动态规划

动态规划的一般流程：暴力的递归解法->带备忘录的递归解法->迭代的动态规划解法

动态规划问题的一般形式就是求最值，（最差的情况下就可能用for循环，直接暴力求解了）

动态规划的核心问题是穷举，因为求最值，肯定要把所有的答案都穷举出来，然后在其中找最值。

穷举所有可行解其实不是很容易的事，只有列出正确的【状态转移方程】，才能正确地穷举。

手把手刷动态规划

1.动态规划解题套路框架

```
# 初始化 base case
dp[0][0][...] = base
# 进行状态转移
for 状态1 in 状态1的所有取值：
    for 状态2 in 状态2的所有取值：
        for ...
            dp[状态1][状态2][...] = 求最值(选择1，选择2...)
```

a.Leetcode509 裴波那契数

暴力递归

```
class Solution {
    public int fib(int n) {
        if(n<=1){
            return n;
        }
        return fib(n-1)+fib(n-2);
    }
}
```

动态规划

```
class Solution {
    public int climbStairs(int n) {
        if(n==1) {
                return n;
        }
        int[] bp = new int[n];
        bp[0] = 1;
        bp[1] = 2;
        for(int i=2;i<n;i++) {
            bp[i] = bp[i-1] + bp[i-2];
        }
        return bp[n-1];
    }
}
```

> 类似问题
>
> Leetcode70 爬楼梯（类似于斐波那契书写，相同，都是当前的由之前的前一个 前前一个来确定）
>
> Leetcode746 使用最小花费爬楼梯(遍历所有，找到最小的即可)

```
class Solution {
 public int minCostClimbingStairs(int[] cost) {
     //动态规划以数组i结尾的最小花费
     int n = cost.length;
     int[] dp = new int[n];
     dp[0]  = cost[0];
     dp[1]  = cost[1];
     // 大于两个之后
     for(int i=2;i<n;i++){
         // 当前的花费为
         dp[i] = Math.min(dp[i-1],dp[i-2]) + cost[i];
     }
     return Math.min(dp[n-1],dp[n-2]);
 }
}
```

b.Leetcode322零钱兑换(求最小兑换数量)

dp的含义就是用前i个搭配amount的最小数量，求最小值(初始化)

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        // dp的含义 用前i个数据，配置成amoun的金额数量 任意数量
        int n = coins.length;
        int[][] dp = new int[n+1][amount+1];
        // 初始化
        for(int i=0;i<n;i++){
            // 赋值最大值
            Arrays.fill(dp[i],amount+1);
            // 当amount为0时，没有办法配置出来
            if(i>0){
                dp[i][0] = 0;
            }
        }
        // 转移方程
        for(int i=1;i<=n;i++){
            for(int j=1;j<=amount;j++){
                if(j>=coins[i-1]){
                    // 能兑换
                    dp[i][j] = Math.min(dp[i-1][j],dp[i][j-coins[i-1]]+1);
                }else{
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[n][amount]==amount+1?-1:dp[n][amount];

    }
}
```

2.动归和回溯对比

a.Leetcode494目标和

给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 -中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

示例：

输入：nums: [1, 1, 1, 1, 1], S: 3
输出：5

解题思路：回溯解法

```
class Solution {
    // 结果
    int res = 0;
    public int findTargetSumWays(int[] nums, int S) {
        dfs(nums,0,S);
        return res;
    }

    // 开始回溯
    public void dfs(int[]nums,int i,int S){
        // 遍历到最后了
        if(i==nums.length){
            if(S==0){
                res++;
            }
            return;
        }

        // 每个数字两种状态
        S += nums[i];
        dfs(nums,i+1,S);
        S -= nums[i];

        S -= nums[i];
        dfs(nums,i+1,S);
        S += nums[i];

    }
}
```

解题思路：动态规划

```
class Solution {
    public int findTargetSumWays(int[] nums, int S) {
        int sum = 0;
        for(int num:nums){
            sum += num;
        }
        // 对其求和
        if((sum+S)%2!=0 || S>sum){
```

