LeetCode 53. 最大子序和

53. 最大子序和

难度 简单 1716

给定一个整数数组 `nums` ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例:

```
输入：[-2,1,-3,4,-1,2,1,-5,4],
输出：6
解释：连续子数组 [4,-1,2,1] 的和最大，为 6。
```

进阶:

如果你已经实现复杂度为 O(*n*) 的解法，尝试使用更为精妙的分治法求解。

算法1

(动态规划) $O(n)$

1. 设 $f(i)$ 表示以第 i 个数字为结尾的最大连续子序列的 和 是多少。
2. 初始化 $f(0) = nums[0]$。
3. 转移方程 $f(i) = \max(f(i-1) + nums[i], nums[i])$。可以理解为当前有两种决策，一种是将第 i 个数字和前边的数字拼接起来；另一种是第 i 个数字单独作为一个新的子序列的开始。
4. 最终答案为 $ans = \max(f(k)), 0 \le k < n$。

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n = nums.size(), ans;
        vector<int> f(n);
        f[0] = nums[0];
        ans = f[0];
        for (int i = 1; i < n; i++) {
            f[i] = max(f[i - 1] + nums[i], nums[i]);
            ans = max(ans, f[i]);
        }
        return ans;
    }
};
********************************************************************************
/*
求出所有子段和的最大值
枚举所有起点、终点

状态表示：f[i]表示所有以i结尾的字段的最大值
状态计算：  f[i] = max(f(i - 1),0) + nums[i];
*/
class Solution {
public:
```

```
    int maxSubArray(vector<int>& nums) {
        int res = INT_MIN,last = 0;
        for(int i = 0;i < nums.size();i ++)
        {
            int now = max(last,0) + nums[i];
            res = max(res,now);
            last = now;
        }

        return res;
    }
};
```

算法2

(分治) $O(nlogn)$

1. 考虑区间 `[l, r]` 内的答案如何计算，令 `mid = (l + r) / 2`，则该区间的答案由三部分取最大值，分别是：
   (1). 区间 `[l, mid]` 内的答案（递归计算）。
   (2). 区间 `[mid + 1, r]` 内的答案（递归计算）。
   (3). 跨越 `mid` 和 `mid+1` 的连续子序列。
2. 其中，第(3)部分只需要从 `mid` 开始向 `l` 找连续的最大值，以及从 `mid+1` 开始向 `r` 找最大值即可，在线性时间内可以完成。
3. 递归终止条件显然是 `l==r` ，此时直接返回 `nums[l]` 。

```
class Solution {
public:
    int calc(int l, int r, vector<int>& nums) {
        if (l == r)
            return nums[l];
        int mid = (l + r) >> 1;
        int lmax = nums[mid], lsum = 0, rmax = nums[mid + 1], rsum = 0;
        for (int i = mid; i >= l; i--) {
            lsum += nums[i];
            lmax = max(lmax, lsum);
        }
        for (int i = mid + 1; i <= r; i++) {
            rsum += nums[i];
            rmax = max(rmax, rsum);
        }
        return max(max(calc(l, mid, nums), calc(mid + 1, r, nums)), lmax +
rmax);
    }

    int maxSubArray(vector<int>& nums) {
        int n = nums.size();
        return calc(0, n - 1, nums);
    }
};
```

LeetCode 120. 三角形最小路径和

120. 三角形最小路径和

难度 中等 334

给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。

例如，给定三角形：

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

自顶向下的最小路径和为 `11`（即，**2** + **3** + **5** + **1** = 11）。

说明：

如果你可以只使用 *O*(*n*) 的额外空间（*n* 为三角形的总行数）来解决这个问题，那么你的算法会很加分。

动态规划 时间$O(n^2)$ 空间$O(1)$

点(i,j)的下一行的相邻数字是$(i+1,j)$和$(i+1,j+1)$。

$f(i,j)$表示从下往上走到位置(i,j)时的最小路径和，计算方式/状态转移方程是

$$f(i,j)=(i,j)+min(f(i+1,j),f(i+1,j+1))$$

复杂度分析：

直接把$f(i,j)$存在位置(i,j)处，不使用额外空间，因此空间复杂度为$O(1)$。

两层for loop，第一次竖着遍历，第二次横着遍历，时间复杂度为$O(n^2)$。

