wordList[i]] = {wordList[j]};
                    else
                        m[wordList[i]].push_back(wordList[j]);
                    if(m.count(wordList[j]) <= 0)
                        m[wordList[j]] = {wordList[i]};
                    else
                        m[wordList[j]].push_back(wordList[i]);
                }
            }
        if(m.count(beginWord) <= 0){//起始字符串可以不在字典中，因此加入图中
            for(int i = 0; i < wordList.size(); i++){
                int cnt = 0;
                for(int k = 0; k < len && cnt <= 1; k++){
                    if(wordList[i][k] != beginWord[k])
                        cnt++;
                }
                if(cnt == 1){
                    if(m.count(beginWord) <= 0)
                        m[beginWord] = {wordList[i]};
                    else
                        m[beginWord].push_back(wordList[i]);
                    if(m.count(wordList[i]) <= 0)
                        m[wordList[i]] = {beginWord};
                    else
                        m[wordList[i]].push_back(beginWord);
                }
            }
        }
        if(m.count(endWord) <= 0) return 0;//endWord不在图中（不存在该节点或该节点无相邻节点）
        //双向BFS
        int ans = 0;
        queue<string> lq;
        lq.push(beginWord);
        unordered_set<string> lvisited;
        lvisited.insert(beginWord);
        unordered_set<string> l_cur;
        l_cur.insert(beginWord);

        queue<string> rq;
        rq.push(endWord);
        unordered_set<string> rvisited;
        rvisited.insert(endWord);
        unordered_set<string> r_cur;
        r_cur.insert(endWord);
        while(!lq.empty() && !rq.empty()){
            ans++;
            int size = lq.size();
            l_cur.clear();
            for(int i = 0; i < size; i++){
                string cur = lq.front();
```

```
                lq.pop();
                for(auto it = m[cur].begin(); it != m[cur].end(); it++){
                    if(r_cur.count(*it) > 0) return ans+1;//注意返回序列长度
                    else if(lvisited.count(*it) <= 0){
                        lq.push(*it);
                        lvisited.insert(*it);
                        l_cur.insert(*it);
                    }
                }
            }

            ans++;
            size = rq.size();
            r_cur.clear();
            for(int i = 0; i < size; i++){
                string cur = rq.front();
                rq.pop();
                for(auto it = m[cur].begin(); it != m[cur].end(); it++){
                    if(l_cur.count(*it) > 0) return ans+1;//注意返回序列长度
                    else if(rvisited.count(*it) <= 0){
                        rq.push(*it);
                        rvisited.insert(*it);
                        r_cur.insert(*it);
                    }
                }
            }
        }
        return 0;
    }
};
```

72. 编辑距离（困难）

1. 题目描述

   给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

   你可以对一个单词进行如下三种操作：

   - 插入一个字符
   - 删除一个字符
   - 替换一个字符

```
示例 1:
输入: word1 = "horse", word2 = "ros"
输出: 3
解释:
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')

示例 2:
输入: word1 = "intention", word2 = "execution"
```

```
输出：5
解释：
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')
```

2. 正确解法——动态规划

我们可以对任意一个单词进行三种操作：

- 插入一个字符；
- 删除一个字符；
- 替换一个字符。

题目给定了两个单词，设为 `A` 和 `B`，这样我们就能够六种操作方法。

但我们可以发现，如果我们有单词 `A` 和单词 `B`：

- 对单词 `A` 删除一个字符和对单词 `B` 插入一个字符是等价的。例如当单词 `A` 为 `doge`，单词 `B` 为 `dog` 时，我们既可以删除单词 `A` 的最后一个字符 `e`，得到相同的 `dog`，也可以在单词 `B` 末尾添加一个字符 `e`，得到相同的 `doge`；
- 同理，对单词 `B` 删除一个字符和对单词 `A` 插入一个字符也是等价的；
- 对单词 `A` 替换一个字符和对单词 `B` 替换一个字符是等价的。例如当单词 `A` 为 `bat`，单词 `B` 为 `cat` 时，我们修改单词 `A` 的第一个字母 `b -> c`，和修改单词 `B` 的第一个字母 `c -> b` 是等价的。

这样以来，本质不同的操作实际上只有三种：

- 在单词 `A` 中插入一个字符；
- 在单词 `B` 中插入一个字符；
- 修改单词 `A` 的一个字符。

这样以来，我们就可以把原问题转化为规模较小的子问题。我们用 `A = horse`，`B = ros` 作为例子，来看一看是如何把这个问题转化为规模较小的若干子问题的。

- **在单词 `A` 中插入一个字符**：如果我们知道 `horse` 到 `ro` 的编辑距离为 `a`，那么显然 `horse` 到 `ros` 的编辑距离不会超过 `a + 1`。这是因为我们可以在 `a` 次操作后将 `horse` 和 `ro` 变为相同的字符串，只需要额外的 `1` 次操作，在单词 `A` 的末尾添加字符 `s`，就能在 `a + 1` 次操作后将 `horse` 和 `ro` 变为相同的字符串；
- **在单词 `B` 中插入一个字符**：如果我们知道 `hors` 到 `ros` 的编辑距离为 `b`，那么显然 `horse` 到 `ros` 的编辑距离不会超过 `b + 1`，原因同上；
- **修改单词 `A` 的一个字符**：如果我们知道 `hors` 到 `ro` 的编辑距离为 `c`，那么显然 `horse` 到 `ros` 的编辑距离不会超过 `c + 1`，原因同上。

那么从 `horse` 变成 `ros` 的编辑距离应该为 `min(a + 1, b + 1, c + 1)`。

注意：为什么我们总是在单词 `A` 和 `B` 的末尾插入或者修改字符，能不能在其它的地方进行操作呢？答案是可以的，但是我们知道，操作的顺序是不影响最终的结果的。例如对于单词 `cat`，我们希望在 `c` 和 `a` 之间添加字符 `d` 并且将字符 `t` 修改为字符 `b`，那么这两个操作无论为什么顺序，都会得到最终的结果 `cdab`。

你可能觉得 `horse` 到 `ro` 这个问题也很难解决。但是没关系，我们可以继续用上面的方法拆分这个问题，对于这个问题拆分出来的所有子问题，我们也可以继续拆分，直到：

- 字符串 `A` 为空，如从 `` 转换到 `ro`，显然编辑距离为字符串 `B` 的长度，这里是 `2`；
- 字符串 `B` 为空，如从 `horse` 转换到 ``，显然编辑距离为字符串 `A` 的长度，这里是 `5`。

因此，我们就可以使用动态规划来解决这个问题了。我们用 `D[i][j]` 表示 `A` 的前 `i` 个字母和 `B` 的前 `j` 个字母之间的编辑距离。

那么我们可以写出如下的状态转移方程：

- 若 `A` 和 `B` 的最后一个字母相同：

$$
\begin{aligned}
D[i][j] &= \min(D[i][j-1]+1, D[i-1][j]+1, D[i-1][j-1]) \\
&= 1+\min(D[i][j-1], D[i-1][j], D[i-1][j-1]-1)
\end{aligned}
$$

- 若 `A` 和 `B` 的最后一个字母不同：

$$
D[i][j] = 1+\min(D[i][j-1], D[i-1][j], D[i-1][j-1])
$$

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int n = word1.length();
        int m = word2.length();
```

```
            // 有一个字符串为空串
            if (n * m == 0) return n + m;

            // DP 数组
            int D[n + 1][m + 1];

            // 边界状态初始化
            for (int i = 0; i < n + 1; i++) {
                D[i][0] = i;
            }
            for (int j = 0; j < m + 1; j++) {
                D[0][j] = j;
            }

            // 计算所有 DP 值
            for (int i = 1; i < n + 1; i++) {
                for (int j = 1; j < m + 1; j++) {
                    int left = D[i - 1][j] + 1;
                    int down = D[i][j - 1] + 1;
                    int left_down = D[i - 1][j - 1];
                    if (word1[i - 1] != word2[j - 1]) left_down += 1;
                    D[i][j] = min(left, min(down, left_down));

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

41. 缺失的第一个正数（困难）

1. 题目描述给你一个未排序的整数数组，请你找出其中没有出现的最小的正整数。

    ```
    示例 1:
    输入: [1,2,0]
    输出: 3
    ```

示例 2: 输入: [3,4,-1,1] 输出: 2

示例 3: 输入: [7,8,9,11,12] 输出: 1 ```

提示：你的算法的时间复杂度应为O(n)，并且只能使用常数级别的额外空间。

2. 正确解法

    题目说常数空间，没说传入的数组空间不能用啊，就是一个就地的鸽巢原理进化版

    - 首先我们可以不考虑负数和零，因为不需要考虑。同样可以不考虑大于 n 的数字，因为首次缺失的正数一定小于或等于 n + 1 。缺失的正数为 n + 1 的情况会单独考虑。
    - 正常数据的话，数组元素值应该是[1,2,3,4,5]连续的数，异常情况为[1,2,4,5]
    - 正常数据我们认为0下标应该存放1，1下标存放2...
    - 遍历数组，找到 1<=元素<=数组长度的元素，如5，将他放到应该放置的位置，即4号索引

- 遇到范围之外的数值，如-1 或者超过数组长度的值，不交换，继续下一个
- 处理之后的数据为[1,2,4,5]，再遍历一遍数组，下标+1应该是正确值，找出第一个不符合的即可

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        int numsSize = nums.size();
         // 换序的过程即将数换到正确位置的过程，最多换n个，故复杂度为O(n)
        for (int i = 0; i < numsSize; i++)
            while ((nums[i]>=1 && nums[i]<=numsSize) && (nums[i] !=
nums[nums[i]-1]))
                swap(nums[i], nums[nums[i]-1]);// 该数字不属于当前位置 交换a[i] 和
a[i]-1位置上的数字
        // 默认缺少的是正整数是 数组长度+1
        int ans = numsSize + 1;
        // 如果中间有缺少的，直接替换ans
        for (int i = 0; i < numsSize; i++) {
            if (nums[i] != (i + 1)) {
                ans = i + 1;
                break;
            }
        }
        return ans;
    }
};
```