```
            return 0;
        }
        int target = (sum+S)/2;
        // 从中挑选几个数，使其等于target
        int n = nums.length;
        // target中存储了用前几个数，得到target的值
        int[][] dp = new int[n+1][target+1];
        // 初始化
        for(int i=0;i<=n;i++){
            // 当target恰好减少为0就为一次
            dp[i][0] = 1;
        }

        // 转移方程
        for(int i=1;i<=n;i++){
            for(int j=0;j<=target;j++){
                if(j>=nums[i-1]){
                    // 前一个装进去或者不装进去
                    dp[i][j] = dp[i-1][j] + dp[i-1][j-nums[i-1]];
                }else{
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[n][target];

    }
}
```

子序列问题

a.Leetcode72 编辑距离(最少操作数)

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

```
class Solution {
    public int minDistance(String word1, String word2) {
        int m = word1.length();
        int n = word2.length();
        // dp m n就是前i个变成第j个所需
        int[][] dp = new int[m+1][n+1];
        //初始化
        for(int i=0;i<=m;i++){
            dp[i][0] = i;
        }
        for(int j=0;j<=n;j++){
            dp[0][j] = j;
        }

        // 转移
        for(int i=1;i<=m;i++){
```

```
            for(int j=1;j<=n;j++){
                // 相等了
                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],Math.min(dp[i-1][j-1],dp[i]
[j-1]))+1;
                }
            }
        }

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

b.Leetcode300最长递增子序列问题

> 子序列是不连续的，求最长，dp的含义是什么

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        // 最长递增子序列问题
        int n = nums.length;
        // 动态规划，以n结尾的最长子序列
        int[] dp = new int[n];
        // 初始化的值为1
        Arrays.fill(dp,1);
        // 最长的子序列不一定在哪个数组里面
        int res = 1;
        // 开始
        for(int i=1;i<n;i++){
            //找前面的值
            for(int j=0;j<i;j++){
                // 判断是否更改
                if(nums[j]<nums[i]){
                    dp[i] = Math.max(dp[i],dp[j]+1);
                    res =  Math.max(res,dp[i]);
                }
            }
        }
        return res;
    }
}
```

c.Leetcode673最长递增子序列(数量问题有几个)

在上述的基础上，添加一个新的数组

```
class Solution {
    public int findNumberOfLIS(int[] nums) {
        //dp以n结尾的数字
        int n = nums.length;
        int[] dp = new int[n];
        // 增加一个数组
        int[] combinations = new int[n];
        // 对其初始化
```

```
        Arrays.fill(combinations,1);
        // 初始化
        Arrays.fill(dp,1);
        // 最长
        int  res = 1;
        // 转移方程
        for(int i=1;i<n;i++){
            for(int j=0;j<i;j++){
                if(nums[i]>nums[j]){
                    // 递增
                    // 里面增加了个判断
                    if(dp[j]+1>dp[i]){
                        dp[i] = dp[j] + 1;
                        combinations[i] = combinations[j];
                    }else if(dp[j]+1==dp[i]){
                        combinations[i] += combinations[j];
                    }
                }
            }
            res = Math.max(res,dp[i]);
        }
        // 再次遍历
        int count = 0;
        for(int i=0;i<n;i++){
            if(dp[i]==res){
                count+=combinations[i];
            }
        }
        return count;
    }
}
```

d.Leetcode384俄罗斯套娃信封问题(同最长递增子序列问题,加了一步排序)

```
class Solution {
    public int maxEnvelopes(int[][] envelopes) {
        // 最长递增子序列问题
        // 先排序
        Arrays.sort(envelopes,(a,b)->(a[0]-b[0]));
        // 开始
        int n = envelopes.length;
        int[] dp = new int[n];
        int maxLen = 1;
        // 初始化
        Arrays.fill(dp,1);
        // 开始
        for(int i=1;i<n;i++){
            // 判断前面的
            for(int j=0;j<i;j++){
                // 能否放进去
                if(envelopes[i][0]>envelopes[j][0]&&envelopes[i][1]>envelopes[j]
[1]){
                    dp[i] = Math.max(dp[i],dp[j]+1);
                    // 最长的更新
                    maxLen = Math.max(maxLen,dp[i]);
                }
            }
```

