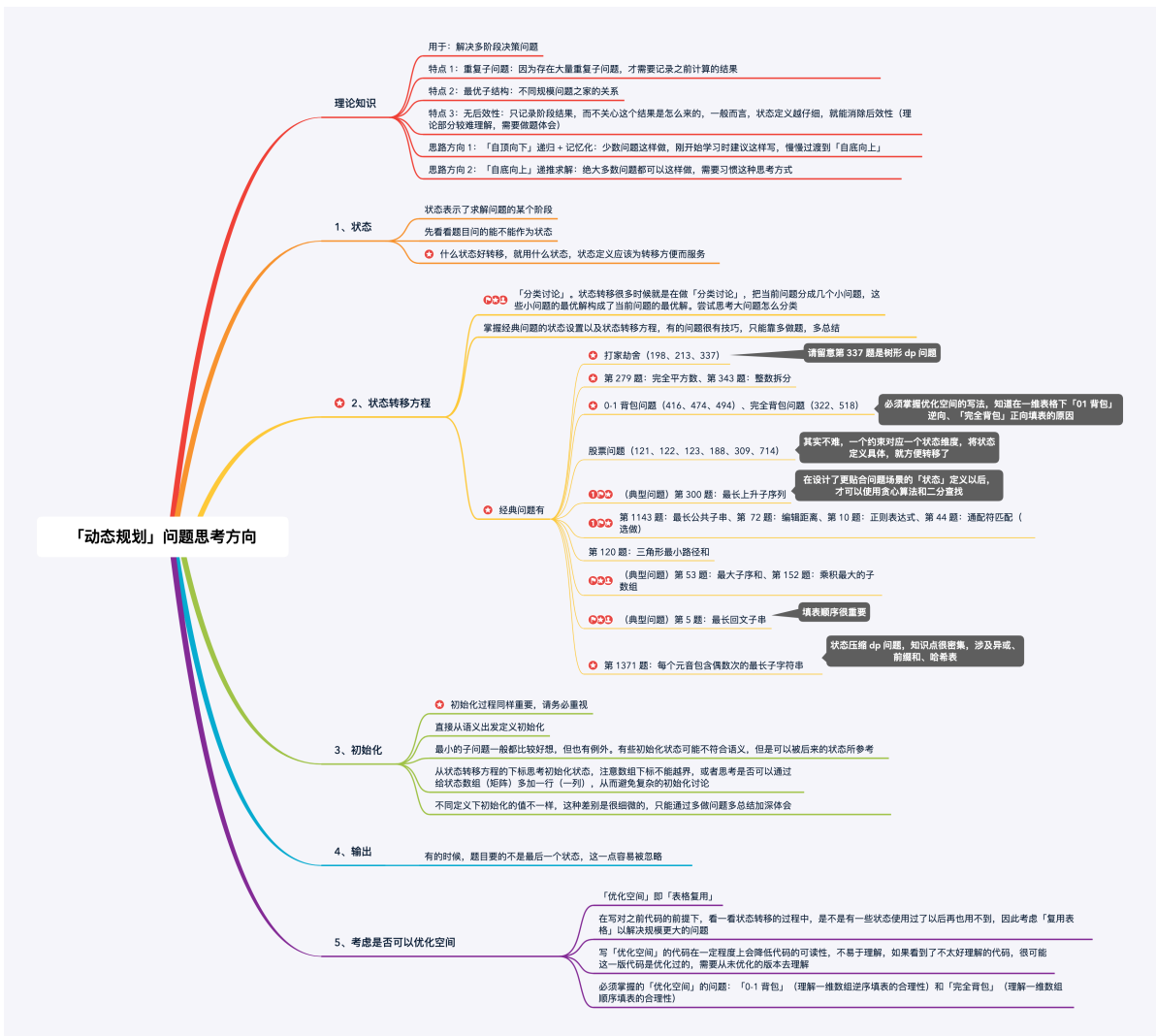


# 动态规划



## 198. 打家劫舍

难度：简单

### 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 不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

示例：

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 。

## Solution

`dp[i]` 表示前 `i` 间房屋能偷窃到的最高总金额

1. 偷窃第 `k` 间房屋, 那么就不能偷窃第 `k-1` 间房屋, 偷窃总金额为前 `k-2` 间房屋的最高总金额与第 `k` 间房屋的金额之和。
2. 不偷窃第 `k` 间房屋, 偷窃总金额为前 `k-1` 间房屋的最高总金额。

```
class Solution {
    public int rob(int[] nums) {
        if(nums == null || nums.length == 0){
            return 0;
        }else if(nums.length == 1){
            return nums[0];
        }

        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        dp[1] = Math.max(dp[0], nums[1]);
        for(int i = 2; i < nums.length; i++){
            dp[i] = Math.max(nums[i] + dp[i-2], dp[i-1]);
        }
        return dp[nums.length-1];
    }
}
```

考虑到每间房屋的最高总金额只和该房屋的前两间房屋的最高总金额相关, 因此可以使用滚动数组, 在每个时刻只需要存储前两间房屋的最高总金额

```
class Solution {
    public int rob(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        int length = nums.length;
        if (length == 1) {
            return nums[0];
        }
        int first = nums[0], second = Math.max(nums[0], nums[1]);
        for (int i = 2; i < length; i++) {
            int temp = second;
            second = Math.max(first + nums[i], second);
            first = temp;
        }
    }
}
```

```
        return second;
    }
}
```

## 213. 打家劫舍 II

难度：中等

### 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。这个地方所有的房屋都围成一圈，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，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。

### 示例：

示例 1：  
输入：[2,3,2]  
输出：3  
解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为他们是相邻的。

示例 2：  
输入：[1,2,3,1]  
输出：4  
解释：你可以先偷窃 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。

### Solution

动态规划。和198题的区别在于收尾相连。则转化为两个单排列问题：

1. 不选第一个房子，即 `nums[1:]`
2. 不选最后一个房子，即 `nums[0:n-1]`

两者取最大值

```
class Solution {
    public int rob(int[] nums) {
        if(nums == null) return 0;
        int n = nums.length;
        if(n == 0){
            return 0;
        }else if(n == 1){
            return nums[0];
        }
        int max1 = robNormal(Arrays.copyOfRange(nums, 1, n));
        int max2 = robNormal(Arrays.copyOfRange(nums, 0, n-1));
        return Math.max(max1, max2);
    }
    public int robNormal(int[] nums){
        if(nums == null || nums.length == 0){
            return 0;
        }
        int length = nums.length;
        if(length == 1){
            return nums[0];
        }
        int first = nums[0];
        int second = Math.max(nums[0], nums[1]);
```

```

        for(int i = 2; i < length; i++){
            int tmp = second;
            second = Math.max(first + nums[i], second);
            first = tmp;
        }
        return second;
    }
}

```

## 91. 解码方法

难度：中等

### 题目描述

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)。

### Solution

```

class Solution {
    public int numDecodings(String s) {
        int n = s.length();
        if(n == 0) return 0;
        char[] charArray = s.toCharArray();
        int[] dp = new int[n];
        if(charArray[0] == '0') return 0;
        dp[0] = 1;
        for(int i = 1; i < n; i++){
            if(charArray[i] != '0'){
                dp[i] = dp[i-1];
            }
            int num = 10 * (charArray[i - 1] - '0') + (charArray[i] - '0');
            if(num >= 10 && num <= 26){
                dp[i] += i == 1 ? 1 : dp[i-2];
            }
        }
        return dp[n-1];
    }
}

```

```
}
```

## \*410. 分割数组的最大值

给定一个非负整数数组和一个整数  $m$ ，你需要将这个数组分成  $m$  个非空的连续子数组。设计一个算法使得这  $m$  个子数组各自和的最大值最小。

**注意:**

数组长度  $n$  满足以下条件:

- $1 \leq n \leq 1000$
- $1 \leq m \leq \min(50, n)$

**Solution**

动态规划+前缀和

```
class Solution {
    public int splitArray(int[] nums, int m) {
        int n = nums.length;
        //dp[i][j]:前i个数分成j段时候的子数组的各自和的最大值的最小值
        int[][] dp = new int[n+1][m+1];
        int[] sub = new int[n+1]; //前缀和
        for(int i = 0; i < n; i++){
            sub[i+1] = sub[i] + nums[i];
        }
        for(int i = 0; i <= n; i++){
            for(int j = 0; j <= m; j++){
                dp[i][j] = Integer.MAX_VALUE;
            }
        }
        dp[0][0] = 0;

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

## 记忆化搜索

本质是 dfs + 记忆化，用在状态的转移方向不确定的情况

## 576. 出界的路径数

给定一个  $m \times n$  的网格和一个球。球的起始坐标为  $(i,j)$ ，你可以将球移到相邻的单元格内，或者往上、下、左、右四个方向上移动使球穿过网格边界。但是，你最多可以移动  $N$  次。找出可以将球移出边界的路径数量。答案可能非常大，返回结果  $\text{mod } 10^9 + 7$  的值。

**示例 1:**

输入:  $m = 2, n = 2, N = 2, i = 0, j = 0$   
输出: 6

## 示例 2:

输入:  $m = 1, n = 3, N = 3, i = 0, j = 1$   
输出: 12

## 说明:

1. 球一旦出界, 就不能再被移动回网格内。
2. 网格的长度和高度在  $[1, 50]$  的范围内。
3.  $N$  在  $[0, 50]$  的范围内。

## Solution

- 状态:  $dp[i][j][k]$ : 当前在  $(i, j)$  点, 还有  $k$  步移出边界的方案数
- 状态转移: 从周围四个格子出发, 移动  $k-1$  步

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

- 边界:  $dp[i][j][0] = 1$

```
class Solution {
    public int findPaths(int m, int n, int N, int x, int y) {
        //dp[i][j][k]: 当前在(i,j)点, 还有k步移出边界的方案数
        if(N == 0) return 0;
        long[][][] dp = new long[m+2][n+2][N+1]; //边上加一圈
        long MOD = 1000000007;
        //边界
        for(int i = 0; i <= m; i++){
            dp[i][0][0] = 1;
            dp[i][n+1][0] = 1;
        }
        for(int j = 0; j <= n; j++){
            dp[0][j][0] = 1;
            dp[m+1][j][0] = 1;
        }

        for(int k = 1; k <= N; k++){
            for(int i = 1; i <= m; i++){
                for(int j = 1; j <= n; j++){
                    dp[i][j][k] = dp[i-1][j][k-1] + dp[i][j-1][k-1] + dp[i+1][j][k-1] + dp[i][j+1][k-1];
                    dp[i][j][k] %= MOD;
                }
            }
        }
        long sum = 0;
        for(int k = 1; k <= N; k++){
            sum = (sum + dp[x+1][y+1][k]) % MOD;
        }
        return (int)sum;
    }
}
```

## 329. 矩阵中的最长递增路径

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）。

### Solution

#### 1. 记忆化深度优先搜索

计算每个点开始的最长递增路径（dfs），取最大值

```
class Solution {
    public int[][] dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    public int rows, cols;
    public int longestIncreasingPath(int[][] matrix) {
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0){
            return 0;
        }
        rows = matrix.length;
        cols = matrix[0].length;
        int[][] memo = new int[rows][cols];
        int ans = 0;
        // 计算每个点的最长递增路径 用记忆化dfs
        for(int i = 0; i < rows; i++){
            for(int j = 0; j < cols; j++){
                ans = Math.max(ans, dfs(matrix, i, j, memo));
            }
        }
        return ans;
    }
    public int dfs(int[][] matrix, int row, int col, int[][]memo){
        //已经计算过了就直接返回
        if(memo[row][col] != 0){
            return memo[row][col];
        }
        ++memo[row][col];
        //四个方向
        for(int[] dir : dirs){
            int newRow = row + dir[0];
            int newCol = col + dir[1];
            if(newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols &&
matrix[newRow][newCol] > matrix[row][col]){
                memo[row][col] = Math.max(memo[row][col], dfs(matrix, newRow,
newCol, memo) + 1);
            }
        }
        return memo[row][col];
    }
}
```

#### 2. 拓扑排序 和[207. 课程表](#)类似

```
class Solution {
    public int longestIncreasingPath(int[][] matrix) {
        if (matrix==null||matrix.length==0){
            return 0;
        }
```

```

    }
    if(matrix == null || matrix.length == 0 || matrix[0].length == 0){
        return 0;
    }
    int rows = matrix.length;
    int cols = matrix[0].length;
    int[][] count = new int[rows][cols]; //入度
    int[][] direction = {{0, 1}, {0, -1}, {-1, 0}, {1, 0}};
    // 上下左右，每发现一个比当前点小的数，当前点入度+1
    for(int i = 0; i < rows; i++){
        for(int j = 0; j < cols; j++){
            for(int[] d : direction){
                int newRow = i + d[0];
                int newCol = j + d[1];
                if(newRow >= 0 && newRow < rows && newCol >= 0 && newCol <
cols && matrix[newRow][newCol] < matrix[i][j]){
                    count[i][j]++;
                }
            }
        }
    }
    Deque<int[]> deque = new LinkedList<>();
    // 所有入度为0的点加入队列
    for(int i = 0; i < rows; i++){
        for(int j = 0; j < cols; j++){
            if(count[i][j] == 0){
                deque.add(new int[]{i, j});
            }
        }
    }
    int ans = 0;
    // bfs
    while(!deque.isEmpty()){
        ans++;
        // 一层一层的出列，而不是一个一个的出，因为课程表那个不关心队列长度
        for (int size = deque.size(); size > 0; size--) {
            int[] pre = deque.poll();
            for(int[] d : direction){
                int x = pre[0] + d[0];
                int y = pre[1] + d[1];
                if(x >= 0 && x < rows && y >= 0 && y < cols && matrix[x][y]
> matrix[pre[0]][pre[1]]){
                    if(--count[x][y] == 0){
                        deque.add(new int[]{x, y});
                    }
                }
            }
        }
    }
    return ans;
}
}

```

**Tips:**

**拓扑排序模板:**



1. 构建入度表和邻接表
2. 所有入度为0的点加入队列
3. bfs

## AcWing 901. 滑雪

给定一个R行C列的矩阵，表示一个矩形网格滑雪场。

矩阵中第 i 行第 j 列的点表示滑雪场的第 i 行第 j 列区域的高度。

一个人从滑雪场中的某个区域内出发，每次可以向上下左右任意一个方向滑动一个单位距离。

当然，一个人能够滑动到某相邻区域的前提是该区域的高度低于自己目前所在区域的高度。

现在给定你一个二维矩阵表示滑雪场各区域的高度，请你找出在该滑雪场中能够完成的最长滑雪轨迹，并输出其长度(可经过最大区域数)。

### Solution



## 数字拆分相关

### 279. 完全平方数

给定正整数  $n$ ，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于  $n$ 。你需要让组成和的完全平方数的个数最少。

### Solution

- 状态：dp[i] 代表组成和为i的完全平方数的最少个数。答案为 dp[n]
- 状态转移：取的最后一个完全平方数为k， $dp[i] = \min(dp[i-k] + 1)$

```
class Solution {
    public int numSquares(int n) {
        int[] dp = new int[n + 1];
        dp[0] = 0;
        for(int i = 1; i <= n; i++){
            dp[i] = i; //最多个数: 取1, 1, 1..., 1, i个1
            for(int j = 1; i - j * j >= 0; j++){
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
            }
        }
        return dp[n];
    }
}
```

2. 也可以用**完全背包**的思想。是coin change 问题，可以先找到自己的coins，即找到小于等于n的所有平方数集合，然后就是用最少的coins来凑target，见322题。

```
class Solution {
    public int numSquares(int n) {
        //构造coins: 小于等于n的所有完全平方数的集合
        List<Integer> coins = new LinkedList<>();
        for(int i = 1; i * i <= n; i++){
            coins.add(i * i);
        }
    }
}
```

```

    }
    int[] dp = new int [n + 1];
    Arrays.fill(dp, n+1); //背包必须装满
    dp[0] = 0;
    for(int coin : coins){ //物品
        for(int j = coin; j <= n; j++){ //体积
            dp[j] = Math.min(dp[j], dp[j-coin] + 1);
        }
    }
    return dp[n];
}
}

```

### 343. 整数拆分

给定一个正整数  $n$ ，将其拆分为**至少**两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

#### Solution

- 状态：  $dp[i]$  代表和为  $i$  的时候，可以获得的最大乘积。返回  $dp[n]$
- 状态转移： 取最后一个数字为  $k$ 。对于每一个状态而言，还要再比较“不再继续分割”和“继续分割：分成  $k$  和  $i-k$ ；分成  $k$  和  $(i-k)$  的分解结果，两者取最大。

$dp[i] = \max(k * (i - k), dp[i-k] * k)$

这里  $(i - k)$  不包括在  $dp[i-k]$  里的原因是：题目说是**至少**两个正整数，循环里条件里也是  $k < i$