```
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        for (int i = triangle.size()-2 ; i>=0 ; --i){ //从倒数第二行开始往上
            for (int j = 0 ; j < i+1; ++j){
                triangle[i][j] += min(triangle[i+1][j], triangle[i+1][j+1]) ;
            }
        }
        return triangle[0][0];
    }
};
**********************************************************************
/*
状态表示：f[i,j]    所有从起点走到第i行第j列的路径和的最小值
状态计算：
1.最后一步从左上下来:f[i - 1,j - 1]
2.最后一步从右上下来:f[i - 1,j]
*/
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
```

```
        int n = triangle.size();
        vector<vector<long long>> f(n,vector<long long>(n));
        f[0][0] = triangle[0][0];
        for(int i = 1;i < n;i ++)
            for(int j = 0;j <= i; j ++)
            {
                f[i][j] = INT_MAX;
                if(j > 0) f[i][j] = min(f[i][j],f[i - 1][j - 1] + triangle[i]
[j]);//可以从左边更新
                if(j < i) f[i][j] = min(f[i][j],f[i - 1][j] + triangle[i][j]);//
可以从右边更新
            }
        long long res= INT_MAX;
        for(int i = 0;i < n;i ++) res = min(res,f[n - 1][i]);//枚举最后一行的状态
        return res;
    }
};
**********************************************************
/*
二维数组>>>>>>>>>滚动数组
因为f[i]只依赖f[i - 1]这层所以可以用滚动数组
把f[][]前一个空全部 & 1 ！！！就可以变成滚动数组
*/
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        int n = triangle.size();
        vector<vector<long long>> f(n,vector<long long>(n));
        f[0][0] = triangle[0][0];
        for(int i = 1;i < n;i ++)
            for(int j = 0;j <= i; j ++)
            {
                f[i & 1][j] = INT_MAX;
                if(j > 0) f[i & 1][j] = min(f[i & 1][j],f[i - 1 & 1][j - 1] +
triangle[i][j]);//可以从左边更新
                if(j < i) f[i & 1][j] = min(f[i & 1][j],f[i - 1 & 1][j] +
triangle[i][j]);//可以从右边更新
            }
        long long res= INT_MAX;
        for(int i = 0;i < n;i ++) res = min(res,f[n - 1 & 1][i]);//枚举最后一行的状
态
        return res;
    }
};
```

LeetCode 63. 不同路径 II

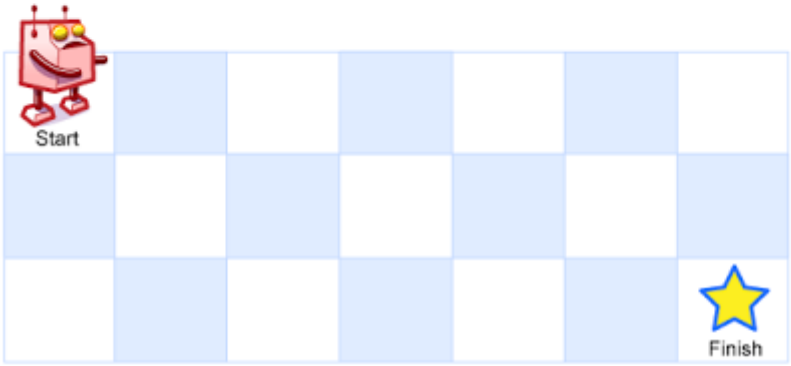
63. 不同路径 II

难度 中等 240

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

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

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径?



网格中的障碍物和空位置分别用 `1` 和 `0` 来表示。

说明： *m* 和 *n* 的值均不超过 100。

示例 1:

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