```
        }
        return maxLen;
    }
}
```

e.Leetcode53最大和的连续子数组

```
class Solution {
    public int maxSubArray(int[] nums) {
        // 最大和的连续子数组
        int max = nums[0];
        int n = nums.length;
        for(int i=1;i<n;i++){
            nums[i] = Math.max(nums[i],nums[i-1]+nums[i]);
            max = Math.max(max,nums[i]);
        }
        return max;
    }
}
```

最长公共子序列问题(LCS问题)

a.Leetcode1143最长公共子序列(经典-子序列问题(long common subsequence,LCS问题))

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。
```

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length();
        int n = text2.length();
        // 开辟空间
        int[][] dp = new int[m+1][n+1];
        // 无需初始化
        for(int i=1;i<=m;i++){
            for(int j=1;j<=n;j++){
                // 判断
                if(text1.charAt(i-1)==text2.charAt(j-1)){
                    // 相等，
                    dp[i][j] = 1+dp[i-1][j-1];
                }else{
                    // 不相等 删除操作
                    dp[i][j] = Math.max(dp[i-1][j],dp[i][j-1]);
                }
            }
        }
        return dp[m][n];

    }
}
```

b.Leetcode583两个字符串的删除操作

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"
```

> 使得相等，用删除的动作，找到最长公共子序列，
>
> 后用两个长度-两个公共就是需要操作的步骤

```
class Solution {
    public int minDistance(String word1, String word2) {
        int m = word1.length();
        int n = word2.length();
        int len = lcs(word1,word2);
        return m+n-len-len;
    }

    public int lcs(String s1,String s2){
        int m = s1.length();
        int n = s2.length();
        // 开辟数组
        int[][] dp = new int[m+1][n+1];
        // 默认初始化
        for(int i=1;i<=m;i++){
            for(int j=1;j<=n;j++){
                // 判断是否相等
                if(s1.charAt(i-1)==s2.charAt(j-1)){
                    dp[i][j] = 1+dp[i-1][j-1];
                }else{
                    dp[i][j] = Math.max(dp[i-1][j],dp[i][j-1]);
                }
            }
        }
        return dp[m][n];
    }
}
```

c.Leetcode712两个字符串的最小ASCII删除和

给定两个字符串s1, s2，找到使两个字符串相等所需删除字符的ASCII值的最小和。

示例 1:

输入: s1 = "sea", s2 = "eat"
输出: 231
解释: 在 "sea" 中删除 "s" 并将 "s" 的值(115)加入总和。
在 "eat" 中删除 "t" 并将 116 加入总和。
结束时，两个字符串相等，115 + 116 = 231 就是符合条件的最小和。

> 解题思路：使其相等，用删除操作，用最长公共子序列

```
class Solution {
    public int minimumDeleteSum(String s1, String s2) {
        // 计算最长公共前缀的asicii的话
        int sum_ascii = lcs(s1,s2);
        int sum_1 = 0;
        int sum_2 = 0;
        for(int i=0;i<s1.length();i++){
            sum_1 += (int)s1.charAt(i);
        }
        for(int j=0;j<s2.length();j++){
            sum_2 += (int)s2.charAt(j);
        }
        int res = sum_1+sum_2-sum_ascii-sum_ascii;
        return res;

    }

    // 最长公共子串前缀
    public int lcs(String s1,String s2){
        // dp里面存ascii
        int m = s1.length();
        int n = s2.length();
        int[][] dp = new int[m+1][n+1];
        // 转移方程
        for(int i=1;i<=m;i++){
            for(int j=1;j<=n;j++){
                //判断
                if(s1.charAt(i-1)==s2.charAt(j-1)){
                    dp[i][j] = (int)s1.charAt(i-1)+dp[i-1][j-1];
                }else{
                    dp[i][j] = Math.max(dp[i-1][j],dp[i][j-1]);
                }
            }
        }
        return dp[m][n];
    }
}
```