```

class Solution {
    public int integerBreak(int n) {
        //dp[i]代表和为i的时候，可以获得的最大乘积
        int[] dp = new int[n + 1];
        for(int i = 1; i <= n; i++){
            dp[i] = 1; //乘积最小是1*1*1...
            for(int k = 1; k < i; k++){
                //分成k和i-k；分成k和(i-k)的分解结果。取最大。
                dp[i] = Math.max(Math.max(dp[i], k * (i-k)), dp[i-k] * k);
            }
        }
        return dp[n];
    }
}

```

## 位运算相关

### 338. 比特位计数

给定一个非负整数  $num$ 。对于  $0 \leq i \leq num$  范围中的每个数字  $i$ ，计算其二进制数中的 1 的数目并将它们作为数组返回。

#### Solution

直接求：

```

class Solution {
    public int[] countBits(int num) {
        int[] ans = new int[num+1];
        for(int i = 0; i <= num; i++){
            ans[i] = popcount(i);
        }
        return ans;
    }
    public int popcount(int x){
        int count;
        for(count = 0; x != 0; count++){
            x &= x-1;
        }
        return count;
    }
}

```

dp的方法：利用已有的计数结果来生成新的计数结果

动态规划 + 最后设置位

```

class Solution {
    public int[] countBits(int num) {
        int[] ans = new int[num+1];
        for(int i = 1; i <= num; i++){
            // i & (i-1): 从右到左第一个为1的位设置为0
            ans[i] = ans[i & (i-1)] + 1;
        }
        return ans;
    }
}

```

## 求 字符串(或数组)之间的某种关系

### 72. 编辑距离

难度：困难

#### 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：

```

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')

```

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')

## Solution

### 动态规划

- 定义状态:  $dp[i][j]$  表示 A 的前  $i$  个字母和 B 的前  $j$  个字母之间的编辑距离,也就是 word1 中前  $i$  个字符,变换到 word2 中前  $j$  个字符,最短需要操作的次数
- 初始化和边界情况: 有一个字符为空的情况,全增加或者全删除,  $dp[i][0]$  和  $dp[0][j]$
- 状态转移:

如果刚好这两个字母相同  $word1[i - 1] = word2[j - 1]$ , 那么  $dp[i][j] = dp[i - 1][j - 1]$

如果这两个字母不相同:

1. 增: A的末尾增加一个,  $dp[i][j] = dp[i][j - 1] + 1$
2. 删: B的末尾增加一个,  $dp[i][j] = dp[i - 1][j] + 1$
3. 改,  $dp[i][j] = dp[i - 1][j - 1] + 1$

取三种情况的最小

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

```
class Solution {
    public int minDistance(String word1, String word2) {
        int len1 = word1.length();
        int len2 = word2.length();
        //其中有一个为空
        if(len1 * len2 == 0){
            return len1 + len2;
        }
        int[][] dp = new int[len1+1][len2+1]; //表示 A 的前 i 个字母和 B 的前 j 个字母之间的编辑距离
        //边界
        for(int i = 0; i <= len1; i++){
            dp[i][0] = i;
        }
        for(int j = 0; j <= len2; j++){
            dp[0][j] = j;
        }
        //dp[i][j]
        for(int i = 1; i <= len1; i++){
            for(int j = 1; j <= len2; j++){
                if(word1.charAt(i-1) == word2.charAt(j-1)){
                    dp[i][j] = dp[i-1][j-1];
                }else{
                    dp[i][j] = Math.min(dp[i-1][j-1]+1, Math.min(dp[i][j-1]+1, dp[i-1][j]+1));
                }
            }
        }
    }
}
```

```
    }  
    return dp[len1][len2];  
  }  
}
```

## 44. 通配符匹配

难度：困难

### 题目描述

给定一个字符串 (s) 和一个字符模式 (p)，实现一个支持 '?' 和 '\*' 的通配符匹配。

- '?' 可以匹配任何单个字符。
- '\*' 可以匹配任意字符串（包括空字符串）。
- 两个字符串完全匹配才算匹配成功。

### 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 = "\*"

输出: true

解释: '\*' 可以匹配任意字符串。

示例 3:

输入:

s = "cb"

p = "?a"

输出: false

解释: '?' 可以匹配 'c'，但第二个 'a' 无法匹配 'b'。

示例 4:

输入:

s = "adceb"

p = "\*a\*b"

输出: true

解释: 第一个 '\*' 可以匹配空字符串，第二个 '\*' 可以匹配字符串 "dce"。

示例 5:

输入:

s = "acdcb"

```
p = "a*c?b"
输出: false
```

## Solution

闫式DP分析法, 链接<https://github.com/shinezzz/LeetCode-Topic>

状态表示: `dp[i][j]` 表示s的前i个字符和p的前j个字符是否匹配, 值为 bool;

状态计算:

1. `p[j]` 为 '\*', `dp[i][j] = dp[i][j - 1] || dp[i - 1][j]`
    - o a) \* 表示空字符, 则保持i不动, j-1, 如果 `dp[i][j-1]` 能匹配, 则 `dp[i][j]` 能匹配。
    - o b) \* 表示多字符, 则保持j不动, i-1, 如果 `dp[i-1][j]` 能匹配, 则 `dp[i][j]` 能匹配。因为\*表示多字符, 则当前i-1加上一个字符后的i也就能匹配
  2. `p[j]` 为 '?', `dp[i][j] = dp[i - 1][j]`
  3. `p[j]` 为 'a-z', `dp[i][j] = s[i - 1] == p[j - 1] && dp[i - 1][j - 1]`
- 2和3可以合并考虑

初始化: `dp[0][0] = true` 表示空串是匹配的。处理一下匹配串 p 以若干个星号开头的情况。因为星号是可以匹配空串的。

```
class Solution {
    public boolean isMatch(String s, String p) {
        int m = s.length();
        int n = p.length();
        boolean[][] dp = new boolean[m+1][n+1];
        //初始化
        dp[0][0] = true;
        // 处理一下匹配串 p 以若干个星号开头的情况
        for(int j = 1; j <= n; j++){
            if(p.charAt(j-1) == '*'){
                dp[0][j] = true;
            }else{
                break;
            }
        }
        // 状态计算
        for(int i = 1; i <= m; i++){
            for(int j = 1; j <= n; j++){
                char si = s.charAt(i-1);
                char pj = p.charAt(j-1);
                if(si == pj || pj == '?'){
                    dp[i][j] = dp[i-1][j-1];
                }else if(pj == '*'){
                    dp[i][j] = dp[i][j-1] || dp[i-1][j];
                }
            }
        }
        return dp[m][n];
    }
}
```

## 718. 最长重复子数组

难度：中等

### 题目描述

给两个整数数组 **A** 和 **B**，返回两个数组中公共的、长度最长的子数组的长度。

### 示例：

输入：  
A: [1,2,3,2,1]  
B: [3,2,1,4,7]  
输出：3  
解释：  
长度最长的公共子数组是 [3, 2, 1]。

### 提示：

- $1 \leq \text{len}(A), \text{len}(B) \leq 1000$
- $0 \leq A[i], B[i] < 100$

### Solution

和44一样，只不过将字符串换成了数组

```
class Solution {
    public int findLength(int[] A, int[] B) {
        int m = A.length;
        int n = B.length;
        int[][] dp = new int[m+1][n+1];
        int ans = 0;
        for(int i = 1; i <= m; i++){
            for(int j = 1; j <= n; j++){
                if(A[i-1] == B[j-1]){
                    dp[i][j] = dp[i-1][j-1] + 1;
                    ans = Math.max(ans, dp[i][j]);
                }
            }
        }
        return ans;
    }
}
```

## 583. 两个字符串的删除操作

难度：中等

### 题目描述

给定两个单词 *word1* 和 *word2*，找到使得 *word1* 和 *word2* 相同所需的最小步数，每步可以删除任意一个字符串中的一个字符。

### 示例：

输入: "sea", "eat"

输出: 2

解释: 第一步将"sea"变为"ea", 第二步将"eat"变为"ea"

#### 提示:

1. 给定单词的长度不超过500。
2. 给定单词中的字符只含有小写字母。

#### Solution

```
class Solution {
    public int minDistance(String word1, String word2) {
        int m = word1.length();
        int n = word2.length();
        if(m * n == 0){
            return m+n;
        }
        int[][] dp = new int[m+1][n+1];
        dp[0][0] = 0;
        for(int i = 0; i <= m; i++){
            dp[i][0] = i;
        }
        for(int j = 0; j <= n; j++){
            dp[0][j] = j;
        }
        for(int i = 1; i <= m; i++){
            for(int j = 1; j <= n; j++){
                if(word1.charAt(i-1) == word2.charAt(j-1)){
                    dp[i][j] = dp[i-1][j-1];
                }else{
                    dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1]) + 1;
                }
            }
        }
        return dp[m][n];
    }
}
```

## 10. 正则表达式匹配

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 `*`。

```
class Solution {
    public boolean isMatch(String s, String p) {
        boolean[][] dp = new boolean[s.length()+1][p.length()+1];
        dp[0][0] = true;
        //初始化
```



```

        for(int j = 2; j <= p.length(); j++){
            // 有*和前一个字符组合就能与任何空字符串(i=0)匹配
            // dp[j-2]也要为true, 才可以说明前j个为true
            dp[0][j] = dp[0][j-2] && p.charAt(j-1) == '*';
        }
        for(int i = 1; i <= s.length(); i++){
            for(int j = 1; j <= p.length(); j++){
                if(s.charAt(i-1) == p.charAt(j-1) || p.charAt(j-1) == '.'){
                    dp[i][j] = dp[i-1][j-1];
                }else if(p.charAt(j-1) == '*'){
                    // dp[i][j-2]: *匹配0个
                    // dp[i-1][j]: *匹配多个, 例如: aabb..., aab*
                    dp[i][j] = dp[i][j-2] || (dp[i-1][j] && (s.charAt(i-1) ==
p.charAt(j-2) || p.charAt(j-2) == '.'));
                }
            }
        }
        return dp[s.length()][p.length()];
    }
}

```

## 博弈问题

### 极小化极大!

取最坏情况下最好

### 486. 预测赢家

给定一个表示分数的非负整数数组。玩家1从数组任意一端拿取一个分数，随后玩家2继续从剩余数组任意一端拿取分数，然后玩家1拿，……。每次一个玩家只能拿取一个分数，分数被拿取之后不再可取。直到没有剩余分数可取时游戏结束。最终获得分数总和最多的玩家获胜。

给定一个表示分数的数组，预测玩家1是否会成为赢家。你可以假设每个玩家的玩法都会使他的分数最大化。

#### 示例 1:

输入: [1, 5, 2]

输出: False

解释: 一开始，玩家1可以从1和2中进行选择。

如果他选择2（或者1），那么玩家2可以从1（或者2）和5中进行选择。如果玩家2选择了5，那么玩家1则只剩下1（或者2）可选。

所以，玩家1的最终分数为  $1 + 2 = 3$ ，而玩家2为 5。

因此，玩家1永远不会成为赢家，返回 False。

#### 示例 2:

输入: [1, 5, 233, 7]

输出: True

解释: 玩家1一开始选择1。然后玩家2必须从5和7中进行选择。无论玩家2选择了哪个，玩家1都可以选择233。

最终，玩家1（234分）比玩家2（12分）获得更多的分数，所以返回 True，表示玩家1可以成为赢家。

#### 注意:

1.  $1 \leq$  给定的数组长度  $\leq 20$ .

2. 数组里所有分数都为非负数且不会大于10000000。
3. 如果最终两个玩家的分数相等，那么玩家1仍为赢家。

## Solution

### 1. 递归

```
class Solution {
    public boolean PredictTheWinner(int[] nums) {
        return winner(nums, 0, nums.length-1, 1) >= 0;
    }
    public int winner(int[] nums, int s, int e, int turn){
        if(s == e){
            return turn * nums[s];
        }
        int a = turn * nums[s] + winner(nums, s+1, e, -turn);
        int b = turn * nums[e] + winner(nums, s, e-1, -turn);
        return turn * Math.max(turn * a, turn * b);
    }
}
```

### 2. 动态规划

更好解的做法：详见1690的讲解。

```
class Solution {
    public boolean PredictTheWinner(int[] piles) {
        int n = piles.length;
        int[][] dp = new int[n][n];
        //dp[i][j]表示剩下[i, j]的数还没选的时候，先手对后手的净胜分的最大值
        for (int len = 1; len <= n; len++) { //枚举区间长度 剩下还没拿的
            for (int l = 0; l + len - 1 < n; l++) { //枚举左端点
                int r = l + len - 1; //右端点
                if (len == 1) { //边界
                    dp[l][r] = piles[l];
                } else {
                    dp[l][r] = Math.max(piles[l] - dp[l + 1][r], piles[r] - dp[l][r - 1]);
                }
            }
        }
        return dp[0][n - 1] >= 0;
    }
}
```

之前的做法：博弈论相关 区间dp

- 状态：dp[i][j] 表示剩下 [i, j] 的数还没选的时候，第一个玩家的最大分数。只用算第一个玩家的分数，因为总分一定，总分减去第一个玩家的分数就是第二个玩家的分数。

```
class Solution {
    public boolean PredictTheWinner(int[] nums) {
        int n = nums.length;
        int[][] dp = new int[n][n];
```

```

//分数个数是奇数，说明最后剩下的一定是玩家1拿的
if(n % 2 == 1){
    for(int i = 0; i < n; i++){
        dp[i][i] = nums[i];
    }
}
//区间dp
for(int len = 2; len <= n; len++){//区间
    for(int j = 0; j + len - 1 < n; j++){//左端点
        int l = j;
        int r = j + len - 1;
        //如果已经取走的分数的个数是偶数，接下来该玩家1走了
        if((n - (r - l + 1)) % 2 == 0){
            dp[l][r] = Math.max(dp[l + 1][r] + nums[l], dp[l][r - 1] +
nums[r]);
        }else{
            //玩家2选
            dp[l][r] = Math.min(dp[l + 1][r], dp[l][r - 1]);
        }
    }
}
int sum = 0; //分数总和
for(int x : nums){
    sum += x;
}
//返回玩家1的分数是否大于等于玩家2的
return dp[0][n-1] >= sum - dp[0][n-1];
}
}

```

相似题，简单版：

### 1423. 可获得的最大点数

几张卡牌 排成一行，每张卡牌都有一个对应的点数。点数由整数数组 `cardPoints` 给出。

每次行动，你可以从行的开头或者末尾拿一张卡牌，最终你必须正好拿 `k` 张卡牌。

你的点数就是你拿到手中的所有卡牌的点数之和。

给你一个整数数组 `cardPoints` 和整数 `k`，请你返回可以获得的最大点数。

示例 1：

输入：`cardPoints = [1,2,3,4,5,6,1]`, `k = 3`

输出：12

解释：第一次行动，不管拿哪张牌，你的点数总是 1。但是，先拿最右边的卡牌将会最大化你的可获得点数。最优策略是拿右边的三张牌，最终点数为  $1 + 6 + 5 = 12$ 。

#### Solution

最后剩下连续 $n-k$ 张牌，滑动窗口求每个连续长度为 $n-k$ 的区间之内的和的最小值，这样拿走的牌就是最大的。

```

class Solution {
    public int maxScore(int[] cardPoints, int k) {
        int n = cardPoints.length;
        int sum = 0;
        int min = Integer.MAX_VALUE;
    }
}

```

```

        int len = n - k;
        int num = 0;
        for (int i = 0, j = 0; i < n; i++) {
            sum += cardPoints[i];
            num += cardPoints[i];
            if (i - j + 1 > len) {
                num -= cardPoints[j];
                j++;
            }
            if (i - j + 1 == len) {
                min = Math.min(min, num);
            }
        }
        return sum - min;
    }
}

```