43. 字符串相乘（中等）

1. 题目描述

   给定两个以字符串形式表示的非负整数 num1 和 num2，返回 num1 和 num2 的乘积，它们的乘积也表示为字符串形式。

```
示例 1:
输入: num1 = "2", num2 = "3"
输出: "6"

示例 2:
输入: num1 = "123", num2 = "456"
输出: "56088"
```

   说明：

   - num1 和 num2 的长度小于110。
   - num1 和 num2 只包含数字 0-9。
   - num1 和 num2 均不以零开头，除非是数字 0 本身。
   - 不能使用任何标准库的大数类型（比如 BigInteger）或直接将输入转换为整数来处理。

2. 简单实现

   模拟竖式乘法，有很多trick，详见代码

```
class Solution {
public:
    string multiply(string num1, string num2) {
        if(num1 == "0" || num2 == "0")
            return "0";
        int s1 = num1.size(), s2 = num2.size();

        vector<int> ans(s1 + s2, 0);//结果数字长度在s1+s2-1和s1+s2位之间
        for(int i = s1-1; i >= 0; i--)
            for(int j = s2-1; j >= 0; j--)
                ans[i + j + 1] += (num1[i] - '0') * (num2[j] - '0');//不管进位，先加
上

        for(int i = s1+s2-1; i >= 0; i--) {
            if(ans[i] > 9) {//统一进位
                int t = ans[i];
                ans[i] = t % 10;
                ans[i - 1] += (t / 10);
            }
        }


        string aa = "";
        int idx = 0;
        if(ans[0] == 0)// 答案长度为s1+s2-1或s1+s2，故只需要判断首位是否为0
            idx = 1;
        while(idx < s1+s2) {
            aa += (ans[idx] + '0');
            idx++;
        }
        return aa;
    }
};
```

361. 轰炸敌人（中等）

1. 题目描述

   想象一下炸弹人游戏，在你面前有一个二维的网格来表示地图，网格中的格子分别被以下三种符号占据：

   - 'W' 表示一堵墙
   - 'E' 表示一个敌人
   - '0'（数字 0）表示一个空位

请你计算一个炸弹最多能炸多少敌人。

由于炸弹的威力不足以穿透墙体，炸弹只能炸到同一行和同一列没被墙体挡住的敌人。

注意：你只能把炸弹放在一个空的格子里

```
示例：
输入：[["0","E","0","0"],["E","0","W","E"],["0","E","0","0"]]
输出：3
解释：对于如下网格
0 E 0 0
E 0 W E
0 E 0 0
假如在位置 (1,1) 放置炸弹的话，可以炸到 3 个敌人
```

2. 简单实现

   遍历每行，从左到右遍历统计空格子左侧可以炸到的敌人数，再从右到左遍历统计空格子右侧可以炸到的敌人数

   遍历每列，从上到下遍历统计空格子上侧可以炸到的敌人数，再从下到上遍历统计空格子下侧可以炸到的敌人数

```
class Solution {
public:
    int maxKilledEnemies(vector<vector<char>>& grid) {
        int m = grid.size();
        if(m <= 0) return 0;
        int n = grid[0].size();
        vector<vector<int>> cnt(m, vector<int>(n, 0));
        int ans = 0;
        for(int i = 0; i < m; i++){
            int cur = 0;
            for(int j = 0; j < n; j++){//左->右
                if(grid[i][j] == '0')
                    cnt[i][j] += cur;
                else if(grid[i][j] == 'E')
                    cur++;
                else//遇到墙
                    cur = 0;
            }
```

```
            cur = 0;
            for(int j = n-1; j >= 0; j--){//右->左
                if(grid[i][j] == 'O')
                    cnt[i][j] += cur;
                else if(grid[i][j] == 'E')
                    cur++;
                else
                    cur = 0;
            }
        }
        for(int j = 0; j < n; j++){
            int cur = 0;
            for(int i = 0; i < m; i++){//上->下
                if(grid[i][j] == 'O')
                    cnt[i][j] += cur;
                else if(grid[i][j] == 'E')
                    cur++;
                else
                    cur = 0;
            }
            cur = 0;
            for(int i = m-1; i >= 0; i--){//下->上
                if(grid[i][j] == 'O'){
                    cnt[i][j] += cur;
                    ans = max(ans, cnt[i][j]);//同时统计答案
                }
                else if(grid[i][j] == 'E')
                    cur++;
                else
                    cur = 0;
            }
        }
        return ans;
    }
};
```

753. 破解保险箱（困难）

1. 题目描述

有一个需要密码才能打开的保险箱。密码是 n 位数, 密码的每一位是 k 位序列 0, 1, ..., k-1 中的一个 。

你可以随意输入密码，保险箱会自动记住最后 n 位输入，如果匹配，则能够打开保险箱。

举个例子，假设密码是 "345"，你可以输入 "012345" 来打开它，只是你输入了 6 个字符.