```
/*
类似于62题Unique Paths，每一个网格都可以由该网格左边或上边的网格转移过来，因此到达某一点的路径数等于到达它上一点的路径数与它左边的路径数之和，不同的是，当某个网格有障碍时，到达该网格的路径数维0。这还是一个递推问题，考虑用动态规划。动态规划数组dp[i][j] = 起点到点(i, j)的路径总数。于是我们就得到递推关系式：当网格为0时，dp[i][j] = dp[i][j-1] + dp[i-1][j]；当网格为1（说明该网格是障碍物），dp[i][j]=0。
*/
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        if(m == 0)return 0;
        int n = obstacleGrid[0].size();
```

```
        if(n == 0)return 0;
        int dp[101][101];
        memset(dp, 0, sizeof(dp));
        dp[0][0] = 1 - obstacleGrid[0][0];
        for(int i = 0; i < m; i++){
            for(int j = 0; j< n; j++){
                if(obstacleGrid[i][j] == 1)
                    dp[i][j] = 0;
                else{
                    if(i > 0)
                        dp[i][j] += dp[i-1][j];
                    if(j > 0)
                        dp[i][j] += dp[i][j-1];
                }
            }
        }
        return dp[m-1][n-1];
    }
};
************************************
/*
状态表示:f[i][j]   从起点到(i,j)的路径
状态计算:
1.最后一步往下走:f[i - 1][j]
2.最后一步往右走:f[i,j - 1]
*/
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& g) {
        int n = g.size(),m = g[0].size();
        vector<vector<long long>> f(n,vector<long long>(m));
        for(int i = 0;i < n;i ++)
            for(int j = 0;j < m;j ++)
            {
                if(g[i][j]) continue;//障碍物
                if(!i && !j) f[i][j] = 1;//左上角
                if(i > 0) f[i][j] += f[i - 1][j];//不是第一行可以从上面过来来
                if(j > 0) f[i][j] += f[i][j - 1];//不是第一列可以从左面过来来
            }
        return f[n - 1][m - 1];
    }
};
```

LeetCode 91. 解码方法

91. 解码方法

难度 中等 311

一条包含字母 `A-Z` 的消息通过以下方式进行了编码：

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

给定一个只包含数字的**非空**字符串，请计算解码方法的总数。

示例 1:

```
输入：  "12"
输出：  2
解释：  它可以解码为 "AB"（1 2）或者 "L"（12）。
```

示例 2:

```
输入：  "226"
输出：  3
解释：  它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF"
(2 2 6) 。
```

(动态规划) $O(n)$

这道题目可以用动态规划来做。
状态表示：$f[i]$ 表示前 i 个数字共有多少种解码方式。
初始化：0个数字解码的方案数1，即 $f[0]=1$。
状态转移：$f[i]$ 可以表示成如下两部分的和：

- 如果第 i 个数字不是0，则 i 个数字可以单独解码成一个字母，此时的方案数等于用前 $i-1$ 个数字解码的方案数，即 $f[i-1]$；
- 如果第 $i-1$ 个数字和第 i 个数字组成的两位数在 10 到 26 之间，则可以将这两位数字解码成一个字符，此时的方案数等于用前 $i-2$ 个数字解码的方案数，即 $f[i-2]$；

时间复杂度分析：状态数是 n 个，状态转移的时间复杂度是 $O(1)$，所以总时间复杂度是 $O(n)$。

```cpp
/*
数字变字符串 解码

状态表示:f[i]    所有由前i个数字解码得到的字符串
状态计算:
1.最后一步由最后一个字母s[i]解码     f[i - 1]
2.最后一步由倒数连个字母s[i - 1],s[i]解码     f[i - 2]
*/
class Solution {
public:
    int numDecodings(string s) {
        if (s.empty()) return 0;
        int n = s.size();
        vector<int> f(n + 1);
        f[0] = 1;
```