```

class Solution {
    public int maxScore(int[] cardPoints, int k) {
        int n = cardPoints.length;
        int[] s = new int[n + 1];
        int sum = 0;
        for (int i = 1; i <= n; i++) {
            sum += cardPoints[i - 1];
            s[i] = s[i - 1] + cardPoints[i - 1];
        }
        int min = sum;
        int len = n - k;
        for (int i = 1; i + len - 1 <= n; i++) {
            int j = i + len - 1;
            min = Math.min(min, s[j] - s[i - 1]);
        }
        return sum - min;
    }
}

```

## 1690. 石子游戏 VII

石子游戏中，爱丽丝和鲍勃轮流进行自己的回合，爱丽丝先开始。

有  $n$  块石子排成一排。每个玩家的回合中，可以从行中 移除 最左边的石头或最右边的石头，并获得与该行中剩余石头值之和 相等的得分。当没有石头可移除时，得分较高者获胜。

鲍勃发现他总是输掉游戏（可怜的鲍勃，他总是输），所以他决定尽力 减小得分的差值。爱丽丝的目标是最大限度地 扩大得分的差值。

给你一个整数数组 `stones`，其中 `stones[i]` 表示 从左边开始 的第  $i$  个石头的值，如果爱丽丝和鲍勃都发挥出最佳水平，请返回他们 得分的差值。

**示例 1：**

输入: stones = [5,3,1,4,2]

输出: 6

解释:

- 爱丽丝移除 2 , 得分  $5 + 3 + 1 + 4 = 13$  。游戏情况: 爱丽丝 = 13 , 鲍勃 = 0 , 石子 = [5,3,1,4] 。
  - 鲍勃移除 5 , 得分  $3 + 1 + 4 = 8$  。游戏情况: 爱丽丝 = 13 , 鲍勃 = 8 , 石子 = [3,1,4] 。
  - 爱丽丝移除 3 , 得分  $1 + 4 = 5$  。游戏情况: 爱丽丝 = 18 , 鲍勃 = 8 , 石子 = [1,4] 。
  - 鲍勃移除 1 , 得分 4 。游戏情况: 爱丽丝 = 18 , 鲍勃 = 12 , 石子 = [4] 。
  - 爱丽丝移除 4 , 得分 0 。游戏情况: 爱丽丝 = 18 , 鲍勃 = 12 , 石子 = [] 。
- 得分的差值  $18 - 12 = 6$  。

## 示例 2:

输入: stones = [7,90,5,1,100,10,10,2]

输出: 122

## 提示:

- `n == stones.length`
- `2 <= n <= 1000`
- `1 <= stones[i] <= 1000`

## Solution

### 博弈论: 取最坏情况下最好



image-20201213141816535

```
class Solution {
    public int stoneGameVII(int[] stones) {
        // f[i][j] 表示剩余[i,j]的时候, 此时的先手-后手的最大差值
        int n = stones.length;
        int[][] f = new int[n + 1][n + 1];
        int[] s = new int[n + 1]; // 前缀和
        for(int i = 1; i <= n; i++) {
            s[i] = s[i - 1] + stones[i - 1];
        }
        // 边界: len==1时候只剩一个石子, 拿走后剩余石头的和为0, f[i][i]=0, 所以这里len从2开始
        for(int len = 2; len <= n; len++) {
            for(int i = 1; i + len - 1 <= n; i++) {
                int j = i + len - 1;
                // 如果f[i][j]为A先手, 则f[i+1][j]为B先手 (B-A的最大差值)
                // A的最坏情况, 就是B的最好情况。A-B的最小值, 就是B-A的最大值加负号, 即-(B-A)
                // s[j] - s[i] - f[i+1][j]表示取左边时的最坏情况
                // 取最坏情况下最好
                f[i][j] = Math.max(s[j] - s[i] - f[i+1][j], s[j-1] - s[i-1] - f[i][j-1]);
            }
        }
        return f[1][n];
    }
}
```

二：周赛时候自己写的区间dp：现在回过去看有点想不通了

```
class Solution {
    public int stoneGameVII(int[] stones) {
        int n = stones.length;
        int[][] dp = new int[n][n];
        //分数个数是偶数，说明最后剩下的一定是B拿的
        if(n % 2 == 0){
            for(int i = 0; i < n; i++){
                dp[i][i] = stones[i];
            }
        }
        //dp[i][j] 表示剩[i, j]的数还没选的时候的差值,也就是B选的数字
        for(int len = 2; len <= n; len++){//区间
            for(int j = 0; j + len - 1 < n; j++){//左端点
                int l = j;
                int r = j + len - 1;
                //如果已经取走的分数的个数是偶数，接下来该A拿了
                if((n - (r - l + 1)) % 2 == 0){
                    dp[l][r] = Math.max(dp[l+1][r], dp[l][r-1]);
                }else{
                    // B拿
                    dp[l][r] = Math.min(dp[l+1][r] + stones[l], dp[l][r-1] +
stones[r]);
                }
            }
        }
        return dp[0][n - 1];
    }
}
```

## 877. 石子游戏

亚历克斯和李用几堆石子在做游戏。偶数堆石子排成一行，每堆都有正整数颗石子  $piles[i]$ 。

游戏以谁手中的石子最多来决出胜负。石子的总数是奇数，所以没有平局。

亚历克斯和李轮流进行，亚历克斯先开始。每回合，玩家从行的开始或结束处取走整堆石头。这种情况一直持续到没有更多的石子堆为止，此时手中石子最多的玩家获胜。

假设亚历克斯和李都发挥出最佳水平，当亚历克斯赢得比赛时返回 `true`，当李赢得比赛时返回 `false`。

**提示：**

$2 \leq piles.length \leq 500$

$piles.length$  是偶数。

$1 \leq piles[i] \leq 500$

$\sum(piles)$  是奇数。

**Solution**

和486题思路相同

```
class Solution {
    public boolean stoneGame(int[] piles) {
        int n = piles.length;
        int[][] dp = new int[n][n];
```

```

        for(int len = 2; len <= n; len++){//区间
            for(int j = 0; j + len - 1 < n; j++){//左端点
                int l = j;
                int r = j + len - 1;
                if((n - (r - l + 1)) % 2 == 0){
                    dp[l][r] = Math.max(dp[l+1][r] + piles[l], dp[l][r-1] +
piles[r]);
                }else{
                    dp[l][r] = Math.min(dp[l+1][r], dp[l][r-1]);
                }
            }
        }
        int sum = 0;
        for(int p : piles){
            sum += p;
        }
        return dp[0][n-1] > sum - dp[0][n-1];
    }
}

```

第二种状态转移的思路（更经典：）

 image-20201213214701869

和486的区别在于，这道题是偶数堆

```

class Solution {
    public boolean stoneGame(int[] piles) {
        int n = piles.length;
        int[][] dp = new int[n][n]; //表示剩下[i,j]的时候，当前先手-后手的最大差值(先手对
后手的净胜分)
        for(int len = 2; len <= n; len++){//区间
            for(int i = 0; i + len - 1 < n; i++){//左端点
                int j = i + len - 1;
                dp[i][j] = Math.max(piles[i] - dp[i + 1][j], piles[j] - dp[i][j
- 1]);
            }
        }
        return dp[0][n-1] >= 0;
    }
}

```

## \*1140. 石子游戏 II

亚历克斯和李继续他们的石子游戏。许多堆石子 排成一行，每堆都有正整数颗石子 piles[i]。游戏以谁手中的石子最多来决出胜负。

亚历克斯和李轮流进行，亚历克斯先开始。最初， $M = 1$ 。

在每个玩家的回合中，该玩家可以拿走剩下的 前  $X$  堆的所有石子，其中  $1 \leq X \leq 2M$ 。然后，令  $M = \max(M, X)$ 。

游戏一直持续到所有石子都被拿走。

假设亚历克斯和李都发挥出最佳水平，返回亚历克斯可以得到的最大数量的石头。

**Solution**

这类题目重点在于理解：**最坏情况下的最好值!**

```
class Solution {
    public int stoneGameII(int[] piles) {
        int n = piles.length;
        //dp[i][j] : 还剩下[i, n-1]的石头还没选,M取j的时候, 当前玩家可以得到的最大数量的石
        头
        int[][] dp = new int[n][n+1];
        int sum = 0;
        for(int i = n-1; i >= 0; i--){
            sum += piles[i];
            for(int M = 1; M <= n; M++){
                if(i + 2 * M >= n){
                    //i+2M 到达了石子堆的末尾, 直接一下子可以全部拿走, 对于当前玩家, 最好的
                    方案是全部取走石子
                    dp[i][M] = sum; //sum[i:n-1]
                }else{
                    for(int x = 1; x <= 2 * M; x++){
                        //不能全部取走, 下一个玩家肯定也会取最多的石头, sum[i:n-1]减去下一个
                        玩家最多能取的石头, 这是最坏情况, 取最坏情况下的最好值, 所以对不同的x取max
                        //dp[i + x][Math.max(M, x)]表示下一个玩家最多能取的石头
                        dp[i][M] = Math.max(dp[i][M], sum - dp[i + x]
[Math.max(M, x)]);
                    }
                }
            }
        }
        return dp[0][1];
    }
}
```

## 1406. 石子游戏 III

Alice 和 Bob 用几堆石子在做游戏。几堆石子排成一行，每堆石子都对应一个得分，由数组 stoneValue 给出。

Alice 和 Bob 轮流取石子，Alice 总是先开始。在每个玩家的回合中，该玩家可以拿走剩下石子中的的前 1、2 或 3 堆石子。比赛一直持续到所有石头都被拿走。

每个玩家的最终得分为他所拿到的每堆石子的对应得分之和。每个玩家的初始分数都是 0。比赛的目标是决出最高分，得分最高的选手将会赢得比赛，比赛也可能会出现平局。

假设 Alice 和 Bob 都采取 最优策略。如果 Alice 赢了就返回 "Alice"，Bob 赢了就返回 "Bob"，平局（分数相同）返回 "Tie"。

### Solution

这一题和1140的区别在于，石头会是负得分，所以不能一下子都拿完，有可能会越拿越小。

```
class Solution {
    public String stoneGameIII(int[] stoneValue) {
        int n = stoneValue.length;
        int[] dp = new int[n + 1];
        // dp[i]表示: 当前剩余[i,n-1]的石头的时候, 当前玩家的最大得分
        // 多冗余一位 dp[n] 表示在区间 [n...n]范围内当前玩家所能拿到的最多的石子的数量, 这个
        区间不存在
        int sum = 0;
        // dp[n] = 0;
```



```

        for (int i = n - 1; i >= 0; i--) {
            sum += stoneValue[i]; //sum[i,n-1], 后缀和
            //如果已经可以一次性全部拿走游戏就结束了, 就全部拿走吗? 不一定, 因为石头可能是负
            //数, 越拿越小
            //当前玩家拿走1,2,3堆
            dp[i] = Integer.MIN_VALUE; //因为会有负数, 所以先给一个负无穷
            for (int k = 1; k <= 3 && i + k <= n; k++) {
                //下一个人也会拿最好的情况, 对于当前玩家来说, 这就是最坏情况
                //取最坏情况下的最好值
                dp[i] = Math.max(dp[i], sum - dp[i + k]);
            }
        }
        //dp[0]代表A的分数, sum-dp[0]为B的分数
        if (dp[0] > sum - dp[0]) {
            return new String("Alice");
        } else if (dp[0] < sum - dp[0]) {
            return new String("Bob");
        } else {
            return new String("Tie");
        }
    }
}

```

## 1510. 石子游戏 IV

Alice 和 Bob 两个人轮流玩一个游戏, Alice 先手。

一开始, 有  $n$  个石子堆在一起。每个人轮流操作, 正在操作的玩家可以从石子堆里拿走 任意 非零 平方数 个石子。

如果石子堆里没有石子了, 则无法操作的玩家输掉游戏。

给你正整数  $n$ , 且已知两个人都采取最优策略。如果 Alice 会赢得比赛, 那么返回 True, 否则返回 False。

### Solution

博弈论中, 必胜态和必败态

**必胜态:** 我拿完后抛给对手, 可以走到对手的某个必败态

**必败态:** 我不管怎么走, 对手都是必胜的, 我只能走到对手的必胜态

```

class Solution {
    public boolean winnerSquareGame(int n) {
        // f[i]表示, 当石子剩下i个的时候, 当前玩家能否赢得这个阶段的比赛
        boolean[] f = new boolean[n + 1];
        for (int i = 1; i <= n; i++) {
            for (int k = 1; k * k <= i; k++) {
                //如果能走到对手的某个必败态, 我就是必胜的
                if (!f[i - k * k]) {
                    f[i] = true;
                    break;
                }
            }
        }
        //不管怎么走, 都走不到对手的必败态, 只能走到对手的必胜态, 我就是必败的。
        return f[n];
    }
}

```

```
}
```

## 1563. 石子游戏 V

几块石子 **排成一行**，每块石子都有一个关联值，关联值为整数，由数组 `stoneValue` 给出。

游戏中的每一轮：Alice 会将这行石子分成两个 **非空行**（即，左侧行和右侧行）；Bob 负责计算每一行的值，即此行中所有石子的值的总和。Bob 会丢弃值最大的行，Alice 的得分为剩下那行的值（每轮累加）。如果两行的值相等，Bob 让 Alice 决定丢弃哪一行。下一轮从剩下的那一行开始。

只 **剩下一块石子** 时，游戏结束。Alice 的分数最初为 `0`。

返回 **Alice 能够获得的最高分数**。

提示：

- `1 <= stoneValue.length <= 500`
- `1 <= stoneValue[i] <= 10^6`

**Solution**

```
class Solution {
    public int stoneGameV(int[] stoneValue) {
        int n = stoneValue.length;
        int[] s = new int[n + 1];
        for (int i = 1; i <= n; i++) {
            s[i] = s[i - 1] + stoneValue[i - 1];
        }
        int[][] dp = new int[n][n]; // 剩下石头 [i, j] 时，A 的最大分数
        for (int len = 2; len <= n; len++) {
            for (int i = 0; i + len - 1 < n; i++) {
                int j = i + len - 1;
                // 分界 [i, k], [k+1, j]
                for (int k = i; k < j; k++) {
                    int left = s[k + 1] - s[i];
                    int right = s[j + 1] - s[k + 1];
                    if (left > right) {
                        // 丢掉左边
                        dp[i][j] = Math.max(dp[i][j], right + dp[k + 1][j]);
                    } else if (left < right) {
                        // 丢掉右边
                        dp[i][j] = Math.max(dp[i][j], left + dp[i][k]);
                    } else {
                        dp[i][j] = Math.max(dp[i][j], left + Math.max(dp[i][k],
                            dp[k + 1][j]));
                    }
                }
            }
        }
        return dp[0][n - 1];
    }
}
```

## 375. 猜数字大小 II

我们正在玩一个猜数游戏，游戏规则如下：

我从 1 到 n 之间选择一个数字，你来猜我选了哪个数字。

每次你猜错了，我都会告诉你，我选的数字比你的大了或者小了。

然而，当你猜了数字 x 并且猜错了的时候，你需要支付金额为 x 的现金。直到你猜到我选的数字，你才算赢得了这个游戏。

**示例：**

n = 10, 我选择了8.

第一轮: 你猜我选择的数字是5，我会告诉你，我的数字更大一些，然后你需要支付5块。

第二轮: 你猜是7，我告诉你，我的数字更大一些，你支付7块。

第三轮: 你猜是9，我告诉你，我的数字更小一些，你支付9块。

游戏结束。8 就是我选的数字。

你最终要支付  $5 + 7 + 9 = 21$  块钱。

**Solution**

**最坏情况下的最小值：**对于不能决定的情况，要选最坏情况。对于能自己决定的情况，要选最小值。

 image-20201214100143663

还是用区间dp

```
class Solution {
    public int getMoneyAmount(int n) {
        int[][] dp = new int[n + 2][n + 2];
        for(int len = 2; len <= n; len++) {
            for(int i = 1; i + len - 1 <= n; i++) {
                int j = i + len - 1;
                dp[i][j] = Integer.MAX_VALUE;
                for(int k = i; k <= j; k++) {
                    dp[i][j] = Math.min(dp[i][j], k + Math.max(dp[i][k - 1],
                        dp[k + 1][j]));
                }
            }
        }
        return dp[1][n];
    }
}
```

## 一、数字三角形模型

### 120. 三角形最小路径和

难度：中等

**题目描述**

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。

**相邻的结点** 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。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。

### Solution

1. 二维动态规划，从三角形顶部走到(i,j)位置的最小路径和为 `dp[i][j]`

```
class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        int m = triangle.size();
        int n = triangle.get(m-1).size();
        int res = Integer.MAX_VALUE;
        int[][] dp = new int[m][n];
        dp[0][0] = triangle.get(0).get(0);
        for(int i = 1; i < m; i++){
            for(int j = 0; j < triangle.get(i).size(); j++){
                if(j == 0){
                    dp[i][j] = dp[i-1][j] + triangle.get(i).get(j);
                }else if(i == j){
                    dp[i][j] = dp[i-1][j-1] + triangle.get(i).get(j);
                }else{
                    dp[i][j] = Math.min(dp[i-1][j-1], dp[i-1][j]) +
triangle.get(i).get(j);
                }
            }
        }

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

2. 改成一维，这时候要从大到小循环

```
class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        int m = triangle.size();
        int n = triangle.get(m-1).size();
        int res = Integer.MAX_VALUE;
        int[] dp = new int[n];
        dp[0] = triangle.get(0).get(0);
```

```

        for(int i = 1; i < m; i++){
            for(int j = triangle.get(i).size() - 1; j >= 0; j--){
                if(j == 0){
                    dp[j] = dp[j] + triangle.get(i).get(j);
                }else if(i == j){
                    dp[j] = dp[j-1] + triangle.get(i).get(j);
                }else{
                    dp[j] = Math.min(dp[j-1], dp[j]) + triangle.get(i).get(j);
                }
            }
        }
        for(int j = 0; j < n; j++){
            res = Math.min(res, dp[j]);
        }
        return res;
    }
}