请返回一个能打开保险箱的最短字符串。

```
示例1:
输入: n = 1, k = 2
输出: "01"
说明: "10"也可以打开保险箱。

示例2:
输入: n = 2, k = 2
输出: "00110"
说明: "01100", "10011", "11001" 也能打开保险箱。
```

提示:

- n 的范围是 [1, 4]。
- k 的范围是 [1, 10]。
- k^n 最大可能为 4096。

2. 正确解法

   想法规律大体上找到了，但是忽略了一点所以没做出来

   - 考虑示例2，可以看做00->01->11->10的压缩，同理，n=3,k=2时，可以有答案""0011101000""，即001->011->111->110->101->010->100->000的压缩,路径上每一步都相当于取前一个数的后n-1位,再在末尾加上0~k-1的某个数,最后结果串最小长度为`n+pow(k,n)-1`
   - 有了这个规律,一开始就想直接写循环来做,从全0开始,每次在末尾依次尝试添加0~k-1直到成功为止,然而,这个是不对的,比如n=2,k=2时,00->01->10->就走不下去了,看题解说,这涉及到欧拉回路的知识

     Hierholzer 算法可以在一个欧拉图中找出欧拉回路。

     我们将所有的 `k - 1` 位数作为节点，共有 k^{n-1} 个节点。每个节点有 `k` 条入边和出边，如果当前节点对应的数为 $a_1a_2\cdots a_k$，那么它的第 `k` 条出边连向 $a_2\cdots a_kk$，第 `k` 条入边由 $ka_1a_2\cdots a_{k-1}$ 连向它，这样我们从一个节点顺着一条边（编号为 `x`）走到另一个节点，就相当于输入了数字 `x`。我们的目标是要经过所有的边，这是因为每一个节点以及它们的出边都对应了 `k` 个数，例如 `k = 4, n = 3` 时，节点分别为 `00, 01, 02, ..., 32, 33`，边分别为 `0, 1, 2, 3`，那么 `00` 和它的出边对应了 `000, 001, 002, 003`，`32` 和它的出边对应了 `320, 321, 322, 323`。一共有 $k^{n-1} * k = k^n$ 个数，即为所有可能的密码。

     由于这个图的每个节点都有 `k` 条入边和出边，因此它一定存在一个欧拉回路，即可以从任意一个节点开始，一次性不重复地走完所有的边且回到该节点。因此我们可以用 Hierholzer 算法找出这条欧拉回路，设开始的节点对应的数位 `init`，欧拉回路中每条边的编号为 `l1, l2, l3, ...`，那么最终的字符串即为 `init l1 l2 l3 ...`。Hierholzer 算法如下：

     - 我们从任意节点 `u` 开始，任意地经过未经过的边，直到我们“无路可走”。可以发现，我们最终一定会停在节点 `u`，这是因为所有节点的入度和出度都相等。
     - 我们得到了一条从 `u` 开始到 `u` 结束的回路，这条回路上仍然有些节点有未经过的出边。我么从某个这样的节点 `v` 开始，把 `v` 看成 `u`，继续得到一条从 `v` 开始到 `v` 结束的回路，再嵌入之前的回路中，即 `u -> ... -> v -> ... -> u` 变为 `u -> ... -> v -> ... -> v -> ... -> u`。以此类推，直到没有未经过的边，此时我们就找到了一条欧拉回路。

     因此,必须用Hierholzer 算法,实际上就是后序的dfs,原理证明不会,但是代码还是好急的

```
class Solution {
```

```
public:
    int visited[10001] = {0};//记录访问情况
    int P = 0;//去掉最高位数字时用的
    void dfs(int k, int i, string& res) {//i为当前字符串对应的数字
        i -= i / P * P;//去掉最高位
        i *= 10;//所有位左移一位
        for (int j = 0; j < k; ++j) {//末尾添加j
            int t = i + j;
            if (!visited[t]) {
                visited[t] = 1;
                dfs(k, t, res);
                res += j + '0';//后序遍历
            }
        }
    }
    string crackSafe(int n, int k) {
        P = pow(10, n - 1);
        visited[0] = 1;
        string res;
        dfs(k, 0, res);
        res.append(n, '0');//最后添加起点
        return res;
    }
};
```

766. 托普利茨矩阵(简单)

1. 题目描述

   如果一个矩阵的每一方向由左上到右下的对角线上具有相同元素，那么这个矩阵是托普利茨矩阵。

   给定一个 M x N 的矩阵，当且仅当它是托普利茨矩阵时返回 True。

```
示例 1:
输入:
matrix = [
  [1,2,3,4],
  [5,1,2,3],
  [9,5,1,2]
]
输出: True
解释:
在上述矩阵中，其对角线为:
"[9]", "[5, 5]", "[1, 1, 1]", "[2, 2, 2]", "[3, 3]", "[4]"。
各条对角线上的所有元素均相同，因此答案是True。

示例 2:
输入:
matrix = [
  [1,2],
  [2,2]
]
```

```
输出: False
解释:
对角线"[1, 2]"上的元素不同。
```

说明:

- matrix 是一个包含整数的二维数组。
- matrix 的行数和列数均在 [1, 20]范围内。
- matrix[i][j] 包含的整数在 [0, 99]范围内。

进阶:

- 如果矩阵存储在磁盘上，并且磁盘内存是有限的，因此一次最多只能将一行矩阵加载到内存中，该怎么办?
- 如果矩阵太大以至于只能一次将部分行加载到内存中，该怎么办?