```
        for (int i = 1; i <= n; i ++ )
        {
            if (s[i - 1] < '0' || s[i - 1] > '9')
                return 0;
            f[i] = 0;
            if (s[i - 1] != '0') f[i] = f[i - 1];
            if (i > 1)
            {
                int t = (s[i-2]-'0')*10+s[i-1]-'0';
                if (t >= 10 && t <= 26)
                    f[i] += f[i - 2];
            }
        }
        return f[n];
    }
};
*****************************************************************
class Solution {
public:
    int numDecodings(string s) {
        int n = s.size();
        vector<int> f(n + 1);
        f[0] = 1;//边界
        for(int i = 1;i <= n;i ++)
        {
            if(s[i - 1] != '0') f[i] += f[i - 1];
            if(i >= 2)
            {
                int t = (s[i - 2] - '0') * 10 + s[i - 1] - '0';
                if(t >= 10 && t <= 26) f[i] += f[i - 2];//t是位
            }
        }
        return f[n];
    }
};
```

LeetCode 198.打家劫舍 !!!

198. 打家劫舍

难度 简单 671

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 。
```

算法

(动态规划) $O(n)$

1. 令 `f[i]` 表示盗窃了第 `i` 个房间所能得到的最大收益， `g[i]` 表示不盗窃第 `i` 个房间所能得到的最大收益。
2. `f[i] = g[i - 1] + nums[i]` ， `g[i] = max(f[i - 1], g[i - 1])` 。
3. 初始值 `f[0] = nums[0], g[0] = 0` ，答案为 `max(f[n - 1], g[n - 1])` 。

优化

- 由于每次更新只用到了上一层的信息，故可以优化空间为常数。

时间复杂度

- 状态数为$O(n)$，转移数为$O(1)$，转移时间为$O(1)$，故总时间复杂度为$O(n)$。

```cpp
class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        if (n == 0)
            return 0;

        int f = nums[0], g = 0;

        for (int i = 1; i < n; i++) {
            int last_f = f, last_g = g;
            f = g + nums[i];
```

```
            g = max(last_f, last_g);
        }

        return max(f, g);
    }
};
*****************************************************
/*
状态表示:
f[i]  在前i个数中，所有不选nums[i]的选法的最大值
g[i]  在前i个数中，所有选择nums[i]的选法的最大值
状态计算:
f[i] = max(f[i - 1],g[i -1])
g[i] = f[i - 1] + nums[i - 1]
*/
class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        vector<int> f(n + 1),g(n + 1);
        for(int i = 1;i <= n;i ++)
        {
            f[i] = max(f[i - 1],g[i - 1]);
            g[i] = f[i - 1] + nums[i - 1];
        }
        return max(f[n],g[n]);
    }
};
```

LeetCode 300.最长上升子序列

300. 最长上升子序列

难度 中等 552

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例:

```
输入: [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。
```

说明:

- 可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
- 你算法的时间复杂度应该为 $O(n^2)$ 。

进阶: 你能将算法的时间复杂度降低到 $O(n \log n)$ 吗?

算法1

（动态规划）$O(n^2)$

用数组dp[i]记录以nums[i]结尾（即nums[i]为最后一个数字）的最长递增子序列的长度，则递推方程为$dp[i]=max(dp[j]+1)$，其中要求$1\leq j<i$且$nums[j]<nums[i]$。

时间复杂度分析：对每个$i(1\leq i\leq n)$，都需要从1遍历到i，则时间复杂度为$O(n^2)$，空间复杂度的话需要一个额外的dp数组，空间复杂度为$O(n^2)$。