```

## 二、子序列子串问题

1. 涉及两个字符串/数组时（比如最长公共子序列），**dp** 数组的含义如下：  
在子数组 `arr1[0..i]` 和子数组 `arr2[0..j]` 中，我们要求的子序列（最长公共子序列）长度为 `dp[i][j]`。
2. 只涉及一个字符串/数组时（比如最长回文子序列），**dp** 数组的含义如下：  
在子数组 `array[i..j]` 中，我们要求的子序列（最长回文子序列）的长度为 `dp[i][j]`。

一些名词

最长上升子序列(LIS):

最长连续序列(LCS):

最长连续递增序列(LCIS):

最长公共子序列(LCS):

### 53. 最大子序和

难度：简单

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。

**Solution**

- 状态: `dp[i]`: 表示以 `nums[i]` 结尾的连续子数组的最大和。
- 状态转移: 由于 `nums[i]` 一定会被选取，状态转移根据 `dp[i - 1]` 正和负来考虑（看倒数第二个数）。

```

class Solution {
    public int maxSubArray(int[] nums) {
        if(nums == null) return 0;
        int n = nums.length;
        int[] dp = new int[n];
        dp[0] = nums[0];
        int max = nums[0];
        for(int i = 1; i < n; i++){
            dp[i] = Math.max(dp[i-1] + nums[i], nums[i]);
            if(dp[i] > max){
                max = dp[i];
            }
        }
        return max;
    }
}

```

## 152. 乘积最大子数组

给你一个整数数组 `nums`，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

**Solution**

```

class Solution {
    public int maxProduct(int[] nums) {
        // dp[i][0]:以nums[i]为结尾的连续子数组的乘积的最大值
        // dp[i][1]:以nums[i]为结尾的连续子数组的乘积的最小值
        if(nums == null || nums.length == 0) return 0;
        int n = nums.length;
        int[][] dp = new int[n][2];
        dp[0][0] = nums[0];
        dp[0][1] = nums[0];
        for(int i = 1; i < n; i++){
            //分正负，负的时候最大最小状态转换
            if(nums[i] >= 0){
                dp[i][0] = Math.max(nums[i], dp[i-1][0] * nums[i]);
                dp[i][1] = Math.min(nums[i], dp[i-1][1] * nums[i]);
            }else{
                dp[i][0] = Math.max(nums[i], dp[i-1][1] * nums[i]);
                dp[i][1] = Math.min(nums[i], dp[i-1][0] * nums[i]);
            }
        }
        int max = nums[0];
        for(int i = 1; i < n; i++){
            max = Math.max(dp[i][0], max);
        }
        return max;
    }
}

```

## 300. 最长上升子序列

难度：中等

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？

**Solution**

需要对「子序列」和「子串」这两个概念进行区分：子序列并不要求连续，子串一定是连续的。

- 状态：dp[i]：表示以 nums[i] 结尾的最长上升子序列的长度。
- 状态转移：dp[i] 就等于下标 i 之前严格小于 nums[i] 的状态值的最大者 +1。

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        int n = nums.length;
        if(n <= 1) return n;
        int[] dp = new int[n];
        Arrays.fill(dp, 1);
        int max = 1;
        for(int i = 1; i < n; i++){
            for(int j = 0; j < i; j++){
                if(nums[i] > nums[j]){
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            if(dp[i] > max){
                max = dp[i];
            }
        }
        return max;
    }
}
```

优化：修改状态定义。

- 状态：tail[i] 表示长度为 i + 1 的所有上升子序列的结尾的最小值。(贪心：结尾的数越小，遍历的时候后面接上一个数，就会有更大的可能性构成一个更长的上升子序列)

todo...

## AcWing 896. 最长上升子序列 II

给定一个长度为N的数列，求数值严格单调递增的子序列的长度最长是多少。

### 数据范围

$1 \leq N \leq 100000$ ,  
 $-109 \leq \text{数列中的数} \leq 109$

### Solution

是前面最长上升子序列的优化版本 复杂度  $O(n \log n)$

找各个长度的上升子序列的结尾最小，（能接在结尾更大的序列后面，一定能接在结尾更小的序列后面，所以要找结尾最小的）

长度越长，结尾最小值越大（q是单调递增的）

q存的是每种长度的最长上升子序列的结尾的最小值是多少

遍历每个数 $a[i]$ ， $a[i]$ 可以接到所有比自己小的数的末尾，每次找小于 $a[i]$ 的最大的数（二分），例如找到了这个数字是 $q_4$ ， $q_4 < a[i]$ ， $q_5 \geq a[i]$ ，则最长上升子序列的长度为 $4+1=5$ ，（ $a[i]$ 可以接到 $q_4$ 后面），然后更新 $q_5 = a[i]$

```
import java.io.*;
import java.util.*;

public class Main{
    public static void main(String[] args) throws IOException{
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine());
        String[] str = br.readLine().split(" ");
        int[] a = new int[n];
        int[] q = new int[n + 1];
        for(int i = 0; i < n; i++) {
            a[i] = Integer.parseInt(str[i]);
        }
        br.close();
        int len = 0;
        for(int i = 0; i < n; i++) {
            int l = 0;
            int r = len;
            // 二分 在q中找小于a[i]的最大的数
            while(l < r) {
                int mid = l + (r - l + 1) / 2;
                if(q[mid] < a[i]) {
                    l = mid;
                }else {
                    r = mid - 1;
                }
            }
            len = Math.max(len, r + 1);
            q[r + 1] = a[i];
        }
        System.out.println(len);
    }
}
```



## 5644. 得到子序列的最少操作次数

给你一个数组 `target`，包含若干互不相同的整数，以及另一个整数数组 `arr`，`arr` 可能包含重复元素。

每一次操作中，你可以在 `arr` 的任意位置插入任一整数。比方说，如果 `arr = [1,4,1,2]`，那么你可以在中间添加 3 得到 `[1,4,3,1,2]`。你可以在数组最开始或最后面添加整数。

请你返回最少操作次数，使得 `target` 成为 `arr` 的一个子序列。

一个数组的子序列指的是删除原数组的某些元素（可能一个元素都不删除），同时不改变其余元素的相对顺序得到的数组。比方说，`[2,7,4]` 是 `[4,2,3,7,2,1,4]` 的子序列（加粗元素），但 `[2,4,2]` 不是子序列。

### 示例 1:

```
输入: target = [5,1,3], arr = [9,4,2,3,4]
输出: 2
解释: 你可以添加 5 和 1，使得 arr 变为 [5,9,4,1,2,3,4]，target 为 arr 的子序列。
```

### 示例 2:

```
输入: target = [6,4,8,1,3,2], arr = [4,7,6,2,3,8,6,1]
输出: 3
```

### 提示:

$1 \leq \text{target.length}, \text{arr.length} \leq 105$

$1 \leq \text{target}[i], \text{arr}[i] \leq 109$

`target` 不包含任何重复元素。

### Solution

转化：求最小操作次数，就是 `n` 减去二者的最大公共子序列长度。

但是，求最大公共子序列是  $n^2$  的做法。可以转换成求最长上升子序列问题(前提是每个数不同)，就可以优化成  $n \log n$ 。

```
class Solution {
    public int minOperations(int[] target, int[] arr) {
        Map<Integer, Integer> map = new HashMap<>(); //target中每个数字的位置
        for(int i = 0; i < target.length; i++) {
            map.put(target[i], i);
        }
        //arr中在target中有的数字，在target中是什么位置
        List<Integer> list = new ArrayList<>();
        for(int i = 0; i < arr.length; i++) {
            if(map.containsKey(arr[i])) {
                list.add(map.get(arr[i]));
            }
        }
        //找list的最长上升子序列
        int n = list.size();
        int[] q = new int[n + 1];
        int len = 0;
        for(int i = 0; i < n; i++) {
            int l = 0;
            int r = len;
            // 二分 在q中找小于a[i]的最大的数
            while(l < r) {
```

```

        int mid = l + (r - l + 1) / 2;
        if(q[mid] < list.get(i)) {
            l = mid;
        }else {
            r = mid - 1;
        }
    }
    len = Math.max(len, r + 1);
    q[r + 1] = list.get(i);
}
return target.length - len;
}
}

```

## 673. 最长递增子序列的个数

给定一个未排序的整数数组，找到最长递增子序列的个数。

**注意:** 给定的数组长度不超过 2000 并且结果一定是32位有符号整数。

```

class Solution {
    public int findNumberOfLIS(int[] nums) {
        int n = nums.length;
        int[] dp = new int[n]; //以nums[i]为结尾的最长上升子序列的长度
        int[] count = new int[n]; //以nums[i]为结尾的最长上升子序列的个数
        Arrays.fill(dp, 1);
        Arrays.fill(count, 1);
        int max = 0;
        int res = 0;
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < i; j++) {
                if(nums[j] < nums[i]) {
                    if(dp[j] + 1 == dp[i]) {
                        count[i] += count[j];
                    }else if(dp[j] + 1 > dp[i]) {
                        dp[i] = dp[j] + 1;
                        count[i] = count[j]; //在所有dp[j]后面接上nums[i],个数不变,长度+1
                    }
                }
            }
            max = Math.max(max, dp[i]);
        }
        for(int i = 0; i < n; i++) {
            if(dp[i] == max) {
                res += count[i];
            }
        }
        return res;
    }
}

```

### 354. 俄罗斯套娃信封问题

给定一些标记了宽度和高度的信封，宽度和高度以整数对形式 (w, h) 出现。当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算最多能有多少个信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

**说明:**

不允许旋转信封。

**示例:**

输入: envelopes = [[5,4],[6,4],[6,7],[2,3]]  
输出: 3  
解释: 最多信封的个数为 3，组合为: [2,3] => [5,4] => [6,7]。

最长上升子序列的二维版本。

```
class Solution {
    public int maxEnvelopes(int[][] envelopes) {
        //按照w升序排，如果w相等，则h降序排
        Arrays.sort(envelopes, new Comparator<int[]>(){
            public int compare(int[] arr1, int[] arr2){
                if(arr1[0] == arr2[0]){
                    return arr2[1] - arr1[1];
                }else{
                    return arr1[0] - arr2[0];
                }
            }
        });
        //提取h
        int[] h = new int[envelopes.length];
        for(int i = 0; i < envelopes.length; i++){
            h[i] = envelopes[i][1];
        }
        //最长上升子序列
        return lengthOfLIS(h);
    }
    public int lengthOfLIS(int[] nums) {
        int n = nums.length;
        if(n <= 1) return n;
        //dp[i]代表以nums[i]为结尾的最长上升子序列的长度
        int[] dp = new int[n];
        Arrays.fill(dp, 1);
        int max = 1;
        for(int i = 1; i < n; i++){
            for(int j = 0; j < i; j++){
                if(nums[i] > nums[j]){
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            if(dp[i] > max){
                max = dp[i];
            }
        }
        return max;
    }
}
```

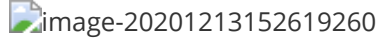
```
}
```

## 5245. 堆叠长方体的最大高度

给你  $n$  个长方体  $\text{cuboids}$ ，其中第  $i$  个长方体的长宽高表示为  $\text{cuboids}[i] = [\text{width}_i, \text{length}_i, \text{height}_i]$ （下标从 0 开始）。请你从  $\text{cuboids}$  选出一个子集，并将它们堆叠起来。

如果  $\text{width}_i \leq \text{width}_j$  且  $\text{length}_i \leq \text{length}_j$  且  $\text{height}_i \leq \text{height}_j$ ，你就可以将长方体  $i$  堆叠在长方体  $j$  上。你可以通过旋转把长方体的长宽高重新排列，以将它放在另一个长方体上。

返回堆叠长方体  $\text{cuboids}$  可以得到的最大高度。



```
class Solution {
    public int maxHeight(int[][] cuboids) {
        //每个长方体内部从小到大排序
        for(int[] cuboid : cuboids) {
            Arrays.sort(cuboid);
        }
        // 第一维从小到大，第二维从小到大，第三维从小到大
        Arrays.sort(cuboids, (o1, o2) -> {
            if(o1[0] != o2[0]) {
                return o1[0] - o2[0];
            } else {
                if(o1[1] != o2[1]) {
                    return o1[1] - o2[1];
                } else {
                    return o1[2] - o2[2];
                }
            }
        });
        // dp求最长递增子序列 dp[i]表示以i为结尾的高度
        int n = cuboids.length;
        int[] dp = new int[n];
        int res = 0;
        for(int i = 0; i < n; i++) {
            dp[i] = cuboids[i][2];
            for(int j = 0; j < i; j++) {
                // 如果j的每一维度都比i要小，说明j可以放在i的上面
                if(cuboids[j][0] <= cuboids[i][0] && cuboids[j][1] <= cuboids[i][1] && cuboids[j][2] <= cuboids[i][2]) {
                    dp[i] = Math.max(dp[i], cuboids[i][2] + dp[j]);
                }
            }
            res = Math.max(res, dp[i]);
        }
        return res;
    }
}
```

## 5. 最长回文子串

### 题目描述

难度：中等

给定一个字符串 `s`，找到 `s` 中最长的回文子串。你可以假设 `s` 的最大长度为 1000。

### 示例 1:

输入: "babad"  
输出: "bab"  
注意: "aba" 也是一个有效答案。

### 示例 2:

输入: "cbbd"  
输出: "bb"

### Solution

#### 动态规划

- 状态: `dp[i][j]` 表示子串 `s[i..j]` 是否为回文子串
- 边界和初始化:

单个字符一定是回文串, 即 `dp[i][i] = true`;

两个字符, 如果他们相等, 则是回文子串

- 状态转移:

`dp[i][j] = (s[i] == s[j]) && dp[i+1][j-1]`

例如对于字符串 `ababa`, 如果我们已经知道 `bab` 是回文串, 那么 `ababa` 一定是回文串, 这是因为它的首尾两个字母都是 `a`

```
class Solution {
    public String longestPalindrome(String s) {
        int len = s.length();
        if(len < 2){
            return s;
        }
        int maxLen = 1;
        int begin = 0;
        // dp[i][j] 表示 s[i, j] 是否是回文串
        boolean[][] dp = new boolean[len][len];
        char[] charArray = s.toCharArray();
        // 单个字符一定是回文串
        for(int i = 0; i < len; i++){
            dp[i][i] = true;
        }
        // 右边界j, 左边界i
        for(int j = 1; j < len; j++){
            // 右边界先不动, 动左边界
            for(int i = 0; i < j; i++){
                if(charArray[i] != charArray[j]){
                    dp[i][j] = false;
                }else{
```

```

        if(j - i < 3){
            dp[i][j] = true;
        }else{
            dp[i][j] = dp[i+1][j-1];
        }
    }
    // 记录回文长度和起始位置
    if(dp[i][j] && j - i + 1 > maxLen){
        maxLen = j - i + 1;
        begin = i;
    }
}
}
return s.substring(begin, begin + maxLen);
}
}

```

用区间

```

class Solution {
    public String longestPalindrome(String s) {
        //dp[i][j]代表: [i,j]是否是回文子串
        int n = s.length();
        boolean[][] dp = new boolean[n][n];
        int max = 0;
        int start = 0;
        for(int len = 1; len <= n; len++){
            for(int i = 0; i + len - 1 < n; i++){
                int j = i + len - 1;
                if(s.charAt(i) == s.charAt(j)){
                    if(len <= 2){
                        dp[i][j] = true;
                    }else{
                        dp[i][j] = dp[i+1][j-1];
                    }
                }else{
                    dp[i][j] = false;
                }
                if(dp[i][j] && len > max){
                    max = len;
                    start = i;
                }
            }
        }
        return s.substring(start, start + max);
    }
}

```

## 132. 分割回文串 II

给定一个字符串  $s$ ，将  $s$  分割成一些子串，使每个子串都是回文串。

返回符合要求的最少分割次数。

示例:

输入: "aab"

输出: 1

解释: 进行一次分割就可将  $s$  分割成 ["aa", "b"] 这样两个回文子串。

### Solution

- 状态:

集合: 从0~i分割成子串, 使子串都是回文串的分割方法

属性: 最少分割次数

- 状态转移: 枚举分割位置k, 如果s[k,i]是回文, 总次数就是前面分割的次数+1

```
dp[i] = Math.min(dp[i], dp[k-1] + 1);
```

```
class Solution {
    public int minCut(String s) {
        int n = s.length();
        if(n < 2) return 0;
        int[] dp = new int[n];
        //初始化 最大次数就是每个字母分一次
        for(int i = 0; i < n; i++){
            dp[i] = i;
        }
        for(int i = 1; i < n; i++){
            // s[0...i]本身就是回文
            if(checkPalindrome(s, 0, i)){
                dp[i] = 0;
                continue;
            }
            // 枚举分割位置k
            for(int k = 1; k <= i; k++){
                // 如果s[k,i]是回文, 总次数就是前面分割的次数+1
                if(checkPalindrome(s, k, i)){
                    dp[i] = Math.min(dp[i], dp[k-1] + 1);
                }
            }
        }
        return dp[n-1];
    }
    // 是否是回文串
    private boolean checkPalindrome(String s, int left, int right) {
        while (left < right) {
            if (s.charAt(left) != s.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }
    }
}
```