2. 简单实现

用链表,一行一行的移位比较

```
class Solution {
public:
    bool isToeplitzMatrix(vector<vector<int>>& matrix) {
        int m = matrix.size();
        int n = matrix[0].size();
        if(m == 1 || n == 1)
            return true;
        list<int> row;
        for(int i = 0; i < n; i++)
            row.push_back(matrix[0][i]);
        for(int i = 1; i < m; i++){
            row.pop_back();
            auto it = row.rbegin();
            for(int j = n-1; j > 0; j--){
                if(matrix[i][j] != *it)
                    return false;
                it++;
            }
            row.push_front(matrix[i][0]);
        }
        return true;
    }
};
```

55. 跳跃游戏(中等)

1. 题目描述

给定一个非负整数数组，你最初位于数组的第一个位置。数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

```
示例 1:
输入: [2,3,1,1,4]
输出: true
解释: 我们可以先跳 1 步，从位置 0 到达 位置 1，然后再从位置 1 跳 3 步到达最后一个位置。

示例 2:
输入: [3,2,1,0,4]
输出: false
解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。
```

2. 简单实现

从右往左遍历,记录能到达最后一个位置的点中最靠左的,只要能到这个点,就一定能到终点

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int size = nums.size();
        int l = size-1;
        for(int i = size - 2; i >= 0; i--)
            if(i+nums[i] >= l)
                l = i;
        return l == 0;
    }
};
```

3. 改进

直接从左向右就行,贪心法计算当前能跳到的最远的地方

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

62. 不同路径(中等)

1. 题目描述

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

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

问总共有多少条不同的路径?

例如，上图是一个7 x 3 的网格。有多少可能的路径?

```
示例 1:
输入: m = 3, n = 2
输出: 3
解释:
从左上角开始，总共有 3 条路径可以到达右下角。
1. 向右 -> 向右 -> 向下
2. 向右 -> 向下 -> 向右
3. 向下 -> 向右 -> 向右

示例 2:
输入: m = 7, n = 3
输出: 28
```

提示:

- 1 <= m, n <= 100
- 题目数据保证答案小于等于 2 * 10 ^ 9

2. 简单实现

二维dp,可以优化为一维

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        if(m ==1 || n==1) return 1;
        vector<int> dp = vector<int>(n+1);
        for(int i = 1; i <= n; i++)
            dp[i] = 1;
        for(int i = 2; i <= m; i++)
            for(int j = 2; j <= n; j++)
                dp[j] = dp[j]+dp[j-1];
        return dp[n];
    }
};
```

70. 爬楼梯(简单)

1. 题目描述

   假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

   注意：给定 n 是一个正整数。

```
示例 1:
输入:  2
输出:  2
解释:  有两种方法可以爬到楼顶。
1. 1 阶 + 1 阶
2. 2 阶

示例 2:
输入:  3
输出:  3
解释:  有三种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

2. 简单实现

```cpp
class Solution {
public:
    int climbStairs(int n) {
        if(n == 1) return 1;
        if(n == 2) return 2;
        int pre1 = 1;
        int pre2 = 2;
        for(int i = 3; i <= n; i++){
            int cur = pre1 + pre2;
            pre1 = pre2;
            pre2 = cur;
        }
        return pre2;
    }
};
```

281. 锯齿迭代器(中等)

1. 题目描述

   给出两个一维的向量，请你实现一个迭代器，交替返回它们中间的元素。

```
示例：
输入：
v1 = [1,2]
v2 = [3,4,5,6]
输出：[1,3,2,4,5,6]
解析：通过连续调用 next 函数直到 hasNext 函数返回 false，
     next 函数返回值的次序应依次为：[1,3,2,4,5,6]。
```

拓展：假如给你 k 个一维向量呢？你的代码在这种情况下的扩展性又会如何呢?

拓展声明：“锯齿” 顺序对于 k > 2 的情况定义可能会有些歧义。所以，假如你觉得 “锯齿” 这个表述不妥，也可以认为这是一种 “循环”。例如：

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

2. 简单实现

直接做的扩展

```
class ZigzagIterator {
public:
    vector<vector<int>> data;//依次存储v1,v2...
    int k;//data.size()
    int idx1;//指向当前数组v
    int idx2;//指向当前要访问的数组中的元素
    ZigzagIterator(vector<int>& v1, vector<int>& v2) {
        idx1 = idx2 = 0;
        k = 0;
        if(v1.size() > 0){
            k++;
            data.push_back(v1);
        }
        if(v2.size() > 0){
            k++;
            data.push_back(v2);
        }
    }

    int next() {
        int ans = data[idx1][idx2];
        //找到下一个要访问的
        idx1++;
        while(idx1 < k && idx2 >= data[idx1].size())//找一圈,直到找到可以访问的
            idx1++;
        if(idx1 == k){//没找到,再找一圈不行就是没有了
            idx1 = 0;
            idx2++;
            while(idx1 < k && idx2 >= data[idx1].size())
                idx1++;
        }
```

```
        return ans;
    }
    bool hasNext() {
        return idx1 < k;
    }
};
```

393. UTF-8编码验证(中等)

1. 题目描述

   UTF-8 中的一个字符可能的长度为 1 到 4 字节，遵循以下的规则：

   1. 对于 1 字节的字符，字节的第一位设为0，后面7位为这个符号的unicode码。
   2. 对于 n 字节的字符 (n > 1)，第一个字节的前 n 位都设为1，第 n+1 位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的unicode码。

   这是 UTF-8 编码的工作方式：

```
Char. number range  |        UTF-8 octet sequence
     (hexadecimal)    |              (binary)
  --------------------+---------------------------------------------
  0000 0000-0000 007F | 0xxxxxxx
  0000 0080-0000 07FF | 110xxxxx 10xxxxxx
  0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
  0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

   给定一个表示数据的整数数组，返回它是否为有效的 utf-8 编码。

   注意: 输入是整数数组。只有每个整数的最低 8 个有效位用来存储数据。这意味着每个整数只表示 1 字节的数据。

```
示例 1:
data = [197, 130, 1], 表示 8 位的序列: 11000101 10000010 00000001.
返回 true 。
这是有效的 utf-8 编码，为一个2字节字符，跟着一个1字节字符。

示例 2:
data = [235, 140, 4], 表示 8 位的序列: 11101011 10001100 00000100.
返回 false 。
前 3 位都是 1 ，第 4 位为 0 表示它是一个3字节字符。
下一个字节是开头为 10 的延续字节，这是正确的。
但第二个延续字节不以 10 开头，所以是不符合规则的。
```

2. 简单实现

   按照规则判断即可

```
class Solution {
public:
    bool validUtf8(vector<int>& data) {
        int size = data.size();
```

```
        int cnt = 0;//后续需要的以10开头的字节数
        for(int i = 0; i < size; i++){
            if(cnt == 0){//新字符
                if(data[i] < 0x80)//1字节
                    continue;
                else if(data[i] < 0xC0)//格式错误
                    return false;
                else if(data[i] < 0xE0)//2字节
                    cnt = 1;
                else if(data[i] < 0xF0)//3字节
                    cnt = 2;
                else if(data[i] < 0xF8)//4字节
                    cnt = 3;
                else //格式错误
                    return false;
            }
            else{
                if(data[i] >= 0x80 && data[i] < 0xC0)//10xxxxxx
                    cnt--;
                else //格式错误
                    return false;
            }
        }
        return cnt == 0;//cnt>0说明最后一个字符格式错误
    }
};
```

8. 字符串转换整数(中等)

1. 题目描述

   请你来实现一个 atoi 函数，使其能将字符串转换成整数。

   首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。接下来的转化规则如下：

   - 如果第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字字符组合起来，形成一个有符号整数。
   - 假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成一个整数。
   - 该字符串在有效的整数部分之后也可能会存在多余的字符，那么这些字符可以被忽略，它们对函数不应该造成影响。