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        if(nums.size()==0)
            return 0;
        vector<int> dp(nums.size(),1);
        int res = 1;
        for(int i= 1;i<nums.size();i++){
            for(int j = 0;j<i;j++){
                if(nums[i]>nums[j])
                    dp[i] = max(dp[i],dp[j]+1);
            }
            if(dp[i]>res)
                res = dp[i];
        }
        return res;
    }
};
```

算法2

（动态规划 二分查找）$O(nlogn)$

在解法1中，对于每个i，都需要遍历dp[1]到dp[i-1]，但其实是不必要的，因为$dp[i]=max(dp[j]+1),1\leq j<i$且$nums[j]<nums[i]$，那么对于j而言，希望dp[j]越大越好，nums[j]越小越好，那么在数组中，若$nums[p]\geq nums[q]$但$dp[p]\leq dp[q]$，那么对于求dp[i]来说，nums[p]是没有用的。

例如在数组[1,2,5,3,7,8]中，nums[2]=5,dp[2]=3（序列[1,2,5]）,nums[3]=3,dp[3]=3（序列[1,2,3]），那么对于数组中下一个数字7来说，下标为2的5就是没有用的，因为存在下标3，使得$nums[3]<nums[2]$且$dp[3]\geq dp[2]$，那么我们就不用考虑下标为2的数字5了。

因此我们可以维护一个新的数组help，help[i]表示最长子序列长度为i时的最小的结尾num值（例如在数组[1,2,5,3,7]中，长度为3的子序列有[1,2,3]，[1,2,5]，[2,5,7]三个，取最小的结尾数字，那么help[3]=3）。

对于数字m，我们只需要找到找到最大的满足help[j] < m的j，那么就意味着把m接在help[j]这个数字后面就可以了，这个子序列的长度是j+1，同时我们需要判断m是否比原来的help[j+1]小，如果m更小的话就需要更新help[j+1]=m，这一定是一个单调递增的数组（因为要求子序列必须是单调递增的，那么序列长度为i+1的子序列最后一个数字一定比序列长度为i的子序列最后一个数字要大），那么我们就可以通过二分查找来找到满足条件的j，因此把原来查找的复杂度由$O(n)$降为$O(logn)$。

时间复杂度分析：如上分析，对于每个m，复杂度为$O(logn)$，有n个数字，因此时间复杂度为$O(nlogn)$，需要额外的help数组，空间复杂度为$O(n)$。

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        if(nums.size()==0)
```

```
            return 0;
        vector<int> help(nums.size()+1,0);
        help[1] = nums[0];
        int maxlen = 1;
        for(int i= 1;i<nums.size();i++){
            int left = 1;
            int right = maxlen;
            while(left<=right){//二分查找
                int mid = (left+right)/2;
                if(help[mid]<nums[i])
                    left = mid+1;
                else
                    right = mid-1;
            }
            help[left] = nums[i];//维护help数组
            if(left>maxlen)//left值是nums[i]的子序列长度
                maxlen=left;
        }
        return maxlen;
    }
};
**********************************************************************
/*
状态表示:f[i]    所有以i结尾的上升子序列长度的最大值
状态计算:
倒数第二个数是空、倒数第二个数是d第0个数、倒数第二个数是第1个数......
倒数第二个数的是第j个数的子序列的最大值是f[j]
f[j] + 1

nums[j] < nums[i] 才能接到倒数第二个数后面
*/
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size();
        vector<int> f(n);

        for(int i = 0;i < n;i ++)
        {
            f[i] = 1;//初始只有自己一个字符
            for(int j = 0;j < i;j ++)
                if(nums[j] < nums[i])
                    f[i] = max(f[i],f[j] + 1);
        }

        int res = 0;
        for(int i = 0;i < n;i ++) res = max(res,f[i]);
        return res;
    }
};
```