```

        return true;
    }
}

```

优化：O(1)的时间复杂度，得到一个子串是否是回文。参考「力扣」第 5 题：最长回文子串 动态规划的解法

```

class Solution {
    public int minCut(String s) {
        int n = s.length();
        if(n < 2) return 0;
        int[] dp = new int[n];
        //初始化 最大次数就是每个字母分一次
        for(int i = 0; i < n; i++){
            dp[i] = i;
        }

        //判断是否是回文子串 参考第5题
        boolean[][] checkPalindrome = new boolean[n][n];
        for(int right = 0; right < n; right++){
            for(int left = 0; left <= right; left++){
                if (s.charAt(left) == s.charAt(right) && (right - left <= 2
|| checkPalindrome[left + 1][right - 1])) {
                    checkPalindrome[left][right] = true;
                }
            }
        }

        for(int i = 1; i < n; i++){
            // s[0...i]本身就是回文
            if(checkPalindrome[0][i]){
                dp[i] = 0;
                continue;
            }
            // 枚举分割位置k
            for(int k = 1; k <= i; k++){
                // 如果s[k,i]是回文，总次数就是前面分割的次数+1
                if(checkPalindrome[k][i]){
                    dp[i] = Math.min(dp[i], dp[k-1] + 1);
                }
            }
        }
        return dp[n-1];
    }
}

```

## 5666. 回文串分割 IV

给你一个字符串  $s$ ，如果可以将它分割成三个 非空 回文子字符串，那么返回 `true`，否则返回 `false`。

当一个字符串正着读和反着读是一模一样的，就称其为 回文字符串。

示例 1：



输入: `s = "abcbdd"`

输出: `true`

解释: `"abcbdd" = "a" + "bcb" + "dd"`, 三个子字符串都是回文的。

示例 2:

输入: `s = "bcbddxy"`

输出: `false`

解释: `s` 没办法被分割成 3 个回文子字符串。

**提示:**

- `3 <= s.length <= 2000`
- `s` 只包含小写英文字母。

**Solution**

预处理`[i,j]`是否是回文串 `f[i][j] = (s[i] == s[j]) && f[i+1][j-1]`

然后 $O(n^2)$ 枚举分割点

## 516. 最长回文子序列

给定一个字符串 `s` , 找到其中最长的回文子序列, 并返回该序列的长度。可以假设 `s` 的最大长度为 1000 。

**示例 1:**

输入:

`"bbbab"`

输出:

`4`

一个可能的最长回文子序列为 `"bbbb"`。

**示例 2:**

输入:

`"cbbd"`

输出:

`2`

一个可能的最长回文子序列为 `"bb"`。

**提示:**

- `1 <= s.length <= 1000`
- `s` 只包含小写英文字母

**Solution**

- 状态: 在子串 `s[i..j]` 中, 最长回文子序列的长度为 `dp[i][j]`

我们要求的就是 `dp[0][n - 1]`

- 状态转移: 看 `s[i]` 和 `s[j]` 的字符

```

if (s[i] == s[j])
    // 它俩一定在最长回文子序列中
    dp[i][j] = dp[i + 1][j - 1] + 2;
else
    // s[i+1..j] 和 s[i..j-1] 谁的回文子序列更长?
    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);

```

要从dp[i+1]推dp[i] 所以i要反着遍历

```

class Solution {
    public int longestPalindromeSubseq(String s) {
        if(s == null || s.length() == 0) return 0;
        int n = s.length();
        int[][] dp = new int[n][n];
        for(int i = 0; i < n; i++){
            dp[i][i] = 1;
        }
        for(int i = n-1; i >= 0; i--){
            for(int j = i + 1; j < n; j++){
                if(s.charAt(i) == s.charAt(j)){
                    dp[i][j] = dp[i+1][j-1] + 2;
                }else{
                    dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
                }
            }
        }
        return dp[0][n-1];
    }
}

```

用区间dp的写法

```

class Solution {
    public int longestPalindromeSubseq(String s) {
        if(s == null || s.length() == 0) return 0;
        int n = s.length();
        int[][] dp = new int[n][n];
        for(int i = 0; i < n; i++){
            dp[i][i] = 1;
        }
        for(int len = 2; len <= n; len++){//区间长度
            for(int i = 0; i + len - 1 < n; i++){//左端点
                int j = i + len - 1;//右端点
                if(s.charAt(i) == s.charAt(j)){
                    dp[i][j] = dp[i+1][j-1] + 2;
                }else{
                    dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
                }
            }
        }
        return dp[0][n-1];
    }
}

```

### 730. 统计不同回文子序列

给定一个字符串  $S$ ，找出  $S$  中不同的非空回文子序列个数，并返回该数字与  $10^9 + 7$  的模。

通过从  $S$  中删除 0 个或多个字符来获得子序列。

如果一个字符序列与它反转后的字符序列一致，那么它是回文字符序列。

如果对于某个  $i$ ， $A_i \neq B_i$ ，那么  $A_1, A_2, \dots$  和  $B_1, B_2, \dots$  这两个字符序列是不同的。

#### Solution

516的进阶版

区间dp

```
class Solution {
    public int countPalindromicSubsequences(String S) {
        //dp[i][j]代表: [i, j]区间的非空回文子序列个数
        int n = S.length();
        int[][] dp = new int[n][n];
        char[] ch = S.toCharArray();
        int mod = 1000000007;
        for (int i = 0; i < n; i++) {
            dp[i][i] = 1;
        }
        //区间dp
        for(int len = 2; len <= n; len++){
            for(int i = 0; i + len - 1 < n; i++){
                int j = i + len - 1;
                if(ch[i] != ch[j]){
                    dp[i][j] = dp[i][j-1] + dp[i+1][j] - dp[i+1][j-1]; //中间的
                    // dp[i+1][j-1] 被统计了两次
                }else{
                    // 所有的回文子串前后都加上s[i]
                    dp[i][j] = 2 * dp[i+1][j-1];
                    // s[i..j]之间的子串有字符与s[i]相同，会产生重复的回文子串，此时需分情况调整

                    int l = i + 1;
                    int r = j - 1;
                    while(l <= r && ch[l] != ch[i]) l++; // 从左侧开始查找相同的元素
                    while(l <= r && ch[r] != ch[i]) r--; // 从右侧开始查找相同的元素
                    if(l > r){
                        //没有找到重复字符 "bcab"
                        dp[i][j] += 2;
                    }else if(l == r){
                        //只有一个重复字符 "bcbcb"
                        dp[i][j] += 1;
                    }else{
                        //找到多个重复字符 "bbcabb"
                        dp[i][j] -= dp[l+1][r-1];
                    }
                }
            }
            dp[i][j] = dp[i][j] < 0 ? dp[i][j] + mod : dp[i][j] % mod;
        }
        return dp[0][n-1];
    }
}
```

## 1143. 最长公共子序列

难度：中等

### 题目描述

给定两个字符串 text1 和 text2，返回这两个字符串的最长公共子序列的长度。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。两个字符串的「公共子序列」是这两个字符串所共同拥有的子序列。

若这两个字符串没有公共子序列，则返回 0。

### 示例 1:

输入: text1 = "abcde", text2 = "ace"  
输出: 3  
解释: 最长公共子序列是 "ace"，它的长度为 3。

### 示例 2:

输入: text1 = "abc", text2 = "abc"  
输出: 3  
解释: 最长公共子序列是 "abc"，它的长度为 3。

### 示例 3:

输入: text1 = "abc", text2 = "def"  
输出: 0  
解释: 两个字符串没有公共子序列，返回 0。

### 提示:

- `1 <= text1.length <= 1000`
- `1 <= text2.length <= 1000`
- 输入的字符串只含有小写英文字符。

### Solution

- 状态定义: `dp[i][j]` 表示text1的前i个字符和text2的前j个字符的最长公共子序列的长度

image-20201121202634607

以text1[i]和text2[j]是否包含在子序列当中来划分

注意：这里划分的四类其实是有重复的，`f[i-1][j]` 代表的是在第一个序列的前i个字母中出现，且在第二个序列的前j个字母中出现的子序列，但01这种情况指的是不选a[i]，一定选b[j]，也就是一定以b[j]为结尾（所以说f[i-1,j]和01这种情况是包含关系）。求max或者min划分是可以重复的，求数量的时候不能重复。

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length();
        int n = text2.length();
```

```

int[][] dp = new int[m+1][n+1];
dp[0][0] = 0;
for(int i = 1; i <= m; i++){
    for(int j = 1; j <= n; j++){
        if(text1.charAt(i-1) == text2.charAt(j-1)){
            dp[i][j] = dp[i-1][j-1] + 1;
        }else{
            dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
        }
    }
}
return dp[m][n];
}
}

```

## 115. 不同的子序列

给定一个字符串 S 和一个字符串 T，计算在 S 的子序列中 T 出现的个数。

一个字符串的一个子序列是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，"ACE" 是 "ABCDE" 的一个子序列，而 "AEC" 不是）

题目数据保证答案符合 32 位带符号整数范围。

### Solution

- 状态：dp[i][j] 表示s[0...i]的子序列中匹配t[0...j]的个数
- 状态转移：
  1. s[i] != t[j] 时：dp[i][j] = dp[i-1][j]
  2. s[i] == t[j] 时：
    - 子序列中不包含 s[i]: dp[i][j] = dp[i-1][j]
    - 子序列中包含 s[i]: dp[i][j] = dp[i-1][j-1]

image-20200727150103430

```

class Solution {
    public int numDistinct(String s, String t) {
        //dp[i][j]:s[0...i]的子序列中匹配t[0...j]的个数
        //前面补一个空字符
        char[] S = (" " + s).toCharArray();
        char[] T = (" " + t).toCharArray();
        int m = s.length();
        int n = t.length();
        int[][] dp = new int[m + 1][n + 1];
        //边界: t是空字符
        for(int i = 0; i <= m; i++){
            dp[i][0] = 1;
        }
        for(int i = 1; i <= m; i++){
            for(int j = 1; j <= n; j++){
                if(s[i] == T[j]){
                    dp[i][j] = dp[i-1][j] + dp[i-1][j-1];
                }else{
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
    }
}

```

```

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

```

### 三、背包模型

#### 322. 零钱兑换

给定不同面额的硬币 `coins` 和一个总金额 `amount`。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

**示例 1:**

```

输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1

```

**示例 2:**

```

输入: coins = [2], amount = 3
输出: -1

```

**思路:**

完全背包

- 状态: `dp[i][j]` 表示只看前 `i` 种硬币, 总金额为 `j` 所需的最少的硬币数量。最后答案为 `dp[coins.length][amount]`
- 状态转移  

$$dp[i][j] = \max(dp[i-1][j], dp[i][j-coins[i]]+1)$$

**Solution**

1. 朴素

```

class Solution {
    public int coinChange(int[] coins, int amount) {
        int n = coins.length;
        if(n == 0) return -1;
        int[][] dp = new int[n+1][amount+1];

        dp[0][0] = 0;
        //第一行其他都是无穷大, 因为背包必须装满
        for(int j = 1; j <= amount; j++){
            dp[0][j] = amount+1;
        }
        for(int i = 1; i <= n; i++){
            for(int j = 0; j <= amount; j++){
                dp[i][j] = dp[i-1][j];
                if(j >= coins[i-1]){
                    dp[i][j] = Math.min(dp[i][j], dp[i][j-coins[i-1]] + 1);
                }
            }
        }
        if(dp[n][amount] == amount+1) return -1;
    }
}

```

```

        return dp[n][amount];
    }
}

```

## 2. 优化

```

class Solution {
    public int coinChange(int[] coins, int amount) {
        int n = coins.length;
        if(n == 0) return -1;
        int[] dp = new int[amount+1];
        //第一行其他都是无穷大，因为背包必须装满
        Arrays.fill(dp, amount+1);
        dp[0] = 0;
        for(int coin : coins){
            for(int j = coin; j <= amount; j++){
                dp[j] = Math.min(dp[j], dp[j-coin] + 1);
            }
        }
        if(dp[amount] == amount+1) return -1;
        return dp[amount];
    }
}

```

## 518. 零钱兑换 II

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

**思路:**

完全背包

- 状态: `dp[i][j]` 表示只看前  $i$  种硬币, 凑成总金额为  $j$  时候的组合数。最后答案为 `dp[coins.length][amount]`
- 状态转移: `dp[i][j] = dp[i-1][j] + dp[i][j-coins[i]]`
- 边界: `dp[0][0] = 1`

**Solution**

1. 朴素

```
class Solution {
    public int change(int amount, int[] coins) {
        int n = coins.length;
        int[][] dp = new int[n+1][amount+1];
        dp[0][0] = 1;
        for(int i = 1; i <= n; i++){
            for(int j = 0; j <= amount; j++){
                dp[i][j] += dp[i-1][j];
                if(j >= coins[i-1]){
                    dp[i][j] += dp[i][j-coins[i-1]];
                }
            }
        }
        return dp[n][amount];
    }
}
```

2. 优化

```
class Solution {
    public int change(int amount, int[] coins) {
        int[] dp = new int[amount+1];
        dp[0] = 1;
        for(int coin : coins){
            for(int j = coin; j <= amount; j++){
                dp[j] += dp[j-coin];
            }
        }
        return dp[amount];
    }
}
```

更多[背包问题]详见背包问题专题。



## 四、状态机

### 股票问题

$dp[i][k][0/1]$  代表第  $i$  天（为下标，从第 0 天开始算）结束的时候，至今最多进行的交易次数为  $k$  次，持有股票（1）/或不持有股票（0）的最大利润

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
```

### 121. 买卖股票的最佳时机

$k=1$

```
class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0) return 0;
        int n = prices.length;
        int[][] dp = new int[n][2];
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        for (int i = 1; i < n; i++) {
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
            dp[i][1] = Math.max(dp[i-1][1], -prices[i]); // dp[i-1][0][0] - prices[i], 因为k=1, dp[i-1][0][0]=0
        }
        return dp[n-1][0];
    }
}
```

空间优化

```
class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0) return 0;
        int n = prices.length;
        int[] dp = new int[2];
        dp[0] = 0;
        dp[1] = -prices[0];
        for (int i = 1; i < n; i++) {
            dp[0] = Math.max(dp[0], dp[1] + prices[i]);
            dp[1] = Math.max(dp[1], -prices[i]);
        }
        return dp[0];
    }
}
```

### 122. 买卖股票的最佳时机 II

$k$  不限

```
class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0) return 0;
```

```

int n = prices.length;
int[][] dp = new int[n][2];
dp[0][0] = 0; //第一天没有操作
dp[0][1] = -prices[0]; //第一天买了
for (int i = 1; i < n; i++) {
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    dp[i][1] = Math.max(dp[i-1][1], dp[i-1][0] - prices[i]);
}
return dp[n-1][0];
}
}

```

空间优化，转一维

```

class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0) return 0;
        int n = prices.length;
        int[] dp = new int[2];
        dp[0] = 0; //第一天没有操作
        dp[1] = -prices[0]; //第一天买了
        for (int i = 1; i < n; i++) {
            dp[0] = Math.max(dp[0], dp[1] + prices[i]);
            dp[1] = Math.max(dp[1], dp[0] - prices[i]);
        }
        return dp[0];
    }
}

```

### 123. 买卖股票的最佳时机 III

k=2

```

class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0) return 0;
        int n = prices.length;
        int K = 2;
        int[][][] dp = new int[n][K+1][2];
        // i=0或者k=0时候的边界情况
        for(int i = 0; i < n; i++){
            dp[i][0][0] = 0;
            dp[i][0][1] = Integer.MIN_VALUE;
        }
        for(int k = 1; k <= K; k++){
            dp[0][k][0] = 0;
            dp[0][k][1] = -prices[0];
        }

        for (int i = 1; i < n; i++) {
            for(int k = 1; k <= K; k++){
                dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] +
prices[i]);
                dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] -
prices[i]);
            }
        }
    }
}

```

```

    }
    return dp[n-1][2][0];
}
}

```

空间优化，三维转二维

```

class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0) return 0;
        int n = prices.length;
        int K = 2;
        int[][] dp = new int[K+1][2];
        // i=0或者k=0时候的边界情况
        dp[0][0] = 0;
        dp[0][1] = Integer.MIN_VALUE;
        for(int k = 1; k <= K; k++){
            dp[k][0] = 0;
            dp[k][1] = -prices[0];
        }
        for (int i = 1; i < n; i++) {
            for(int k = 1; k <= K; k++){
                dp[k][0] = Math.max(dp[k][0], dp[k][1] + prices[i]);
                dp[k][1] = Math.max(dp[k][1], dp[k-1][0] - prices[i]);
            }
        }
        return dp[K][0];
    }
}

```

## 188. 买卖股票的最佳时机 IV

k给定，和123题相比，用原来的方法的话会超过内存限制，因为传入的 k 值可能会非常大。

一次交易由买入和卖出构成，至少需要两天，所以有效的限制 k 应该不超过  $n/2$ ，如果超过，就采取k无限制的情况（122题）。