子序列解题模板

第一种模板是一个一维的dp数组（最长递增子序列）

```
int n = arr.length;
int[] dp = new int[n];
for(int i=1;i<n;i++){
    for(int j=0;j<i;j++){
        dp[i] = 最值(dp[i],dp[j])+....
    }
}
```

第二种模板是一个二维的dp数组

```
int n = arr.length;
int[][] dp = new int[n][n];
for(int i=1;i<n;i++){
    for(int j=1;j<n;j++){
        if(arr[i]==arr[j]){
            dp[i][j] = ...
        }else{
            dp[i][j] = 最值()
        }
    }
}
```

> 第二种常用在，两个字符串/数组或者只涉及一种字符串/数组对的情况(例如最长回文子序列)

a.判断回文链表

请判断一个链表是否为回文链表。

示例 1:

```
输入: 1->2
输出: false
```

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public boolean isPalindrome(ListNode head) {
        if(head.next==null){
            return true;
        }
        // 用O(n)和O(1)的时间复杂度原链表上操作
        // 解题思路：先找中间的链表结点，之后后续反转，然后两边从头比较
        ListNode medium = findMedium(head);
        //对其处理
        ListNode node = medium.next;
        // 断开
        medium.next = null;
        // 对其进行翻转
        ListNode newhead = reverse(node);

        // 开始遍历判断
        while(head!=null&&newhead!=null){
            if(head.val!=newhead.val){
                return false;
            }
            head = head.next;
            newhead = newhead.next;
        }
```

```
        // 判断是否都为空
        return true;

    }
    public ListNode reverse(ListNode node){
        ListNode pre = null;
        ListNode next = null;
        while(node!=null){
            next = node.next;
            node.next = pre;
            pre = node;
            node = next;
        }
        return pre;
    }

    // 寻找中间的结点
    public ListNode findMedium(ListNode head){
        ListNode slow = head;
        ListNode fast = head;
        while(fast.next!=null&&fast.next.next!=null){
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;

    }
}
```

b.Leetcode005 最长回文子串(注意是子串)

> 解题思路：中心扩散法

```
class Solution {
    // 子串
    public String longestPalindrome(String s) {
        // 中心扩散法
        char[] arr = s.toCharArray();
        // 最长子串记录
        String res = "";
        // 中心扩散法
        for(int i=0;i<arr.length;i++){
            String res1 = check(s,i,i);
            String res2 = check(s,i,i+1);

            res = res.length()>res1.length()?res:res1;
            res = res.length()>res2.length()?res:res2;
        }
        return res;
    }
    // 开始
    public String check(String s,int left,int right){
        while(left>=0&&right<=s.length()-1&&s.charAt(left)==s.charAt(right)){
            left--;
            right++;
```

```
        }
        // 返回
        return s.substring(left+1,right);

    }
}
```

c.最长回文子序列

正方形，对角线为1，从底往上走，右上角的值是答案。结果当前的结果取决于前一个的结果

```
class Solution {
    public int longestPalindromeSubseq(String s) {
        // 最长公共子序列用二维
        char[] arr = s.toCharArray();
        int n = arr.length;
        // dp代表以i和j为两端的回文子序列
        int[][] dp = new int[n][n];
        // 初始化 只有一个数的时候是1
        for(int i=0;i<n;i++){
            dp[i][i] = 1;
        }
        // 从底往上 从左往右
        for(int i=n-1;i>=0;i--){
            for(int j=i+1;j<n;j++){
                if(arr[i]==arr[j]){
                    dp[i][j] = dp[i+1][j-1] + 2;
                }else{
                    dp[i][j] = Math.max(dp[i+1][j],dp[i][j-1]);
                }
            }
        }
        // 最后是右上角那个点
        return dp[0][n-1];
    }
}
```

d.构建回文的最小插入次数

给你一个字符串 s ， 每一次操作你都可以在字符串的任意位置插入任意字符。

请你返回让 s 成为回文串的 最少操作次数 。

「回文串」是正读和反读都相同的字符串。

示例 1：

输入：s = "zzazz"
输出：0
解释：字符串 "zzazz" 已经是回文串了，所以不需要做任何插入操作。

```
class Solution {
    public int minInsertions(String s) {
        // 类似于查找最长回文序列
        char[] arr = s.toCharArray();
        // 动态规划
```