   注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换，即无法进行有效转换。

   在任何情况下，若函数不能进行有效的转换时，请返回 0 。

   提示：

   - 本题中的空白字符只包括空格字符 ' ' 。
   - 假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 [−231, 231 − 1]。如果数值超过这个范围，请返回 INT_MAX (231 − 1) 或 INT_MIN (−231) 。

```
示例 1:
输入: "42"
输出: 42

示例 2:
输入: "   -42"
输出: -42
解释: 第一个非空白字符为 '-', 它是一个负号。
     我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。

示例 3:
输入: "4193 with words"
输出: 4193
解释: 转换截止于数字 '3' ，因为它的下一个字符不为数字。

示例 4:
输入: "words and 987"
输出: 0
解释: 第一个非空字符是 'w', 但它不是数字或正、负号。
     因此无法执行有效的转换。

示例 5:
输入: "-91283472332"
输出: -2147483648
解释: 数字 "-91283472332" 超过 32 位有符号整数范围。
     因此返回 INT_MIN (−231) 。
```

2. 简单实现

```
class Solution {
public:
    int myAtoi(string str) {
        if(str == "") return 0;
        int len = str.size();
        int idx = 0;
        while(idx < len && str[idx] == ' ')//跳过开头空格
            idx++;
        if(idx >= len) return 0;
        int pos = true;
        if(str[idx] == '+')
            idx++;
        else if(str[idx] == '-'){
            pos = false;
            idx++;
        }
        else if(!(str[idx] >= '0' && str[idx] <= '9'))//无效转换
            return 0;
        long ans = 0;//用long方便溢出判断
        long int_max = pow(2, 31) - 1;
        long int_min = -(int_max+1);
        while(idx < len && str[idx] >= '0' && str[idx] <= '9'){
            ans = ans*10 + int(str[idx]-'0');
```

```
            idx++;
            if(pos && ans >= int_max)//正数溢出
                return int_max;
            if(!pos && ans >= -int_min)//负数溢出
                return int_min;
        }
        if(pos)
            return ans;
        else
            return -ans;
    }
};
```

79. 单词搜索(中等)

1. 题目描述

   给定一个二维网格和一个单词，找出该单词是否存在于网格中。

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

```
示例:
board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]
给定 word = "ABCCED", 返回 true
给定 word = "SEE", 返回 true
给定 word = "ABCB", 返回 false
```

提示:

- board 和 word 中只包含大写和小写英文字母。
- 1 <= board.length <= 200
- 1 <= board[i].length <= 200
- 1 <= word.length <= 10^3

2. 简单实现

```
class Solution {
public:
    vector<vector<bool>> visited;
    int m,n;
    bool dfs(vector<vector<char>>& board, int x, int y, string& word, int idx){
        if(idx == word.size()) return true;
        if(x-1 >= 0 && !visited[x-1][y] && board[x-1][y] == word[idx]){
            visited[x-1][y] = true;
            if(dfs(board, x-1, y, word, idx+1)) return true;
            visited[x-1][y] = false;
```

```
        }
        if(x+1 < m && !visited[x+1][y] && board[x+1][y] == word[idx]){
            visited[x+1][y] = true;
            if(dfs(board, x+1, y, word, idx+1)) return true;
            visited[x+1][y] = false;
        }
        if(y-1 >= 0 && !visited[x][y-1] && board[x][y-1] == word[idx]){
             visited[x][y-1] = true;
             if(dfs(board, x, y-1, word, idx+1)) return true;
             visited[x][y-1] = false;
        }
        if(y+1 < n && !visited[x][y+1] && board[x][y+1] == word[idx]){
            visited[x][y+1] = true;
            if(dfs(board, x, y+1, word, idx+1)) return true;
            visited[x][y+1] = false;
        }
        return false;
    }
    bool exist(vector<vector<char>>& board, string word) {
        m = board.size();
        if(m == 0) return false;
        n = board[0].size();
        if(n == 0) return false;
        if(word.size() <= 0 || word.size() > m*n) return false;
        visited = vector<vector<bool>>(m, vector<bool>(n, false));
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(board[i][j] == word[0]){//找到一个可能的开头，以此为起点BFS
                    visited[i][j] = true;
                    if(dfs(board, i, j, word, 1))
                        return true;
                    visited[i][j] = false;
                }
        return false;
    }
};
```

98. 验证二叉搜索树(中等)

1. 题目描述

   给定一个二叉树，判断其是否是一个有效的二叉搜索树。

   假设一个二叉搜索树具有如下特征：

   - 节点的左子树只包含小于当前节点的数。
   - 节点的右子树只包含大于当前节点的数。
   - 所有左子树和右子树自身必须也是二叉搜索树。

```
示例 1:
输入:
    2
   / \
```

```
  1   3
输出: true

示例 2:
输入:
    5
   / \
  1   4
     / \
    3   6
输出: false
解释: 输入为: [5,1,4,null,null,3,6]。
     根节点的值为 5 ，但是其右子节点值为 4 。
```

2. 简单实现

```
class Solution {
public:
    bool judge(TreeNode* root, long& pre){
        if(root->left && !judge(root->left, pre)) return false;
        if(pre >= root->val) return false;
        pre = root->val;
        if(root->right && !judge(root->right, pre)) return false;
        return true;
    }
    bool isValidBST(TreeNode* root) {
        if(!root) return true;
        long pre = long(INT_MIN) - 1;
        return judge(root, pre);
    }
};
```

198. 打家劫舍(简单)

1. 题目描述

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。

```
示例 1:
输入: [1,2,3,1]
输出: 4
解释: 偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
     偷窃到的最高金额 = 1 + 3 = 4 。

示例 2:
输入: [2,7,9,3,1]
输出: 12
解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9), 接着偷窃 5 号房屋 (金额 = 1)。
     偷窃到的最高金额 = 2 + 9 + 1 = 12 。
```

2. 简单实现

   第一反应容易写成奇数位和与偶数位和的比较,这里容易犯错

```cpp
class Solution {
public:
int rob(vector<int>& nums){
    if(nums.empty()) return 0;
    if(nums.size() < 2) return nums[0];
    vector<int> dp(nums.size(), 0);
    dp[0] = nums[0];
    dp[1] = nums[1] > nums[0] ? nums[1] : nums[0];
    for(int i = 2; i < nums.size(); ++i)
        dp[i] = max(nums[i] + dp[i-2], dp[i-1]);
    return dp[nums.size() - 1];
    }
};
```

133. 克隆图(中等)