```

class Solution {
    public int maxProfit(int K, int[] prices) {
        if(prices == null || prices.length == 0) return 0;
        int n = prices.length;
        if(K > n / 2){
            return maxProfitwithOutK(prices);
        }
        int[][] dp = new int[K+1][2];
        // i=0或者k=0时候的边界情况
        dp[0][0] = 0;
        dp[0][1] = Integer.MIN_VALUE;
        for(int k = 1; k <= K; k++){
            dp[k][0] = 0;
            dp[k][1] = -prices[0];
        }
        for (int i = 1; i < n; i++) {
            for(int k = 1; k <= K; k++){
                dp[k][0] = Math.max(dp[k][0], dp[k][1] + prices[i]);
                dp[k][1] = Math.max(dp[k][1], dp[k-1][0] - prices[i]);
            }
        }
    }
}

```

```

    }
    return dp[k][0];
}

public int maxProfitWithoutK(int[] prices) {
    if(prices == null || prices.length == 0) return 0;
    int n = prices.length;
    int[] dp = new int[2];
    dp[0] = 0; //第一天没有操作
    dp[1] = -prices[0]; //第一天买了
    for (int i = 1; i < n; i++) {
        dp[0] = Math.max(dp[0], dp[1] + prices[i]);
        dp[1] = Math.max(dp[1], dp[0] - prices[i]);
    }
    return dp[0];
}
}

```

### 309. 最佳买卖股票时机含冷冻期

#### 题目描述

给定一个整数数组，其中第  $i$  个元素代表了第  $i$  天的股票价格。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

卖出股票后，你无法在第二天买入股票（即冷冻期为 1 天）。

#### 示例:

输入: `[1,2,3,0,2]`

输出: `3`

解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]

#### Solution

##### 动态规划

- 状态: `dp[i]` 表示第  $i$  天结束时的最大利润，分成三种不同的状态
  - `dp[i][0]` 目前持有一支股票对应的最大利润
  - `dp[i][1]` 目前持有 0 支股票，并且处于冷冻期（也就是说第  $i+1$  天无法买股票），对应的最大利润
  - `dp[i][2]` 目前持有 0 支股票，并且不处于冷冻期，对应的最大利润
- 状态转移: 找最后一个不同点，也就是看第  $i$  天进行了什么操作
  - 第  $i$  天没动，或当天买入了: `dp[i][0] = max(dp[i-1][0], dp[i-1][2] - prices[i])`
  - 只能是第  $i$  天卖了: `dp[i][1] = dp[i-1][0] + prices[i]`
  - 当天没动: `dp[i][2] = max(dp[i-1][1], dp[i-1][2])`

最后取 `max(dp[n][0], dp[n][1], dp[n][2])`，也就是 `max(dp[n][1], dp[n][2])`

- 边界:

`dp[0][0] = -prices[0]`

`dp[0][1] = dp[0][2] = 0`

```

class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        if(n == 0) return 0;
        int[][] dp = new int[n+1][3];
        dp[0][0] = -prices[0];
        for(int i = 1; i <= n; i++){
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][2] - prices[i-1]);
            dp[i][1] = dp[i-1][0] + prices[i-1];
            dp[i][2] = Math.max(dp[i-1][1], dp[i-1][2]);
        }
        return Math.max(dp[n][1], dp[n][2]);
    }
}

```

因为状态dp[i]只和dp[i-1]有关，可以二维转一维：

```

class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        if(n == 0) return 0;
        int[] dp = new int[3];
        dp[0] = -prices[0];
        for(int i = 1; i <= n; i++){
            int newf0 = Math.max(dp[0], dp[2] - prices[i-1]);
            int newf1 = dp[0] + prices[i-1];
            int newf2 = Math.max(dp[1], dp[2]);
            dp[0] = newf0;
            dp[1] = newf1;
            dp[2] = newf2;
        }
        return Math.max(dp[1], dp[2]);
    }
}

```

上面是增加状态表示，或者也可以修改状态转移方程，改成  $dp[i-2][0] - prices[i]$

```

dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])

```

```

class Solution {
    public int maxProfit(int[] prices) {
        if(prices == null || prices.length == 0) return 0;
        int n = prices.length;
        int[][] dp = new int[n][2];
        dp[0][0] = 0; //第一天没有操作
        dp[0][1] = -prices[0]; //第一天买了
        for (int i = 1; i < n; i++) {
            if(i == 1){
                dp[1][0] = Math.max(dp[0][0], dp[0][1] + prices[1]);
                dp[1][1] = Math.max(dp[0][1], dp[0][0] - prices[1]);
                continue;
            }
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
            dp[i][1] = Math.max(dp[i-1][1], dp[i-2][0] - prices[i]);
        }
    }
}

```

```

        return dp[n-1][0];
    }
}

```

## 714. 买卖股票的最佳时机含手续费

每次买的时候算上手续费

```

class Solution {
    public int maxProfit(int[] prices, int fee) {
        if(prices == null || prices.length == 0) return 0;
        int n = prices.length;
        int[] dp = new int[2];
        dp[0] = 0; //第一天没有操作
        dp[1] = -prices[0] - fee; //第一天买了
        for (int i = 1; i < n; i++) {
            dp[0] = Math.max(dp[0], dp[1] + prices[i]);
            dp[1] = Math.max(dp[1], dp[0] - prices[i] - fee);
        }
        return dp[0];
    }
}

```

## 五、状态压缩dp

### \*1371. 每个元音包含偶数次的最长子字符串

#### 题目描述

难度：中等

给你一个字符串  $s$ ，请你返回满足以下条件的最长子字符串的长度：每个元音字母，即 'a', 'e', 'i', 'o', 'u'，在子字符串中都恰好出现了偶数次。

#### Solution

##### 前缀和+哈希表+状态压缩

首先根据题目中“恰好出现偶数次”，想到可以用异或来表示。

- **状态压缩**：每一位分别表示每一个元音字母出现的次数，1表示出现次数为奇数，0表示出现次数为偶数。例如：假如到第  $i$  个位置，u o i e a 出现的奇偶性分别为 1 1 0 0 1，那么我们就可以将其压成一个二进制数  $(11001)_2 = (25)$  作为它的状态，则可以用  $2^5 = 32$  位的数组来表示
- **32位的数组中存的是什么**：存的是首次出现这个status的位置
- **前缀和的理解**：当32位数组中下一次再出现这个status时，二者的位置差的这部分子串就是要求的子字符串(偶数个元音异或后那一位还是0)

比如现在我们找到了一个符合要求的子串  $[L,R]$ ，同时有另一个未知符合要求与否的子串  $[0,L]$ ，那么元音字符次数在  $[0,L] + [L,R]$  与  $[0,R]$  的奇偶性是相同的，也就是：出现两个相同状态的数，他们中间必定出现了偶数次数的aeiou

- **map[0] = 0**：如果第一个是辅音，如果map[0]=-1的话就没办法更新ans了
- **map[status] = i+1 和 i+1-map[status]** 为什么是i+1而不是i：

1、如果子字符串出现在原字符串的中间，那么新的i+1减去原来的i+1和 新的i减去原来的i效果是一样的，也就是用i+1和i没什么影响

2、子字符串从头截取，也就是子字符串的第一位和原字符串第一位重合的情况，由于前面  $map[0]=0$ ，所以这边  $i-(-1)=i+1-0$ 。

```
class Solution {
    public int findTheLongestSubstring(String s) {
        int[] map = new int[32];
        Arrays.fill(map, -1);
        int ans = 0;
        int status = 0;
        map[0] = 0;
        for(int i = 0; i < s.length(); i++){
            if(s.charAt(i) == 'a'){
                status ^= 1 << 0;
            }
            if(s.charAt(i) == 'e'){
                status ^= 1 << 1;
            }
            if(s.charAt(i) == 'i'){
                status ^= 1 << 2;
            }
            if(s.charAt(i) == 'o'){
                status ^= 1 << 3;
            }
            if(s.charAt(i) == 'u'){
                status ^= 1 << 4;
            }
            if(map[status] < 0){
                // 还没出现过 更新map
                map[status] = i + 1;
            }else{
                // map[status]为正，代表这个status出现过
                // 出现两个相同状态的数，他们中间必定出现了偶数次数的aeiou
                // 求最大区间
                ans = Math.max(ans, i + 1 - map[status]);
            }
        }
        return ans;
    }
}
```

## 526. 优美的排列

假设有从 1 到 N 的 N 个整数，如果从这 N 个数字中成功构造出一个数组，使得数组的第 i 位 ( $1 \leq i \leq N$ ) 满足如下两个条件中的一个，我们就称这个数组为一个优美的排列。条件：

第 i 位的数字能被 i 整除

i 能被第 i 位上的数字整除

现在给定一个整数 N，请问可以构造多少个优美的排列？

示例1:

输入: 2

输出: 2

解释:

第 1 个优美的排列是 [1, 2]:

第 1 个位置 ( $i=1$ ) 上的数字是1, 1能被  $i$  ( $i=1$ ) 整除

第 2 个位置 ( $i=2$ ) 上的数字是2, 2能被  $i$  ( $i=2$ ) 整除

第 2 个优美的排列是 [2, 1]:

第 1 个位置 ( $i=1$ ) 上的数字是2, 2能被  $i$  ( $i=1$ ) 整除

第 2 个位置 ( $i=2$ ) 上的数字是1,  $i$  ( $i=2$ ) 能被 1 整除

说明:

1.  $N$  是一个正整数, 并且不会超过15。

## Solution

状态压缩dp

- 状态: `dp[state]` 用二进制来表示数。长度:  $2^N$ 。  
state = 11, 也就是1011, 代表当前使用了第一个数、第二个数、第四个数, 一共有多少个不同的排列
- 状态转移: 假设state里有*i*个数, 现在要往第*i+1*个位置上放数, 现在要枚举那些没被选过的数字*j*。  
如果*j*没有在state里出现过, 并且*j*和*i+1*互为倍数, 把放*j*的方案加上: `dp[state | (1 << j)] += dp[state]`
- 初始化: `dp[0] = 1`, 一个数都没有的时候是一种方案

```
class Solution {
    public int countArrangement(int N) {
        // dp[state]表示: 如state = 11, 也就是1011, 代表当前使用了第一个数、第二个数、第四
        // 个数的情况下, 一共有多少个不同的排列
        int[] dp = new int[1 << N];
        dp[0] = 1;
        for(int i = 0; i < (1 << N); i++){
            // 计算已经放的数字个数
            int s = 1;
            for(int j = 0; j < N; j++){
                s += i >> j & 1; // i >> j & 1表示:i的第j位是否是1
            }
            // 放下一个数
            for(int j = 1; j <= N; j++){
                // 没有被选过(对应位置是0), 并且满足互为倍数
                if((i >> (j-1) & 1) == 0 && (s % j == 0 || j % s == 0)){
                    // 把放j的方案加上
                    // dp[010111] += dp[010101]
                    dp[i | (1 << j - 1)] += dp[i];
                }
            }
        }
        return dp[(1 << N) - 1];
    }
}
```



## 464. 我能赢吗

在 "100 game" 这个游戏中，两名玩家轮流选择从 1 到 10 的任意整数，累计整数和，先使得累计整数和达到 100 的玩家，即为胜者。

如果我们将游戏规则改为“玩家不能重复使用整数”呢？

例如，两个玩家可以轮流从公共整数池中抽取从 1 到 15 的整数（不放回），直到累计整数和  $\geq 100$ 。

给定一个整数 `maxChoosableInteger`（整数池中可选择的最最大数）和另一个整数 `desiredTotal`（累计和），判断先出手的玩家是否能稳赢（假设两位玩家游戏时都表现最佳）？

你可以假设 `maxChoosableInteger` 不会大于 20，`desiredTotal` 不会大于 300。

**示例：**

输入：

```
maxChoosableInteger = 10
desiredTotal = 11
```

输出：

```
false
```

解释：

无论第一个玩家选择哪个整数，他都会失败。

第一个玩家可以选择从 1 到 10 的整数。

如果第一个玩家选择 1，那么第二个玩家只能选择从 2 到 10 的整数。

第二个玩家可以通过选择整数 10（那么累积和为  $11 \geq \text{desiredTotal}$ ），从而取得胜利。

同样地，第一个玩家选择任意其他整数，第二个玩家都会赢。

### Solution

状态压缩dp

- 状态：dp[state]表示"每个"取数(A和B共同作用的结果)状态下的输赢。state 是当前状态，用二进制表示，1代表对应位的这个数字选过了，0代表没选过。01,10,11分别表示：A和B已经选了1，已经选了2，已经选了1、2的状态。

```
class Solution {
    public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
        // 第一个选的数字就能直接赢
        if(maxChoosableInteger >= desiredTotal) return true;
        // 全部选完了也赢不了
        if((1 + maxChoosableInteger) * maxChoosableInteger / 2 < desiredTotal)
        return false;
        Boolean[] dp = new Boolean[1 << maxChoosableInteger];
        return dfs(maxChoosableInteger, desiredTotal, 0, dp);
    }

    public boolean dfs(int maxChoosableInteger, int desiredTotal, int state,
        Boolean[] dp){
        // 记忆化
        // 包装类的默认值为null
        if(dp[state] != null){
            return dp[state];
        }
        for(int i = 1; i <= maxChoosableInteger; i++){
            //当前选的位
```

```

        int tmp = (1 << (i - 1));
        //如果当前位没有被选过
        if((tmp & state) == 0){
            // A能赢:
            // 1.当前选了i就直接赢了
            // 2.当前选了i之后, 后面对方不管怎么选都输
            if(i >= desiredTotal || !dfs(maxChoosableInteger, desiredTotal -
i, tmp | state, dp)){
                dp[state] = true;
                return true;
            }
        }
    }
    //如果选什么数字都赢不了
    dp[state] = false;
    return false;
}
}

```

### 1349. 参加考试的最大学生数

给你一个  $m * n$  的矩阵 `seats` 表示教室中的座位分布。如果座位是坏的（不可用），就用 '#' 表示；否则，用 '.' 表示。

学生可以看到左侧、右侧、左上、右上这四个方向上紧邻他的学生的答卷，但是看不到直接坐在他前面或者后面的学生的答卷。请你计算并返回该考场可以容纳的一起参加考试且无法作弊的最大学生人数。

学生必须坐在状况良好的座位上。

提示：

`seats` 只包含字符 '.' 和 '#'

$m == \text{seats.length}$

$n == \text{seats}[i].\text{length}$

$1 \leq m \leq 8$

$1 \leq n \leq 8$

#### Solution

`dp[i][j]` 表示第  $i + 1$  排(数组从0开始计数)的学生排列状态为十进制数  $j$  的情况下，前  $i$  排能够做最大学生人数加上  $j$  状态下对应的学生人数

```

class Solution {
    public int maxStudents(char[][] seats) {
        int m = seats.length;
        int n = seats[0].length;
        int[] validity = new int[m]; // 记录每一横排位置是否能坐
        int stateSize = 1 << n; // 每一横排可由学生排布的方式有  $2^n$  种
        int[][] dp = new int[m][stateSize];
        int ans = 0;
        // 初始化 validity 好的桌椅表示1, 坏的表示0
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                validity[i] = (validity[i] << 1) + (seats[i][j] == '.' ? 1 : 0);
            }
        }
        // 初始化 dp 数组
        for(int i = 0; i < m; i++){

```

```

        for(int j = 0; j < stateSize; j++){
            dp[i][j] = -1;
        }
    }

    for(int i = 0; i < m; i++){//行数
        for(int j = 0; j < stateSize; j++){//每行的排布方式
            // j的状态下能否坐下第i横排 并且 j状态下左右没人
            if(((j & validity[i]) == j) && ((j & (j >> 1)) == 0)){
                if(i == 0){
                    // 第一排
                    dp[i][j] = countOne(j);
                }else{
                    // 遍历前一排的状态
                    for(int k = 0; k < stateSize; k++){
                        // 左上方没人, 右上方没人
                        if((j & (k >> 1)) == 0 && (j & (k << 1)) == 0 &&
(dp[i - 1][k] != -1)){
                            dp[i][j] = Math.max(dp[i][j], dp[i-1][k] +
countOne(j));
                        }
                    }
                }
                ans = Math.max(ans, dp[i][j]);
            }
        }
    }
    return ans;
}