```
        int n = arr.length;
        int[][] dp = new int[n][n];
        // 正方形右上角
        // 初始化
        for(int i=0;i<n;i++){
            dp[i][i] = 0;
        }
        // 转移方程 从下往上 从左到右
        for(int i=n-2;i>=0;i--){
            for(int j=i+1;j<n;j++){
                // 相等无需处理
                if(arr[i]==arr[j]){
                    // 从当前往外走
                    dp[i][j] = dp[i+1][j-1];
                }else{
                    // 不相等 添加哪一个进来
                    dp[i][j] = Math.min(dp[i+1][j],dp[i][j-1]) + 1;
                }
            }
        }
        return dp[0][n-1];
    }
}
```

背包问题

子集背包问题

a.Leetcode416 分割等和子集

给定一个**只包含正整数**的**非空**数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

```
class Solution {
    public boolean canPartition(int[] nums) {
        // 从背包中挑选一部分数据，使其等于总和的一半
        int sum = 0;
        for(int num:nums){
            sum += num;
        }
        if(sum%2!=0){
            return false;
        }
        int target = sum/2;
        int n = nums.length;
        // 动态数组，判断是否能装满 用前i个数，组成target
        boolean[][] dp = new boolean[n+1][target+1];
        // 初始化
        dp[0][0] = true;
        // 转移方程
        for(int i=1;i<=n;i++){
```

```
            for(int j=1;j<=target;j++){
                //判断
                if(j>=nums[i-1]){
                    // 可以
                    dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i-1]];
                }else{
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[n][target];
    }
}
```

0-1背包问题

a.背包装载最大重量

给你一个可装载重量为`W`的背包和`N`个物体，每个物品有重量和价值两个属性。其中第`i`个物体的重量为`wt[i]`,价值为`val[i]`,现在让你用这个背包装东西，最多能装多少价值的东西?

```
public static int process(int[] wt,int[] vt,int w) {
        // 动态规划
        int n =wt.length;
        // dp表示前n个 装多少重量
        int[][] dp = new int[n+1][w+1];
        // 求最值max

        for(int i=1;i<=n;i++) {
            for(int j=1;j<=w;j++) {
                //查看是否能放
                if(j>=wt[i-1]) {
                    dp[i][j] = Math.max(dp[i-1][j], dp[i-1][j-wt[i-1]]+vt[i-
13]);
                }else {
                    // 延续之前的
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[n][w];

    }
```

b.背包装载是否能恰好装满

给n个物品，限重m，每个物品wi，从这几个物品中是否能选出几个恰好能装满?

```
public static boolean check(int[] nums,int target) {
        int n = nums.length;
        boolean[][] dp = new boolean[n+1][target+1];
        dp[0][0] = true;

        for(int i=1;i<=n;i++) {
```

```
            for(int j=1;j<=target;j++) {
                // 判断能不能放
                if(j>=nums[i-1]) {
                    //可以放 也可以不放
                    dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i-1]];
                }else {
                    // 不能放
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        // 默认
        return dp[n][target];
    }
```

c.Leetcode494目标和

给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 -中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

```
class Solution {
    public int findTargetSumWays(int[] nums, int S) {
        // 目标和的问题
        // 先统计nums的和
        int sum = 0;
        for(int num:nums){
            sum += num;
        }
        if((sum+S)%2!=0 || S>sum){
            return 0;
        }
        int target = (sum+S)/2;
        // 从前i个选出target的值的总和问题
        int n = nums.length;
        int[][] dp = new int[n+1][target+1];
        // 初始化
        for(int i=0;i<=n;i++){
            dp[i][0] = 1;
        }
        // 如果target变为0则计数一次
        for(int i=1;i<=n;i++){
            for(int j=0;j<=target;j++){
                // 转移方程
                if(j>=nums[i-1]){
                    dp[i][j] = dp[i-1][j] + dp[i-1][j-nums[i-1]];
                }else{
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[n][target];
    }
}
```

完全背包问题

东西是否可以重复放置。

a.Leetcode322 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。

你可以认为每种硬币的数量是无限的。

> 求最小的长度

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        int n = coins.length;
        // 动态规划数组dp的含义利用前i个硬币 兑换处amount的最小的数量
        int[][] dp = new int[n+1][amount+1];
        // 初始化 最大最小的初始化
        for(int i=0;i<=n;i++){
            // 使其放置为最大值
            Arrays.fill(dp[i],amount+1);
            if(i>=0){
                // 如果amount则没有
                dp[i][0] = 0;
            }
        }

        // 转移方程
        for(int i=1;i<=n;i++){
            for(int j=0;j<=amount;j++){
                if(j>=coins[i-1]){
                    // 可以兑换
                    dp[i][j] = Math.min(dp[i-1][j],dp[i][j-coins[i-1]]+1);
                }else{
                    dp[i][j] = dp[i-1][j];
                }
            }
        }

        return dp[n][amount]==amount+1?-1:dp[n][amount];


    }
}
```