1. 题目描述

   给你无向 连通 图中一个节点的引用，请你返回该图的 深拷贝（克隆）。

   图中的每个节点都包含它的值 val（int） 和其邻居的列表（list[Node]）。

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

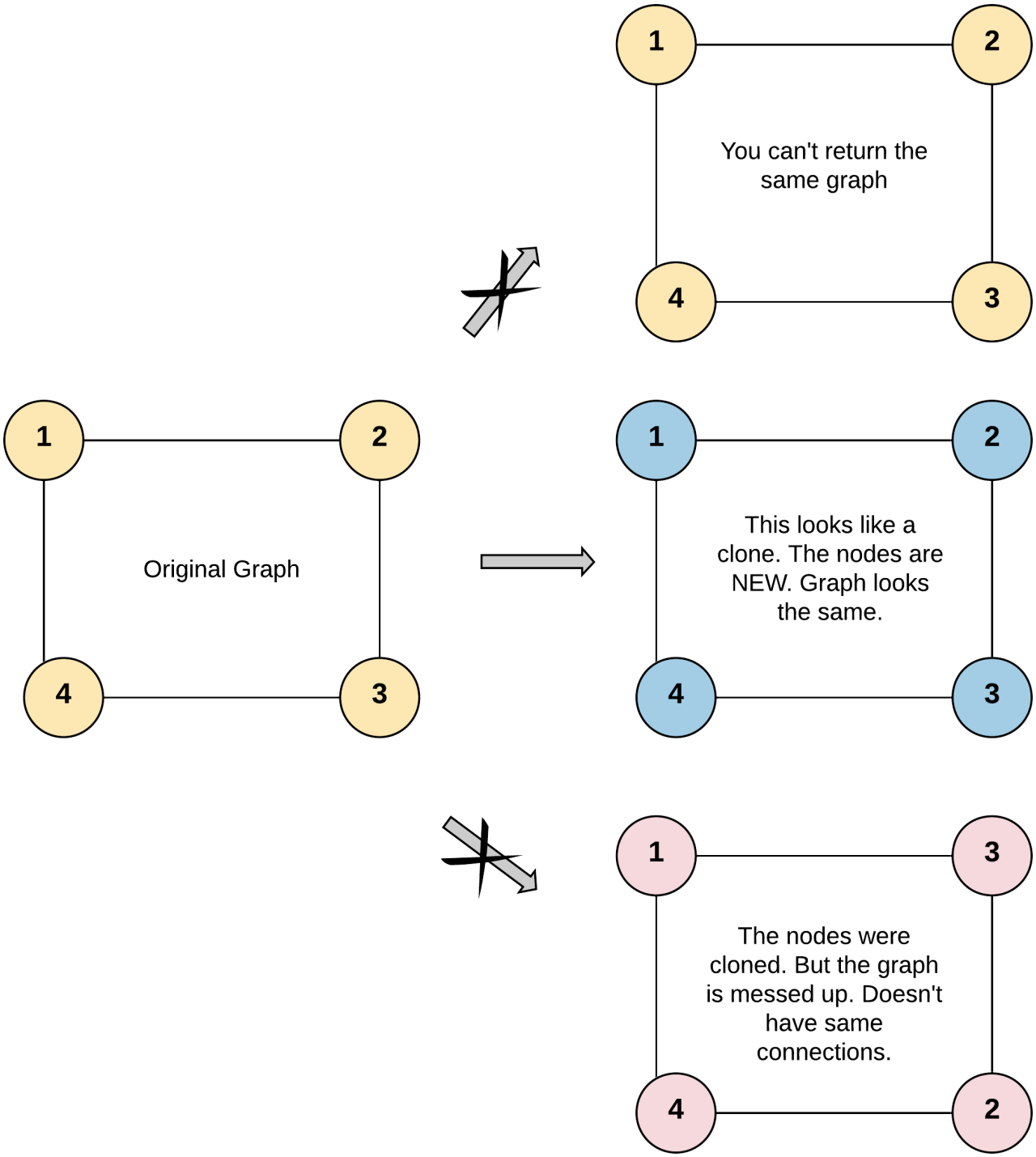
测试用例格式:

简单起见，每个节点的值都和它的索引相同。例如，第一个节点值为 1（val = 1），第二个节点值为 2（val = 2），以此类推。该图在测试用例中使用邻接列表表示。

邻接列表 是用于表示有限图的无序列表的集合。每个列表都描述了图中节点的邻居集。

给定节点将始终是图中的第一个节点（值为 1）。你必须将 给定节点的拷贝 作为对克隆图的引用返回。

示例 1:



输入：adjList = [[2,4],[1,3],[2,4],[1,3]] 输出：[[2,4],[1,3],[2,4],[1,3]] 解释： 图中有 4 个节点。 节点 1 的值是 1，它有两个邻居：节点 2 和 4 。 节点 2 的值是 2，它有两个邻居：节点 1 和 3 。 节点 3 的值是 3，它有两个邻居：节点 2 和 4 。 节点 4 的值是 4，它有两个邻居：节点 1 和 3 。

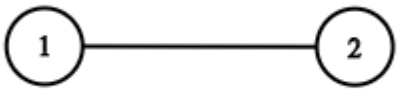
示例 2：

输入：adjList = [[]] 输出：[[]] 解释：输入包含一个空列表。该图仅仅只有一个值为 1 的节点，它没有任何邻居。

示例 3：

输入：adjList = [] 输出：[] 解释：这个图是空的，它不含任何节点。

示例 4：



输入：adjList = [[2],[1]] 输出：[[2],[1]]

提示：

- 节点数不超过 100 。
- 每个节点值 Node.val 都是唯一的，1 <= Node.val <= 100。
- 无向图是一个简单图，这意味着图中没有重复的边，也没有自环。
- 由于图是无向的，如果节点 p 是节点 q 的邻居，那么节点 q 也必须是节点 p 的邻居。
- 图是连通图，你可以从给定节点访问到所有节点。

2. 简单实现

   关键在于用map对新旧Node做映射

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> neighbors;
    Node() {}
    Node(int _val, vector<Node*> _neighbors) {
        val = _val;
        neighbors = _neighbors;
    }
};
*/
```

```
class Solution {
public:
    unordered_map<Node*,Node*> mp;
    Node* cloneGraph(Node* node) {
        if(!node)   return NULL;
        if(mp.find(node) != mp.end())  return mp[node];
        Node* c = new Node(node -> val, {});
        mp[node] = c;
        for(int i = 0; i < node -> neighbors.size(); ++ i)
            c -> neighbors.push_back(cloneGraph(node -> neighbors[i]));
        return c;
    }
};
```

48. 旋转图像(中等)

1. 题目描述

   给定一个 n × n 的二维矩阵表示一个图像。

   将图像顺时针旋转 90 度。

   说明：你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

```
示例 1:
给定 matrix =
[
  [1,2,3],
  [4,5,6],
  [7,8,9]
],
原地旋转输入矩阵，使其变为:
[
  [7,4,1],
  [8,5,2],
  [9,6,3]
]

示例 2:
给定 matrix =
[
  [ 5, 1, 9,11],
  [ 2, 4, 8,10],
  [13, 3, 6, 7],
  [15,14,12,16]
],
原地旋转输入矩阵，使其变为:
[
  [15,13, 2, 5],
  [14, 3, 4, 1],
  [12, 6, 8, 9],
```