// 计算二进制中有多少个1
public int countOne(int x){
    int count = 0;
    while(x != 0){
        x &= (x - 1);
        count++;
    }
    return count;
}
}

```

## \*AcWing 291. 蒙德里安的梦想

求把NM的棋盘分割成若干个12的的长方形，有多少种方案。

例如当N=2，M=4时，共有5种方案。当N=2，M=3时，共有3种方案。

如下图所示：



### Solution



- 等价于找到所有横放  $1 \times 2$  小方格的方案数，因为所有横放确定了，那么竖放方案是唯一的。
- 用  $f[i][j]$  记录第  $i$  列第  $j$  个状态。  $j$  状态位等于1表示上一列有横放格子，本列有格子捅出来。转移方程很简单，本列的每一个状态都由上列所有“合法”状态转移过来  $f[i][j] += f[i - 1][k]$

- 两个转移条件：i列和i-1列同一行不同时捅出来；本列捅出来的状态j和上列捅出来的状态k求或，得到上列是否为奇数空行状态，奇数空行不转移（不能有连续奇数个0，否则竖着没法放）。
- 初始化条件 `f[0][0] = 1`，第0列只能是状态0，无任何格子捅出来。
- 返回 `f[m][0]`。第m+1列不能有东西捅出来。

棋盘一共有0~m-1列

`f[i][j]` 表示 前i-1列的方案数已经确定，从i-1列伸出，并且第i列的状态是j的所有方案数

`f[m][0]` 表示 前m-1列的方案数已经确定，从m-1列伸出，并且第m列的状态是0的所有方案数  
也就是m列不放小方格，前m-1列已经完全摆放好

```
import java.util.*;
public class Main{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int N = 12, M = 1 << N;
        long[][] f = new long[N][M];
        boolean[] st = new boolean[M];
        while(true) {
            int n = sc.nextInt();
            int m = sc.nextInt();
            if(n == 0 || m == 0) break;
            for(int i = 0; i < N; i++) {
                Arrays.fill(f[i], 0);
            }
            // 预处理, st[i] = true表示: 当前不存在连续奇数个0
            for(int i = 0; i < 1 << n; i++) {
                st[i] = true;
                int cnt = 0; //表示当前连续0的个数
                for(int j = 0; j < n; j++) {
                    if((i >> j & 1) == 1) { // (i >> j) & 1表示: i的二进制表示中第j位是
                        //出现了1, 连续的0中断了, 计算这一段连续0的个数是奇是偶
                        if((cnt & 1) == 1) {
                            st[i] = false; //连续奇数个0
                        }
                        cnt = 0; //重新计数
                    } else {
                        cnt++;
                    }
                }
                //加入最后一段
                if((cnt & 1) == 1) {
                    st[i] = false;
                }
            }

            f[0][0] = 1; //第0列只能是状态0, 无任何格子捅出来
            for(int i = 1; i <= m; i++) { //列数
                for(int j = 0; j < 1 << n; j++) { //第i列的状态
                    for (int k = 0; k < 1 << n; k++) { //第i - 1列的状态
                        if((j & k) == 0 && st[j | k]) {
                            f[i][j] += f[i - 1][k]; //状态转移
                        }
                    }
                }
            }
        }
    }
}
```

```

        System.out.println(f[m][0]); //第m + 1列不能有东西插出来
    }
}
}

```

## AcWing 91. 最短Hamilton路径

给定一张  $nn$  个点的带权无向图，点从  $0 \sim n-1$  标号，求起点  $0$  到终点  $n-1$  的最短Hamilton路径。Hamilton路径的定义是从  $0$  到  $n-1$  不重不漏地经过每个点恰好一次。

### Solution



### Solution

- $f[i][j]$  表示当前已经走过点的集合为  $i$ ，移动到  $j$  的所有路径的  $\min$
- 枚举倒数第二个点  $k$ ，状态转移方程就是找一个中间点  $k$ ，将已经走过点的集合  $i$  中去除掉  $j$ （表示  $j$  不在经过的点的集合中），然后再加上从  $k$  到  $j$  的权值。
- 用二进制表达已经走过点的集合  $i$ ：例如  $i$  是  $\{0,1,4\}$ ，那么  $i = 10011$
- 输出  $f[111\cdots 11(n个1)][n-1]$  表示  $0 \sim n-1$  都走过，且当前移动到  $n-1$  这个点

```

import java.util.*;
public class Main{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[][] a = new int[n][n];
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                a[i][j] = sc.nextInt();
            }
        }
        sc.close();

        int[][] f = new int[1 << n][n];
        for(int i = 0; i < 1 << n; i++) {
            for(int j = 0; j < n; j++) {
                f[i][j] = Integer.MAX_VALUE >> 1;
            }
        }
        f[1][0] = 0;

        for(int i = 0; i < 1 << n; i++) { //二进制枚举走过的点的集合
            for(int j = 0; j < n; j++) { //第j个点
                // i的第j位为1的时候(也就是集合包含点j)，f[i][j]才可以被转移
                if((i >> j & 1) == 1) {
                    for(int k = 0; k < n; k++) { //枚举倒数第二个点k
                        if((i - (1 << j) >> k & 1) == 1) { //去掉j后的集合中要包含k
                            // 走到点k的路径就不能经过点j,把j从集合中去掉 0--k--j
                            f[i][j] = Math.min(f[i][j], f[i - (1 << j)][k] +
a[k][j]);
                        }
                    }
                }
            }
        }

        System.out.println(f[(1 << n) - 1][n - 1]);
    }
}

```

```
}  
}
```

## 1681. 最小不兼容性

给你一个整数数组 `nums` 和一个整数 `k`。你需要将这个数组划分到 `k` 个相同大小的子集中，使得同一个子集里面没有两个相同的元素。

一个子集的 不兼容性 是该子集里面最大值和最小值的差。

请你返回将数组分成 `k` 个子集后，各子集 不兼容性 的 和 的 最小值，如果无法分成分成 `k` 个子集，返回 `-1`。

子集的定义是数组中一些数字的集合，对数字顺序没有要求。

### 实例

输入: `nums = [6,3,8,1,3,1,2,2]`, `k = 4`

输出: 6

解释: 最优的子集分配为 `[1,2]`, `[2,3]`, `[6,8]` 和 `[1,3]`。

不兼容性和为  $(2-1) + (3-2) + (8-6) + (3-1) = 6$ 。

### Solution

- 状态表示: 记  $f[mask]$  表示当选择的元素集合为 `mask` 时**最小**的不兼容性的和, `mask` 的第 `i` 位为 1 表示 `nums[i]` 已经被选择过, 0 表示 `nums[i]` 未被选择过。
- 状态转移: 枚举最后一个选择的子集

$$f[mask] = \min f[mask \oplus sub] + value[sub]$$

```
class Solution {  
    public int minimumIncompatibility(int[] nums, int k) {  
        int n = nums.length;  
        int m = n / k;  
        int[] dp = new int[1 << n];  
        int[] value = new int[1 << n];  
        Arrays.fill(value, -1);  
        Arrays.fill(dp, Integer.MAX_VALUE);  
        dp[0] = 0;  
        //预处理value数组, value[j]代表 j这个集合的 不兼容性  
        for(int i = 0; i < (1 << n); i++) {  
            int count = countOne(i); // 计算二进制中1的数量  
            if(count == m) { //条件1: 子集中有m个1  
                //条件2: 每个数字只能出现一次  
                Map<Integer, Integer> map = new HashMap<>(); //来存子集中每个数字出现的次数  
  
                boolean flag = true;  
                int max = Integer.MIN_VALUE;  
                int min = Integer.MAX_VALUE;  
                for(int j = 0; j < n; j++) {  
                    if((i >> j & 1) == 1) { //i的第j位=1, 代表选了nums[j]  
                        min = Math.min(min, nums[j]);  
                        max = Math.max(max, nums[j]);  
                        if(map.containsKey(nums[j])) {  
                            flag = false;  
                            break;  
                        }  
                    } else {  
                        map.put(nums[j], 1);  
                    }  
                }  
                if(flag) {  
                    value[i] = max - min;  
                }  
            }  
        }  
        //dp转移  
        for(int i = 0; i < (1 << n); i++) {  
            for(int j = 0; j < (1 << n); j++) {  
                if((i & j) == 0) {  
                    dp[i] = Math.min(dp[i], dp[j] + value[i ^ j]);  
                }  
            }  
        }  
        return dp[(1 << n) - 1] == Integer.MAX_VALUE ? -1 : dp[(1 << n) - 1];  
    }  
}
```

```

    }
    }
}
//如果这个子集合法
if(flag) {
    value[i] = max - min;//更新子集i的不兼容性
}
}
}
//进行状态转移 ,dp[i]代表当选择的元素集合为i时最小的不兼容性的和
for(int i = 0; i < (1 << n); i++) {
    if(countOne(i) % m == 0) { //当前的选择方式组合mask是否有n/k的倍数个1
        //枚举最后一个选择的子集j
        for(int j = i; j != 0; j = (j - 1) & i) {
            if(value[j] != -1 && dp[i ^ j] != Integer.MAX_VALUE) { //i^j
表示从i中去掉j这个集合
                dp[i] = Math.min(dp[i], dp[i ^ j] + value[j]);
            }
        }
    }
}
return dp[(1 << n) - 1] == Integer.MAX_VALUE ? -1 : dp[(1 << n) - 1]; //最后返回dp[1111111],表示每个数字都选了以后
}
public int countOne(int x) {
    int res = 0;
    while(x != 0) {
        x &= (x - 1);
        res++;
    }
    return res;
}
}

```

模板总结：枚举所有子集：

```

for(int i = 0; i < (1 << n); i++) {
    for(int j = i; j != 0; j = (j - 1) & i) {

```

## 1723. 完成所有工作的最短时间

给你一个整数数组 `jobs`，其中 `jobs[i]` 是完成第  $i$  项工作要花费的时间。

请你将这些工作分配给  $k$  位工人。所有工作都应该分配给工人，且每项工作只能分配给一位工人。工人的工作时间是完成分配给他们的所有工作花费时间的总和。请你设计一套最佳的工作分配方案，使工人的最大工作时间得以最小化。

返回分配方案中尽可能最小的最大工作时间。

示例 2：

输入：`jobs = [1,2,4,7,8]`,  $k = 2$

输出：11

解释：按下述方式分配工作：

1 号工人：1、2、8（工作时间 =  $1 + 2 + 8 = 11$ ）

2 号工人：4、7（工作时间 =  $4 + 7 = 11$ ）

最大工作时间是 11。

提示:

- `1 <= k <= jobs.length <= 12`
- `1 <= jobs[i] <= 10^7`

## Solution

也就是把n个数字，分成k份，每份的数量不一定相同，使得每份的最大工作时间最小。

- `dp[p][mask]` 表示：前 p 个工人为了完成作业子集 i，需要花费的最大工作时间的最小值，mask 的第 i 位为 1 表示 `jobs[i]` 已经被选择过，0 表示 `jobs[i]` 还未被选择过。
- 状态转移：  
枚举 i 的每个子集 j  
第 k 个人完成子集 j 的工作，前 k-1 个人完成子集 i ^ j

```
class Solution {
    public int minimumTimeRequired(int[] jobs, int k) {
        int n = jobs.length;
        int[][] dp = new int[k + 1][1 << n];
        int[] value = new int[1 << n];
        for(int i = 0; i <= k; i++) {
            Arrays.fill(dp[i], Integer.MAX_VALUE / 2);
        }
        dp[0][0] = 0;
        // 预处理value数组，value[j]代表 j这个集合的总工作时间
        for (int i = 0; i < (1 << n); i++) {
            for (int j = 0; j < n; j++) {
                if(((i >> j) & 1) == 1) { //如果i的第j位==1
                    value[i] += jobs[j];
                }
            }
        }
        for (int p = 1; p <= k; p++) {
            for (int i = 0; i < (1 << n); i++) {
                // 枚举最后一个选择的子集j
                for(int j = i; j != 0; j = (j - 1) & i) {
                    dp[p][i] = Math.min(dp[p][i], Math.max(dp[p - 1][i ^ j],
value[j]));
                }
            }
        }
        return dp[k][(1 << n) - 1];
    }
}
```

## 六、区间dp

### 282. 石子合并(Acwing)

#### 题目描述

设有N堆石子排成一排，其编号为1, 2, 3, ..., N。

每堆石子有一定的质量，可以用一个整数来描述，现在要将这N堆石子合并成为一堆。



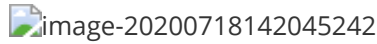
每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻，合并时由于选择的顺序不同，合并的总代价也不相同。

例如有4堆石子分别为 1 3 5 2，我们可以先合并1、2堆，代价为4，得到4 5 2，又合并 1, 2堆，代价为9，得到9 2，再合并得到11，总代价为4+9+11=24；

如果第二步是先合并2, 3堆，则代价为7，得到4 7，最后一次合并代价为11，总代价为4+7+11=22。

问题是：找出一种合理的方法，使总的代价最小，输出最小代价。

## Solution



```
import java.util.*;

public class Main{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] nums = new int[n + 1];
        int[] s = new int[n + 1];
        //前缀和
        for(int i = 1; i <= n; i++) {
            nums[i] = sc.nextInt();
            s[i] = s[i - 1] + nums[i];
        }
        sc.close();
        //dp[i][j] 代表将[i, j]合并的所有方案中的最小代价
        int[][] dp = new int[n + 1][n + 1];
        for(int len = 2; len <= n; len++) { //枚举区间长度
            for(int i = 1; i + len - 1 <= n; i++) { //枚举左端点
                int j = i + len - 1;
                dp[i][j] = Integer.MAX_VALUE;
                //枚举最后一个合并的点k（分割点）
                for(int k = i; k < j; k++) {
                    dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k + 1][j] + s[j]
- s[i - 1]);
                }
            }
        }
        System.out.println(dp[1][n]);
    }
}
```

## AcWing 1068. 环形石子合并

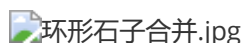
将  $n$  堆石子绕圆形操场排放，现要将石子有序地合并成一堆。

规定每次只能选相邻的两堆合并成新的一堆，并将新的一堆的石子数记做该次合并的得分。

请编写一个程序，读入堆数  $n$  及每堆的石子数，并进行如下计算：

- 选择一种合并石子的方案，使得做  $n-1$  次合并得分总和最大。
- 选择一种合并石子的方案，使得做  $n-1$  次合并得分总和最小。

## Solution



```

import java.util.*;
public class Main{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] a = new int[2 * n + 1];
        int[] s = new int[2 * n + 1]; //前缀和
        int[][] dpMax = new int[2 * n + 1][2 * n + 1];
        int[][] dpMin = new int[2 * n + 1][2 * n + 1];
        for(int i = 1; i <= n; i++) {
            a[i] = sc.nextInt();
            a[i + n] = a[i];
        }
        for(int i = 1; i <= 2 * n; i++) {
            s[i] = s[i - 1] + a[i];
            Arrays.fill(dpMax[i], Integer.MIN_VALUE);
            Arrays.fill(dpMin[i], Integer.MAX_VALUE);
        }
        sc.close();
        for(int len = 1; len <= n; len++) { //枚举区间
            for(int i = 1; i + len - 1 <= 2 * n; i++) { //左端点
                int j = i + len - 1; //右端点
                if(len == 1) {
                    dpMax[i][j] = dpMin[i][j] = 0;
                } else {
                    for(int k = i; k < j; k++) { //分割点
                        dpMax[i][j] = Math.max(dpMax[i][j], dpMax[i][k] +
                        dpMax[k + 1][j] + s[j] - s[i - 1]);
                        dpMin[i][j] = Math.min(dpMin[i][j], dpMin[i][k] +
                        dpMin[k + 1][j] + s[j] - s[i - 1]);
                    }
                }
            }
        }
        int max = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE;
        for(int i = 1; i <= n; i++) {
            max = Math.max(max, dpMax[i][i + n - 1]);
            min = Math.min(min, dpMin[i][i + n - 1]);
        }
        System.out.println(min);
        System.out.println(max);
    }
}

```

## AcWing 320. 能量项链

在Mars星球上，每个Mars人都随身佩带着一串能量项链，在项链上有  $N$  颗能量珠。

能量珠是一颗有头标记与尾标记的珠子，这些标记对应着某个正整数。

并且，对于相邻的两颗珠子，前一颗珠子的尾标记一定等于后一颗珠子的头标记。

因为只有这样，通过吸盘（吸盘是Mars人吸收能量的一种器官）的作用，这两颗珠子才能聚合成一颗珠子，同时释放出可以被吸盘吸收的能量。

如果前一颗能量珠的头标记为  $m$ ，尾标记为  $r$ ，后一颗能量珠的头标记为  $r$ ，尾标记为  $n$ ，则聚合后释放的能量为  $mrn$ （Mars单位），新产生的珠子的头标记为  $m$ ，尾标记为  $n$ 。

需要时，Mars人就用吸盘夹住相邻的两颗珠子，通过聚合得到能量，直到项链上只剩下一颗珠子为止。

显然，不同的聚合顺序得到的总能量是不同的，请你设计一个聚合顺序，使一串项链释放出的总能量最大。

例如：设 $N=4$ ，4颗珠子的头标记与尾标记依次为(2, 3) (3, 5) (5, 10) (10, 2)。

我们用记号 $\oplus$ 表示两颗珠子的聚合操作， $(j \oplus k)$ 表示第 $j$ ,  $k$ 两颗珠子聚合后所释放的能量。则

第4、1两颗珠子聚合后释放的能量为： $(4 \oplus 1) = 10 * 2 * 3 = 60$ 。

这一串项链可以得到最优值的一个聚合顺序所释放的总能量为 $((4 \oplus 1) \oplus 2) \oplus 3$   
 $= 10 * 2 * 3 + 10 * 3 * 5 + 10 * 5 * 10 = 710$ 。

## Solution

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sr = new Scanner(System.in);
        int n = sr.nextInt();
        int N = 210;
        int[] a = new int[N];
        int[][] dp = new int[N][N];
        for (int i = 1; i <= n; i++) {
            a[i] = sr.nextInt();
            a[i + n] = a[i];
        }
        for (int len = 3; len <= n + 1; len++) {
            for (int l = 1; l + len - 1 <= n * 2; l++) {
                int r = l + len - 1;
                for (int k = l + 1; k <= r - 1; k++) {
                    dp[l][r] = Math.max(dp[l][r], dp[l][k] + dp[k][r] + a[l] *
a[k] * a[r]);
                }
            }
        }
        int res = 0;
        for (int l = 1; l <= n; l++) {
            res = Math.max(res, dp[l][l + n]);
        }
        System.out.println(res);
    }
}
```