b.Leetcode518零钱兑换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。

```
class Solution {
    public int change(int amount, int[] coins) {
        // 完全背包恰好问题
        // dp的含义用前i个，弄出amount的组合数量
```

```
        int n = coins.length;
        int[][] dp = new int[n+1][amount+1];
        // 初始化
        for(int i=0;i<=n;i++){
            // 将amount变为0的时候 可为一次
            dp[i][0] = 1;
        }
        // 挑选
        for(int i=1;i<=n;i++){
            for(int j=0;j<=amount;j++){
                // 判断
                if(j>=coins[i-1]){
                    // 可以兑换 求数量
                    dp[i][j] = dp[i-1][j] + dp[i][j-coins[i-1]];
                }else{
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        // 返回
        return dp[n][amount];
    }
}
```

贪心问题

区间调度问题

区间调度问题，给你很多形如`[start,end]`的区间。请设计一个算法，算出**这些区间中最多能有几个互不相交的区间。**

现实生活的应用：比如你今天有好几个活动，每个活动都可以用区间`[start,end]`表示这个开始和结束的时间，请问你今天**今儿最多能参加几个活动呢?** 显然你一个人不能同时参加两个活动，所以说这个问题就是求这些时间区间的最大不相交子集。

解题思路:

- 从区间集合intvs中选择一个区间x，这个x是在当前所有区间中结束最早的(end最小);
- 把所有与x区间相交的区间从区间集合intvs中删除;
- 重复步骤1和2，直到intvs为空位置。

算法实现的话，可以按每个区间的`end`数值升序排序，这样就好很多了。

a.Leetcode435 无重叠区间

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意:

可以认为区间的终点总是大于它的起点。
区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。
示例 1:

输入: [[1,2], [2,3], [3,4], [1,3]]

输出: 1

解释: 移除 [1,3] 后，剩下的区间没有重叠

> 无重叠的空间，使其最大的重叠，然后总长度减去即可了。

```
class Solution {
    public int eraseOverlapIntervals(int[][] intervals) {
        if(intervals.length==0){
            return 0;
        }
        // 找到最长的，去除即可了
        // 按照结束时间来排序
        Arrays.sort(intervals,(a,b)->(a[1]-b[1]));
        // 结果
        int res = 1;
        // 记录结束时间
        int end = intervals[0][1];
        // 对剩余的遍历
        for(int[] other:intervals){
            // 记录当前的开始时间
            int start = other[0];
            // 如果当前开始时间>=结束时间
            if(start>=end){
                res++;
                // 更新
                end = other[1];
            }
        }
        int len = intervals.length;
        return len-res;
    }
}
```

b.Leetcode452用最少数量的箭来引爆气球

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以纵坐标并不重要，因此只要知道开始和结束的横坐标就足够了。开始坐标总是小于结束坐标。

一支弓箭可以沿着 x 轴从不同点完全垂直地射出。在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

给你一个数组 points ， 其中 points [i] = [xstart,xend] ， 返回引爆所有气球所必须射出的最小弓箭数。

示例 1：

输入：points = [[10,16],[2,8],[1,6],[7,12]]
输出：2
解释：对于该样例，x = 6 可以射爆 [2,8],[1,6] 两个气球，以及 x = 11 射爆另外两个气球