```
  [16, 7,10,11]
]
```

2. 简单实现

   旋转=对角线翻转+水平翻转

```
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int len = matrix.size();
        if(len <= 1) return;
        for(int i = 0; i < len; i++)
            for(int j = i+1; j < len; j++)
                swap(matrix[i][j], matrix[j][i]);
        for(int i = 0; i < len; i++)
            for(int j = 0; j < len/2; j++)
                swap(matrix[i][j], matrix[i][len-1-j]);
    }
};
```

269. 火星词典(困难)

要会员

347. 前K个高频元素(中等)

1. 题目描述

   给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

```
示例 1:
输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]

示例 2:
输入: nums = [1], k = 1
输出: [1]
```

提示：

- 你可以假设给定的 k 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
- 你的算法的时间复杂度必须优于 O(n log n) , n 是数组的大小。
- 题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。
- 你可以按任意顺序返回答案。

2. 简单实现

   先计数再用最小堆排序, O(klogk); pair<>也可以排序,按先first后second的顺序比较大小

```
class Solution {
```

```
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> record;  //(元素，频率)
        //遍历数组，录入频率
        for(int i = 0; i < nums.size(); i++)
            record[nums[i]]++;
        int n = record.size();

        //扫描record。维护当前出现频率最多的k个元素
        //最小堆。如果当前元素的频率大于优先队列中最小频率元素的频率，则替换
        //优先队列中，按频率排序，所以数据对是(频率，元素)形式
        priority_queue< pair<int,int> , vector< pair<int,int> >, greater<
pair<int,int> > > pq;
        for(auto iter = record.begin(); iter != record.end(); iter++){
            if(k == pq.size()){ //队列已满
                if(iter->second > pq.top().first){
                    pq.pop();
                    pq.push(make_pair(iter->second,iter->first));
                }
            }
            else
                pq.push(make_pair(iter->second,iter->first));
        }
        vector<int> result(k);
        int idx = k - 1;
        while(!pq.empty()){
            result[idx--] = pq.top().second;
            pq.pop();
        }
        return result;
    }
};
```

317. 离建筑物最近的距离(困难)

要会员

857. 雇佣K名工人的最低成本(困难)

1. 题目描述

   有 N 名工人。 第 i 名工人的工作质量为 quality[i] ，其最低期望工资为 wage[i] 。

   现在我们想雇佣 K 名工人组成一个工资组。在雇佣 一组 K 名工人时，我们必须按照下述规则向他们支付工资：

   1. 对工资组中的每名工人，应当按其工作质量与同组其他工人的工作质量的比例来支付工资。
   2. 工资组中的每名工人至少应当得到他们的最低期望工资。

   返回组成一个满足上述条件的工资组至少需要多少钱。

```
示例 1:
输入: quality = [10,20,5], wage = [70,50,30], K = 2
输出: 105.00000
解释: 我们向 0 号工人支付 70，向 2 号工人支付 35。

示例 2:
输入: quality = [3,1,10,10,1], wage = [4,8,2,2,7], K = 3
输出: 30.66667
解释: 我们向 0 号工人支付 4，向 2 号和 3 号分别支付 13.33333。
```

提示：

- 1 <= K <= N <= 10000，其中 N = quality.length = wage.length
- 1 <= quality[i] <= 10000
- 1 <= wage[i] <= 10000
- 与正确答案误差在 10^-5 之内的答案将被视为正确的。

2. 简单实现
    - 先思考给定k个工人,如何计算所需的工资: 为了满足所有人的最低期望工资,最少要工资为`所有工人单位质量期望工资(quality/wage)的最大值*所有人质量总和`
    - 因此,可以按单位质量期望工资(quality/wage)从小到大排序,对于第i>=k人,在前面选k-1个质量总和最低的人与第i人一起,为当前的工资最小值
    - 因此用大根堆动态维护质量最低的k-1个人的质量,然后遍历

```cpp
class Solution {
public:
    static bool cmp(pair<double, int>& a, pair<double, int>& b){
        return a.first <= b.first;
    }
    double mincostToHireWorkers(vector<int>& quality, vector<int>& wage, int K) {
        int n = quality.size();
        double ans = 1e20;
        if(K == 1){//K为1时直接选工资最低的
            for(int i = 0; i < n; i++)
                ans = min(ans, double(wage[i]));
            return ans;
        }
        vector<pair<double, int>> cost(n);//记录<单位质量期望工资, 索引>
        for(int i = 0; i < n; i++)
            cost[i] = make_pair(double(wage[i])/quality[i], i);
        sort(cost.begin(), cost.end());//排序
        priority_queue<int> q;//大跟堆
        int qual_sum = 0;//当前k-1个最小的质量和
        for(int i = 0; i < K-1; i++){
            q.push(quality[cost[i].second]);
            qual_sum += quality[cost[i].second];
        }
        for(int i = K-1; i < n; i++){
            int cur_q = quality[cost[i].second];
            ans = min(ans, cost[i].first*(qual_sum+cur_q));
            if(cur_q < q.top()){//当前的质量更低
                qual_sum = qual_sum - q.top() + cur_q;
```

```
                q.pop();
                q.push(cur_q);
            }
        }
        return ans;
    }
};
```

14. 最长公共前缀(简单)

1. 题目描述

   编写一个函数来查找字符串数组中的最长公共前缀。如果不存在公共前缀，返回空字符串 ""。

```
示例 1:
输入: ["flower","flow","flight"]
输出: "fl"

示例 2:
输入: ["dog","racecar","car"]
输出: ""
解释: 输入不存在公共前缀。
```

   说明: 所有输入只包含小写字母 a-z 。

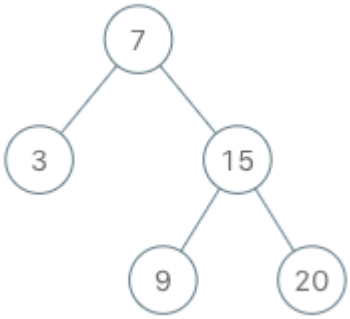
2. 简单实现

```
class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        string ans = "";
        int idx = 0;
        int size = strs.size();
        if(size == 0) return "";
        if(size == 1) return strs[0];
        while(1){
            if(idx >= strs[0].size())
                break;
            char cur = strs[0][idx];
            int i = 1;
            for(; i < size; i++){
                if(idx >= strs[i].size() || strs[i][idx] != cur)
                    break;
            }
            if(i < size)
                break;
            ans += cur;
            idx++;
        }
        return ans;
    }
};
```

173. 二叉搜索树迭代器(中等)

1. 题目描述

   实现一个二叉搜索树迭代器。你将使用二叉搜索树的根节点初始化迭代器。调用 next() 将返回二叉搜索树中的下一个最小的数。




```
示例：
BSTIterator iterator = new BSTIterator(root);
iterator.next();    // 返回 3
iterator.next();    // 返回 7
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 9
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 15
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 20
iterator.hasNext(); // 返回 false
```

提示：

- next() 和 hasNext() 操作的时间复杂度是 O(1)，并使用 O(h) 内存，其中 h 是树的高度。
- 你可以假设 next() 调用总是有效的，也就是说，当调用 next() 时，BST 中至少存在一个下一个最小的数。

2. 简单实现

   非递归的中序遍历,保存栈,next()最差情况是O(h),但不总是,所以近似O(1)