## 664. 奇怪的打印机

### 题目描述

有台奇怪的打印机有以下两个特殊要求：

打印机每次只能打印同一个字符序列。

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'。

## Solution

- 状态: `dp[i][j]` 表示打印 `[i,j]` 的字符需要的最少打印次数
- 状态转移: 如果 `s[j]` 在前面 `[i,j-1]` 出现过, 则可以跟着前面一起打印出来。

```
class Solution {
    public int strangePrinter(String s) {
        if(s == null || s.length() <= 0) return 0;
        int n = s.length();
        int[][] dp = new int[n][n];
        for(int i = 0; i < n; i++){
            dp[i][i] = 1;
        }
        for(int len = 2; len <= n; len++){//枚举区间
            for(int i = 0; i + len - 1 < n; i++){//枚举左端点
                int j = i + len - 1;//右端点
                char sj = s.charAt(j);
                // sj和前面一个区间[i,j-1]的第一个字符相等,可以直接顺带打印出来
                if(sj == s.charAt(i)){
                    dp[i][j]=dp[i][j-1];
                    continue;
                }
                dp[i][j] = dp[i][j-1] + 1;//最多: 右边的字母单独打印
                //枚举[i, j-1]中的字母, 如果跟s[j]相等就可以一起打印,从这个位置分割字符串
                for(int k = i+1; k < j; k++){
                    // System.out.println("k=" + k);
                    if(s.charAt(k) == sj){
                        dp[i][j] = Math.min(dp[i][j], dp[i][k-1] + dp[k][j-1]);
                    }
                }
            }
        }
        return dp[0][n-1];
    }
}
```

```
class Solution {
    public int strangePrinter(String s) {
        //dp[i][j]代表打印[i,j]区间内的字符所需的最少次数
        if(s == null || s.length() <= 0) return 0;
        int n = s.length();
        int[][] dp = new int[n+1][n+1];
```

```

        for(int len = 1; len <= n; len++){
            for(int i = 0; i + len - 1 < n; i++){
                int j = i + len - 1;
                dp[i][j] = dp[i+1][j] + 1; //最大: s[i]单独打印一次
                for(int k = i + 1; k <= j; k++){
                    if(s.charAt(k) == s.charAt(i)){
                        dp[i][j] = Math.min(dp[i][j], dp[i][k-1] + dp[k+1]
[j]); //在打印s[i]的时候可以把s[k]一起打印出来
                    }
                }
            }
        }
        return dp[0][n-1];
    }
}

```

## 312. 戳气球

难度：困难

### 题目描述

有  $n$  个气球，编号为  $0$  到  $n-1$ ，每个气球上都标有一个数字，这些数字存在数组 `nums` 中。

现在要求你戳破所有的气球。如果你戳破气球  $i$ ，就可以获得 `nums[left] * nums[i] * nums[right]` 个硬币。这里的 `left` 和 `right` 代表和  $i$  相邻的两个气球的序号。注意当你戳破了气球  $i$  后，气球 `left` 和气球 `right` 就变成了相邻的气球。

求所能获得硬币的最大数量。

### 说明:

- 你可以假设 `nums[-1] = nums[n] = 1`，但注意它们不是真实存在的所以并不能被戳破。
- $0 \leq n \leq 500$ ,  $0 \leq \text{nums}[i] \leq 100$

### 示例:

输入: `[3,1,5,8]`

输出: `167`

解释: `nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []`  
`coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167`

## Solution


动态规划 区间dp

- 状态: `dp[i][j]` 代表戳破 `[i, j]` 的气球所能获得硬币的最大数量。
- 状态转移: 枚举区间——左端点——最后一个戳破的气球

```

dp[i][j] = Math.max(dp[i][j], dp[i][k-1] + dp[k+1][j] + expand[i-1] * expand[k]
* expand[j+1]);

```

 image-20200719111703228

```

class Solution {
    public int maxCoins(int[] nums) {

```

```

        if(nums == null || nums.length == 0){
            return 0;
        }
        int n = nums.length;
        int[][] dp = new int[n+2][n+2];
        //扩展nums[-1] = nums[n] = 1
        int[] expand = new int[n+2];
        expand[0] = 1;
        expand[n+1] = 1;
        for(int i = 0; i < n; i++){
            expand[i+1] = nums[i];
        }

        for(int len = 1; len <= n; len++){//枚举区间
            for(int i = 1; i + len - 1 <= n; i++){//左端点
                int j = i + len - 1;//右端点
                for(int k = i; k <= j; k++){//枚举最后一个戳破的气球
                    dp[i][j] = Math.max(dp[i][j], dp[i][k-1] + dp[k+1][j] +
                    expand[i-1] * expand[k] * expand[j+1]);
                }
            }
        }
        return dp[1][n];
    }
}

```

### 1039. 多边形三角剖分的最低得分

给定  $N$ ，想象一个凸  $N$  边多边形，其顶点按顺时针顺序依次标记为  $A[0]$ ,  $A[1]$ , ...,  $A[N-1]$ 。

假设您将多边形剖分为  $N-2$  个三角形。对于每个三角形，该三角形的值是顶点标记的乘积，三角剖分的分数是进行三角剖分后所有  $N-2$  个三角形的值之和。

返回多边形进行三角剖分后可以得到的最低分。

#### Solution

区间dp

```

class Solution {
    public int minScoreTriangulation(int[] A) {
        //dp[i][j]代表[i, j]区间的最小分数
        int n = A.length;
        int[][] dp = new int[n][n];
        for(int len = 3; len <= n; len++){//区间长度
            for(int i = 0; i + len - 1 < n; i++){//左端点
                int j = i + len - 1;
                dp[i][j] = Integer.MAX_VALUE;
                for(int k = i + 1; k < j; k++){//分割点
                    dp[i][j] = Math.min(dp[i][j], dp[i][k] + A[i] * A[k] * A[j]
                    + dp[k][j]);
                }
            }
        }
        return dp[0][n-1];
    }
}

```

## AcWing 479. 加分二叉树

设一个 $n$ 个节点的二叉树 $tree$ 的中序遍历为 $(1, 2, 3, \dots, n)$ ，其中数字 $1, 2, 3, \dots, n$ 为节点编号。

每个节点都有一个分数（均为正整数），记第 $i$ 个节点的分数为 $d_i$ ， $tree$ 及它的每个子树都有一个加分，任一棵子树 $subtree$ （也包含 $tree$ 本身）的加分计算方法如下：

$subtree$ 的左子树的加分  $\times$   $subtree$ 的右子树的加分  $+$   $subtree$ 的根节点的分数

若某个子树为空，规定其加分为1。叶子的加分就是叶节点本身的分数，不考虑它的空子树。

试求一棵符合中序遍历为 $(1, 2, 3, \dots, n)$ 且加分最高的二叉树 $tree$ 。

要求输出：

(1)  $tree$ 的最高加分

(2)  $tree$ 的前序遍历

**Solution**

```
import java.util.*;
public class Main {
    public static int[][] g;
    public static void main(String[] args) {
        Scanner sr = new Scanner(System.in);
        int n = sr.nextInt();
        int[] a = new int[n + 1];
        for (int i = 1; i <= n; i++) {
            a[i] = sr.nextInt();
        }
        int[][] dp = new int[n + 1][n + 1]; // dp[i][j]代表中序遍历为[i, j]时的最高加分
        g = new int[n + 1][n + 1]; // 记录对应的根节点在哪里
        for (int i = 1; i <= n; i++) {
            dp[i][i] = a[i];
            g[i][i] = i;
        }
        for (int len = 2; len <= n; len++) { // 遍历区间长度
            for (int i = 1; i + len - 1 <= n; i++) { // 遍历左端点
                int j = i + len - 1; // 右端点
                // [i, j]
                // 根节点的不同位置
                for (int k = i; k <= j; k++) {
                    int left = k == i ? 1 : dp[i][k - 1];
                    int right = k == j ? 1 : dp[k + 1][j];
                    int score = left * right + a[k];
                    if (dp[i][j] < score) {
                        g[i][j] = k;
                        dp[i][j] = score;
                    }
                }
            }
        }
        System.out.println(dp[1][n]);
        dfs(1, n);
    }
    public static void dfs(int l, int r) {
        if (l > r) {
            return;
        }
    }
```

```

        int root = g[l][r];
        System.out.print(root+ " ");
        dfs(l, root - 1);
        dfs(root + 1, r);
    }
}

```

## 七、树形dp

### 337. 打家劫舍 III

难度：中等

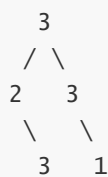
#### 题目描述

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例：

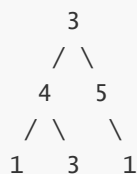
输入：[3,2,3,null,3,null,1]



输出：7

解释：小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7。

输入：[3,4,5,1,3,null,1]



输出：9

解释：小偷一晚能够盗取的最高金额 = 4 + 5 = 9。

#### Solution

树形结构的动态规划。

对每个节点来说，有两种状态，偷和不偷，两者取max。

用 `int[] result = new int[2]` 来存两种状态下的收益。

1. 偷当前节点：最大收益 = 当前节点的钱 + 不偷左儿子的钱 + 不偷右儿子的钱
2. 不偷当前节点：最大收益 = 左儿子所能获得的最大收益 + 右儿子所能获得的最大收益



其中, 儿子所能获得的最大收益 =  $\max(\text{偷当前儿子所能获得的最大收益}, \text{不偷当前儿子所能获得的最大收益})$

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int rob(TreeNode root) {
        int[] result = countSum(root);
        return Math.max(result[0], result[1]);
    }
    public int[] countSum(TreeNode root){
        int[] result = new int[2];
        if(root == null){
            return result;
        }
        // 当前节点左儿子偷与不偷所能获得的收益
        int[] left = countSum(root.left);
        // 当前节点右儿子偷与不偷所能获得的收益
        int[] right = countSum(root.right);
        // 不偷当前节点的收益 = 左儿子所能获得的最大收益 + 右儿子所能获得的最大收益
        result[0] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
        // 偷当前节点的收益 = 当前节点的钱 + 不偷左儿子的钱 + 不偷右儿子的钱
        result[1] = root.val + left[0] + right[0];
        return result;
    }
}
```

## AcWing 285. 没有上司的舞会

Ural大学有N名职员, 编号为1~N。


他们的关系就像一棵以校长为根的树, 父节点就是子节点的直接上司。

每个职员有一个快乐指数, 用整数  $H_i$  给出, 其中  $1 \leq i \leq N$ 。

现在要召开一场周年庆宴会, 不过, 没有职员愿意和直接上司一起参会。

在满足这个条件的前提下, 主办方希望邀请一部分职员参会, 使得所有参会职员的快乐指数总和最大, 求这个最大值。

### Solution

 没有上司的舞会.png

```
import java.io.*;
import java.util.*;
public class Main {
    static int N = 6010;
    static int idx;
    static int[] h = new int[N];
    static int[] e = new int[N];
```

```

static int[] ne = new int[N];
static int[] d = new int[N];
static int[][] dp = new int[N][2];
static int[] happy = new int[N];

public static void add(int a, int b) {
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx++;
}

public static void dfs(int u) {
    for(int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i]; //所有子节点
        dfs(j);
        dp[u][0] += Math.max(dp[j][1], dp[j][0]); //不选当前点u, 子节点可以选也可以不选, 取最大
        dp[u][1] += dp[j][0]; //选当前点u, 子节点一定不能选
    }
    dp[u][1] += happy[u]; //选当前点u, 还要加上当前点的happy值
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
    int n = Integer.parseInt(br.readLine());
    for(int i = 1; i <= n; i++) {
        happy[i] = Integer.parseInt(br.readLine());
    }
    Arrays.fill(h, -1);
    for(int i = 0; i < n - 1; i++) {
        String[] str = br.readLine().split(" ");
        int a = Integer.parseInt(str[0]);
        int b = Integer.parseInt(str[1]);
        add(b, a);
        d[a]++; //入度
    }
    br.close();

    //找哪个是根节点 入度为0
    int root = 1;
    while(root <= n) {
        if(d[root] == 0) {
            break;
        }else {
            root++;
        }
    }
    dfs(root);

    System.out.println(Math.max(dp[root][0], dp[root][1])); //返回选根节点/不选根
节点的 最大值
    }
}

```

## 八、计数类DP

### AcWing 900. 整数划分

一个正整数 $n$ 可以表示成若干个正整数之和，形如： $n=n_1+n_2+\dots+n_k$ ，其中 $n_1\geq n_2\geq\dots\geq n_k, k\geq 1$ 。

我们将这样的一种表示称为正整数 $n$ 的一种划分。

现在给定一个正整数 $n$ ，请你求出 $n$ 共有多少种不同的划分方法。

image-20201126150235865

```
import java.util.*;
public class Main{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        sc.close();
        //dp[i][j]表示所有总和为i，恰好分成j个数字的方案数
        int[][] dp = new int[n + 1][n + 1];
        dp[0][0] = 1;
        for(int i = 1; i <= n; i++) {
            for(int j = 1; j <= i; j++) {
                dp[i][j] = (dp[i - 1][j - 1] + dp[i - j][j]) % 1000000007;
            }
        }
        int ans = 0;
        for(int j = 1; j <= n; j++) {
            ans = (ans + dp[n][j]) % 1000000007;
        }
        System.out.println(ans);
    }
}
```

## 八、数位统计dp

### AcWing 338. 计数问题

给定两个整数 $a$ 和 $b$ ，求 $a$ 和 $b$ 之间的所有数字中0~9的出现次数。

例如， $a=1024$ ， $b=1032$ ，则 $a$ 和 $b$ 之间共有9个数如下：

1024 1025 1026 1027 1028 1029 1030 1031 1032

其中'0'出现10次，'1'出现10次，'2'出现7次，'3'出现3次等等...

```
import java.io.*;
import java.util.*;
public class Main{
    // 统计从1-n，x出现的次数
    public static int count(int n, int x) {
        if(n == 0) return 0;
        List<Integer> list = new ArrayList<Integer>();
        //将n的每一位加入到list中
        while(n != 0) {
            list.add(n % 10);
            n /= 10;
        }
    }
}
```

```

    }
    int len = list.size();

    int res = 0;
    //对于0,不需要枚举最高位
    for(int i = len - 1 - (x == 0 ? 1 : 0); i >= 0; i--) {
        if(i < len - 1) {
            res += get(list, len - 1, i + 1) * power10(i);
            if(x == 0) {
                res -= power10(i);
            }
        }
        if(list.get(i) > x) {
            res += power10(i);
        }else if(list.get(i) == x) {
            res += get(list, i-1, 0) + 1;
        }
    }
    return res;
}

public static int power10(int x) {
    int res = 1;
    while(x != 0) {
        res *= 10;
        x--;
    }
    return res;
}

public static int get(List<Integer> list, int l, int r) {
    int res = 0;
    for(int i = l; i >= r; i--) {
        res = res * 10 + list.get(i);
    }
    return res;
}

public static void main(String[] args) throws IOException {
    BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
    String[] str = in.readLine().split(" ");
    int a = Integer.parseInt(str[0]);
    int b = Integer.parseInt(str[1]);
    while(a > 0 || b > 0) {
        if(a > b) {
            int tmp = a;
            a = b;
            b = tmp;
        }
        //枚举每个数字
        for(int i = 0; i <= 9; i++) {
            System.out.print(count(b, i) - count(a - 1, i) + " ");
        }
        System.out.println();
        str = in.readLine().split(" ");
        a = Integer.parseInt(str[0]);
    }
}

```

```
        b = Integer.parseInt(str[1]);  
    }  
}  
}
```

## 九、单调队列优化dp

## 十、斜率优化dp