LeetCode 72. 编辑距离

72. 编辑距离

难度 困难 575

给定两个单词 *word1* 和 *word2*，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：

1. 插入一个字符
2. 删除一个字符
3. 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')
```

算法

(动态规划) $O(n^2)$

经典的编辑距离问题。

状态表示：$f[i,j]$ 表示将 *word1* 的前 i 个字符变成 *word2* 的前 j 个字符，最少需要进行多少次操作。
状态转移，一共有四种情况（假定word的下标从1开始）：

1. 将 $word1[i]$ 删除或在 $word2[j]$ 后面添加 $word1[i]$，则其操作次数等于 $f[i-1,j]+1$；
2. 将 $word2[j]$ 删除或在 $word1[i]$ 后面添加 $word2[j]$，则其操作次数等于 $f[i,j-1]+1$；
3. 如果 $word1[i]=word2[j]$，则其操作次数等于 $f[i-1,j-1]$；
4. 如果 $word1[i]\neq word2[j]$，则其操作次数等于 $f[i-1,j-1]+1$；

```
/*
状态表示:f[i][j]     所有将第一个字符串的前i个字母变成第二个字符串的前j个字母的方案的最小值
状态计算:
1.添加操作:f[i,j - 1] + 1    加之前第一个字符串的前i个字母变成第二个字符串的前j - 1个字母已经匹配
2.删除操作:f[i - 1,j] + 1    删之前第一个字符串的前i - 1个字母变成第二个字符串的前j个字母已经匹配
3.替换操作:
```

```
    f[i - 1][j - 1]     第一个字符串的个第i个字母变和第二个字符串的第j个字母已经相等，不要替换
    f[i - 1][j - 1] + 1     第一个字符串的个第i个字母变和第二个字符串的第j个字母不相等，需要替换


四个最小值取min
*/
class Solution {
public:
    int minDistance(string word1, string word2) {
        int n = word1.size(), m = word2.size();
        if (!n || !m) return n + m;
        vector<vector<int>>f =
            vector<vector<int>>(n + 1,vector<int>(m + 1));
        f[0][0] = 0;
        for (int i = 1; i <= n; i ++ ) f[i][0] = i;
        for (int j = 1; j <= m; j ++ ) f[0][j] = j;
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= m; j ++ )
            {
                f[i][j] = f[i - 1][j - 1] + (word1[i - 1] != word2[j - 1]);
                f[i][j] = min(f[i][j], f[i - 1][j] + 1);
                f[i][j] = min(f[i][j], f[i][j - 1] + 1);
            }
        return f[n][m];
    }
};

**********************************************************
class Solution {
public:
    int minDistance(string word1, string word2) {
        int n = word1.size(),m = word2.size();
        vector<vector<int>> f(n + 1,vector<int>(m + 1));
        for(int i = 0;i <= n;i ++) f[i][0] = i;//边界情况，把第一个字符串变成0，要进行i次删除操作
        for(int i = 0;i <= m;i ++) f[0][i] = i;//边界情况，把第二个字符串变成0，要进行i次删除操作

        for(int i = 1;i <= n;i ++)
        {
            for(int j = 1;j <= m;j ++)
            {
                f[i][j] = min(f[i - 1][j],f[i][j - 1]) + 1;
                f[i][j] = min(f[i][j],f[i -1][j - 1] + (word1[i - 1] != word2[j - 1]));
            }
        }
        return f[n][m];
    }
};
```

LeetCode 518. 零钱兑换 II

518. 零钱兑换 II

难度 中等 116

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

示例 1:

```
输入: amount = 5, coins = [1, 2, 5]
输出: 4
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

示例 2:

```
输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额2的硬币不能凑成总金额3。
```

示例 3:

```
输入: amount = 10, coins = [10]
输出: 1
```

注意:

你可以假设:

- 0 <= amount (总金额) <= 5000
- 1 <= coin (硬币面额) <= 5000
- 硬币种类不超过 500 种
- 结果符合 32 位符号整数

算法

(动态规划，完全背包) $O(amount \cdot n)$

1. 将总金额视为背包容量，将硬币的面额视为重量，价值视为 1，此题就是变种的固定容量完全背包问题。
2. 设计状态 $f(j)$ 表示硬币总面额为 j 时的方案数，对于每一种硬币面额 $coins(i)$，j 从 $coins(i)$ 枚举到 $amount$，累计转移 $f(j) = f(j) + f(j - coins(i))$。
3. 初始时 $f(0) = 1$，最终答案为 $f(amount)$。