> 求区间，注意比较用Integer.compare(), 否则出现溢出

```
class Solution {
    public int findMinArrowShots(int[][] points) {
        if(points.length==0){
            return 0;
        }
```

```
        // 计算最大的没有重叠区间的，总长度-没有重叠的，即可了
        // 比较因为溢出过大而无法通过
        //Arrays.sort(points,(a,b)->(a[1]-b[1]));
        Arrays.sort(points,(a,b)->(Integer.compare(a[1],b[1])));
        // 结果
        int res = 1;
        // 结束
        int end_p = points[0][1];
        // 遍历其它的
        for(int[] point:points){
            int  start_p = point[0];
            if(start_p>end_p){
                ++res;
                // 更新
                end_p = point[1];
            }
        }
        return res;
    }
}
```

跳跃问题

a.跳跃游戏

给定一个非负整数数组 `nums` ，你最初位于数组的 **第一个下标** 。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标。

示例 1：

输入：nums = [2,3,1,1,4]
输出：true
解释：可以先跳 1 步，从下标 0 到达下标 1, 然后再从下标 1 跳 3 步到达最后一个下标。

> 注意：这里是跳跃到最后一个那里，维护一个最长的长度即可了

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

b.Leetcode045 跳跃游戏II

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

> 找到最小跳跃到最后的一个位置的次数
>
> 维护最长长度，维护最久能达到的end

```
class Solution {
    public int jump(int[] nums) {
        // 肯定能跳过去
        int end = 0;
        int long_path = 0;
        // 跳跃次数
        int count = 0;
        int len = nums.length-1;
        for(int i=0;i<len;i++){
            // 维护最长长度
            long_path = Math.max(long_path,i+nums[i]);
            if(end==i){
                count++;
                end = long_path;
            }
        }
        return count;
    }
}
```

股票问题

a.Leetcode121买卖股票的最佳时机

给定一个数组 prices ，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。

你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0 。

> k=1次买卖
>
> 维护一个当前i之前的最小的最低价，并且动态维护利润

```
class Solution {
    public int maxProfit(int[] prices) {
        // 买卖股票的时机
        // 结果
        int max_profit = 0;
        // 维护一个当前i之前的最小值，
        int min_prices = prices[0];
        for(int i=1;i<prices.length;i++){
            max_profit = Math.max(prices[i]-min_prices,max_profit);
            // 更新
            min_prices = Math.min(min_prices,prices[i]);
        }
        return max_profit;
    }
}
```

b.Leetcode122买卖股票的最佳时机II

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：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）。

> 可进行无限次的交易；
>
> 股票的状态有两个

```
class Solution {
    public int maxProfit(int[] prices) {
        // 当前股票的状态
        // 未持有
        int dp0 = 0;
        // 持有
        int dp1 = -prices[0];
        //  其余天数的遍历
        int n = prices.length;
        for(int i=1;i<n;i++){
            // 未持有的更新=之前就是未持有 之前持有今儿卖了
            int newdp0 = Math.max(dp0,dp1+prices[i]);
            // 持有的更新
            int newdp1 = Math.max(dp1,dp0-prices[i]);

            dp0 = newdp0;
            dp1 = newdp1;
        }
        // 未持有的
        return dp0;
    }
}
```

c.买卖股票的最佳时机IIII

给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 两笔 交易。

注意：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）。

> K=2 5个状态

```
class Solution {
    public int maxProfit(int[] prices) {
        // k只能进行两次交易 5个状态
        // 一次也未交易
        int dp0 = 0;
        // 第一次买入了
        int dp1 = -prices[0];
        // 第一次卖出了
        int dp2 = 0;
```

```
        // 第二次买入了
        int dp3 = Integer.MIN_VALUE;
        // 第三次卖出了
        int dp4 = Integer.MIN_VALUE;
        // 对其遍历
        int n = prices.length;
        for(int i=1;i<n;i++){
            int newdp0 = dp0;
            int newdp1 = Math.max(dp1,dp0-prices[i]);
            int newdp2 = Math.max(dp2,dp1+prices[i]);
            int newdp3 = Math.max(dp3,newdp2-prices[i]);
            int newdp4 = Math.max(dp4,newdp3+prices[i]);

            dp0 = newdp0;
            dp1 = newdp1;
            dp2 = newdp2;
            dp3 = newdp3;
            dp4 = newdp4;
        }
        return Math.max(dp2,dp4);

    }
}
```

d.买卖股票的最佳时机IV

给定一个整数数组 prices ， 它的第 i 个元素 prices[i] 是一支给定的股票在第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 k 笔交易。