```
class BSTIterator {
public:
    stack<TreeNode*> s;
    BSTIterator(TreeNode* root) {
        while(root){
            s.push(root);
            root = root->left;
        }
    }
    /** @return the next smallest number */
    int next() {
        TreeNode* cur = s.top();
        s.pop();
```

```
            int ans = cur->val;
            if(cur->right){
                cur = cur->right;
                while(cur){
                    s.push(cur);
                    cur = cur->left;
                }
            }
            return ans;
        }
        /** @return whether we have a next smallest number */
        bool hasNext() {
            return !s.empty();
        }
    };
```

246. 中心对称数(简单)

要会员

833. 字符串中的查找与替换(中等)

1. 题目描述

   对于某些字符串 S，我们将执行一些替换操作，用新的字母组替换原有的字母组（不一定大小相同）。

   每个替换操作具有 3 个参数：起始索引 i，源字 x 和目标字 y。规则是如果 x 从原始字符串 S 中的位置 i 开始，那么我们将用 y 替换出现的 x。如果没有，我们什么都不做。

   举个例子，如果我们有 S = “abcd” 并且我们有一些替换操作 i = 2，x = “cd”，y = “ffff”，那么因为 “cd” 从原始字符串 中的位置 2 开始，我们将用 “ffff” 替换它。

   再来看 S = “abcd” 上的另一个例子，如果我们有替换操作 i = 0，x = “ab”，y = “eee”，以及另一个替换操作 i = 2，x = “ec”，y = “ffff”，那么第二个操作将不执行任何操作，因为原始字符串中 S[2] = 'c'，与 x[0] = 'e' 不匹配。

   所有这些操作同时发生。保证在替换时不会有任何重叠： S = "abc", indexes = [0, 1], sources = ["ab","bc"] 不是有效的测试用例。

```
 示例 1:
输入: S = "abcd", indexes = [0,2], sources = ["a","cd"], targets = ["eee","ffff"]
输出: "eeebffff"
解释:
"a" 从 S 中的索引 0 开始，所以它被替换为 "eee"。
"cd" 从 S 中的索引 2 开始，所以它被替换为 "ffff"。

示例 2:
输入: S = "abcd", indexes = [0,2], sources = ["ab","ec"], targets = ["eee","ffff"]
输出: "eeecd"
解释:
"ab" 从 S 中的索引 0 开始，所以它被替换为 "eee"。
"ec" 没有从原始的 S 中的索引 2 开始，所以它没有被替换。
```

提示：

- 0 <= indexes.length = sources.length = targets.length <= 100
- 0 < indexes[i] < S.length <= 1000
- 给定输入中的所有字符都是小写字母。

2. 简单实现

   按indexes排序,然后从后往前依次替换,可以防止替换后影响后续替换indexex的问题

```
class Node{//某个替换操作
public:
    int indexe;//替换的起始位置
    int idx;//indexe在indexes中的索引
    Node(int a, int b) : indexe(a), idx(b) {};
};

class Solution {
public:
    static bool cmp(Node* a, Node* b){
        return a->indexe < b->indexe;
    }
    string findReplaceString(string S, vector<int>& indexes, vector<string>&
sources, vector<string>& targets) {
        int len = S.size();
        vector<Node*> v(indexes.size());
        for(int i = 0; i < indexes.size(); i++)
            v[i] = new Node(indexes[i], i);
        sort(v.begin(), v.end(), cmp);//排序
        for(int i = indexes.size()-1; i >= 0; i--){//从后往前
            int cur_len = sources[v[i]->idx].size();
            if(v[i]->indexe+cur_len<=len
               && S.substr(v[i]->indexe, cur_len)==sources[v[i]->idx])
                S.replace(v[i]->indexe, cur_len, targets[v[i]->idx]);//替换
        }
        return S;
    }
};
```

138. 复制带随即指针的链表(中等)

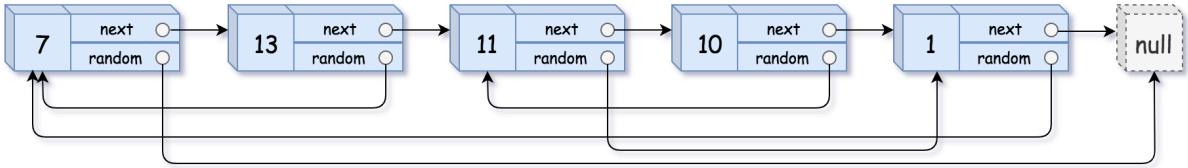
1. 题目描述

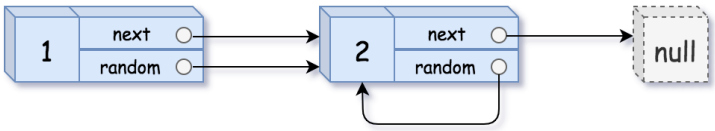
   给定一个链表，每个节点包含一个额外增加的随机指针，该指针可以指向链表中的任何节点或空节点。

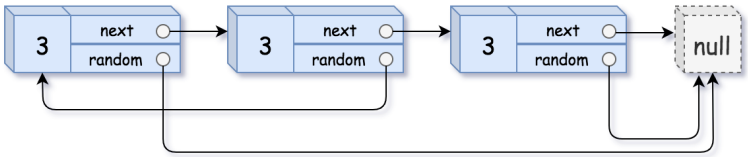
   要求返回这个链表的 深拷贝。

   我们用一个由 n 个节点组成的链表来表示输入/输出中的链表。每个节点用一个 [val, random_index] 表示：

   val：一个表示 Node.val 的整数。 random_index：随机指针指向的节点索引（范围从 0 到 n-1）；如果不指向任何节点，则为 null 。








```
示例 1:
输入: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出: [[7,null],[13,0],[11,4],[10,2],[1,0]]

示例 2:
输入: head = [[1,1],[2,1]]
输出: [[1,1],[2,1]]

示例 3:
输入: head = [[3,null],[3,0],[3,null]]
输出: [[3,null],[3,0],[3,null]]

示例 4:
输入: head = []
输出: []
解释: 给定的链表为空（空指针），因此返回 null。
```

提示:

- -10000 <= Node.val <= 10000
- Node.random 为空（null）或指向链表中的节点。
- 节点数目不超过 1000 。

2. 简单实现

   两次遍历

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* next;
```

```
    Node* random;

    Node() {}

    Node(int _val, Node* _next, Node* _random) {
        val = _val;
        next = _next;
        random = _random;
    }
};
*/
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if(!head) return NULL;
        map<Node*, Node*> m;//old->new
        //复制所有节点，构建map
        Node* ans = new Node(head->val, NULL, NULL);
        Node* old = head;
        Node* cur = ans;
        m[old] = cur;
        while(old->next){
            old = old->next;
            cur->next = new Node(old->val, NULL, NULL);
            cur = cur->next;
            m[old] = cur;
        }
        //构建random
        old = head;
        cur = ans;
        while(old){
            cur->random = m[old->random];
            old = old->next;
            cur = cur->next;
        }
        return ans;
    }
};
```