时间复杂度

- 状态数为 $O(amount)$，转移总数为$O(n)$，每次转移的时间复杂度为 $O(1)$，故总时间复杂度为 $O(amount \cdot n)$。

```
/*
状态表示:f[i][j]     所有由前i种硬币凑出的总钱数等于j的方法的数量
```

```
状态计算:第i类物品选0,1,2,3.....
0个:f[i - 1][j]
t个:f[i - 1][j - t * coins[i]]

f[i,j] = f[i -1,j] + f[i-1,j-c] + f[i-1,j-2c] +....+f[i-1,j-kc]
f[i,j-c] = f[i-1,j-c] + f[i-1,j-2c]+....+f[i-1,j-kc]

优化成一层循环!!!
f[i,j] = f[i-1,j] + f[i,j-c]
优化成一维!!!
f[j] = f[j] + f[j - c]
*/
class Solution {
public:
    int change(int amount, vector<int>& coins) {
        int n = coins.size();
        vector<int> f(amount + 1, 0);
        f[0] = 1;

        for (int i = 0; i < n; i++)
            for (int j = coins[i]; j <= amount; j++)
                f[j] += f[j - coins[i]];

        return f[amount];
    }
};
```

LeetCode 664.奇怪的打印机

664. 奇怪的打印机

难度 困难 38

有台奇怪的打印机有以下两个特殊要求：

1. 打印机每次只能打印同一个字符序列。
2. 每次可以在任意起始和结束位置打印新字符，并且会覆盖掉原来已有的字符。

给定一个只包含小写英文字母的字符串，你的任务是计算这个打印机打印它需要的最少次数。

示例 1:

```
输入："aaabbb"
输出：2
解释：首先打印 "aaa" 然后打印 "bbb"。
```

示例 2:

```
输入："aba"
输出：2
解释：首先打印 "aaa" 然后在第二个位置打印 "b" 覆盖掉原来的字符 'a'。
```

提示: 输入字符串的长度不会超过 100。

算法

(动态规划) $O(n^3)$

1. 状态 $f(i,j)$ 表示打印区间 [i, j] 所需要的最少步数。
2. 初始值 $f(i,i)=1$，$f(i,i-1)=0$；
3. 转移条件，每次扩展到新的长度时，[i, j] 区间的首字符尤为重要，因为区间 `[i, j]` 的某个最优解一定是第一次将首字符全部写满区间 `[i, j]`；
4. 接下来，假设区间中最右侧仍保留着第一次写的首字符的位置为 k。若 k 就是 i，则表示除了首字符之外，区间内其余所有字符都被重新覆盖过，则可以给出的转移为 $f(i,j)=min(f(i,j),1+f(i+1,j))$；若 $k>i$，则可以转移 $f(i,j)=min(f(i,j),f(i,k-1)+f(k+1,j))$，可以分为两段区间求和的原因是，区间 `[i, k - 1]` 仍然可以用首字符按照以上规则进行划分，且不必考虑 k 位置的字符，所以才可以用求出的子结果，不影响整个优化过程；区间 `[k + 1, j]` 则显然是直接利用求出的子结果。
5. 最终答案为 $f(0,n-1)$。

时间复杂度

- 状态数为 $O(n^2)$，转移数为 $O(n)$，故总时间复杂度为 $O(n^3)$。

```
/*
状态表示：f[L,R]    所有将[L,R]染成最终样子的方式的最小值
状态计算：染到L,L+1,L+2.....R
只需L染色:f[L+1,R] + 1
[L,K]染色:f[L,K-1]+f[K+1,R]
*/
class Solution {
public:
    int strangePrinter(string s) {
        int n = s.length();
```

```
        if (n == 0)
            return 0;
        vector<vector<int>> f(n + 1, vector<int>(n + 1));
        for (int i = 0; i <= n; i++) {
            f[i][i] = 1;
            if (i > 0)
                f[i][i - 1] = 0;
        }

        for (int len = 2; len <= n; len++)
            for (int i = 0; i < n - len + 1; i++) {
                int j = i + len - 1;
                f[i][j] = 1 + f[i + 1][j];
                for (int k = i + 1; k <= j; k++)
                    if (s[i] == s[k])
                        f[i][j] = min(f[i][j], f[i][k - 1] + f[k + 1][j]);
            }

        return f[0][n - 1];
    }
};
```

LeetCode 10. 正则表达式匹配

10. 正则表达式匹配

难度 困难 1017

给你一个字符串 `s` 和一个字符规律 `p`，请你来实现一个支持 `'.'` 和 `'*'` 的正则表达式匹配。

```
'.' 匹配任意单个字符
'*' 匹配零个或多个前面的那一个元素
```

所谓匹配，是要涵盖 **整个** 字符串 `s` 的，而不是部分字符串。

说明:

- `s` 可能为空，且只包含从 `a-z` 的小写字母。
- `p` 可能为空，且只包含从 `a-z` 的小写字母，以及字符 `.` 和 `*`。

示例 1:

```
输入：
s = "aa"
p = "a"
输出：false
解释："a" 无法匹配 "aa" 整个字符串。
```

示例 2:

```
输入：
s = "aa"
p = "a*"
输出：true
解释：因为 '*' 代表可以匹配零个或多个前面的那一个元素，
在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a'
重复了一次。
```

算法

(动态规划) $O(nm)$

状态表示：f[i][j]表示p从j开始到结尾，是否能匹配s从i开始到结尾
状态转移：

1. 如果p[j+1]不是通配符 `'*'` ，则f[i][j]是真，当且仅当s[i]可以和p[j]匹配，且f[i+1][j+1]是真；
2. 如果p[j+1]是通配符 `'*'` ，则下面的情况只要有一种满足，f[i][j]就是真；
   - f[i][j+2]是真；
   - s[i]可以和p[j]匹配，且f[i+1][j]是真；

第1种情况下的状态转移很好理解，那第2种情况下的状态转移怎么理解呢？

> 最直观的转移方式是这样的：枚举通配符 `'*'` 可以匹配多少个p[j]，只要有一种情况可以匹配，则f[i][j]就是真；
> 这样做的话，我们发现，f[i][j]除了枚举0个p[j]之外，其余的枚举操作都包含在f[i+1][j]中了，所以我们只需判断
> f[i+1][j]是否为真，以及s[i]是否可以和p[j]匹配即可。

时间复杂度分析：n 表示s的长度，m 表示p的长度，总共 nm 个状态，状态转移复杂度 $O(1)$，所以总时间复杂度是 $O(nm)$.

```
/*
状态表示:f[i,j]表示s的前i个字母和p的前i个字母是否匹配
    1.p[j] != '*',f[i,j]=f[i-1,j-1] && (s[i]和p[j]匹配)
    2.p[j] == '*',需要枚举*表示多少个字母
        f[i,j] = f[i,j-2] || f[i-1,j] && (s[i]和p[j - 1]匹配)
*/
class Solution {
public:
    vector<vector<int>>f;
    int n, m;
    bool isMatch(string s, string p) {
        n = s.size();
        m = p.size();
        f = vector<vector<int>>(n + 1, vector<int>(m + 1, -1));
        return dp(0, 0, s, p);
    }

    bool dp(int x, int y, string &s, string &p)
    {
        if (f[x][y] != -1) return f[x][y];
        if (y == m)
            return f[x][y] = x == n;
        bool first_match = x < n && (s[x] == p[y] || p[y] == '.');
        bool ans;
        if (y + 1 < m && p[y + 1] == '*')
        {
            ans = dp(x, y + 2, s, p) || first_match && dp(x + 1, y, s, p);
        }
        else
            ans = first_match && dp(x + 1, y + 1, s, p);
        return f[x][y] = ans;
    }
};
```