注意：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

> K次交易的重点在于，两个for循环，但是对交易判断

```
class Solution {
    public int maxProfit(int K, int[] prices) {
        int n = prices.length;
        if(n<1){
            return 0;
        }
          // 判断k
        K = Math.min(K,n/2);
        // 动态规划 // 第几天 持有未持有  交易次数
        int[][][] dp = new int[n][2][K+1];
        // 第一题初始化
        for(int k=0;k<=K;k++){
            // 未持有
            dp[0][0][k] = 0;
            // 持有了
            dp[0][1][k] = -prices[0];
        }

        // 转移方程
        for(int i=1;i<n;i++){
            for(int k=1;k<=K;k++){
                // 未持有 什么时候交易次数 买了才加
```

```
                dp[i][0][k] = Math.max(dp[i-1][0][k],dp[i-1][1][k]+prices[i]);
                dp[i][1][k] = Math.max(dp[i-1][1][k],dp[i-1][0][k-1]-prices[i]);
            }
        }
        return dp[n-1][0][K];
    }
}
```

e.买卖股票时机含冷冻期

> k次，冷冻期。3种状态，买了，卖了冷冻期，卖了不处于冷冻期

```
class Solution {
    public int maxProfit(int[] prices) {
        // k次有冷冻期
        // 持有
        int dp0 = -prices[0];
        // 未持有处于冷冻期
        int dp1 = 0;
        // 未持有不处于冷冻期了
        int dp2 = 0;
        for(int i=1;i<prices.length;i++){
            int newdp0 = Math.max(dp0,dp2-prices[i]);
            int newdp1 = Math.max(dp1,dp0+prices[i]);
            int newdp2 = Math.max(dp2,dp1);

            dp0 = newdp0;
            dp1 = newdp1;
            dp2 = newdp2;
        }

        return Math.max(dp1,dp2);
    }
}
```

f.买卖股票的最佳时机含手续费

k次交易，含有手续费

```
class Solution {
    public int maxProfit(int[] prices, int fee) {
        // k次交易
        // 未持有
        int dp0 = 0;
        // 持有了
        int dp1 = -prices[0];
        for(int i=1;i<prices.length;i++){
            int newdp0 = Math.max(dp0,dp1+prices[i]-fee);
            int newdp1 = Math.max(dp1,dp0-prices[i]);
            dp0 = newdp0;
            dp1 = newdp1;
        }
        return dp0;
```

```
    }
}
```

打家劫舍问题

第一道是比较标准的动态规划问题，而第二道融入了环形数组的条件，第三道更绝，把动态规划的自底向上和自顶向下解法和二叉树结合起来

a.Leetcode198 打家劫舍

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。

> 解题思路：在初始化的过程中不同于爬楼梯，需要先确定最大值

```java
class Solution {
    public int rob(int[] nums) {
        int n = nums.length;
        if(n<=1){
            return nums[0];
        }
        // 动态规划 以i结尾的金额
        int[] dp = new int[n];
        dp[0] = nums[0];
        // 打劫的话
        dp[1] = Math.max(nums[0],nums[1]);
        for(int i=2;i<n;i++){
            // 偷本次的 或者不投这家
            dp[i] = Math.max(dp[i-2]+nums[i],dp[i-1]);
        }
        // 最后返回
        return dp[n-1];
    }
}
```

> 倒着抢劫

```java
class Solution {
    public int rob(int[] nums) {
        // 倒着开始抢
        int dp_i_1 = 0;
        int dp_i_2 = 0;
        // 当前天
        int dp_i = 0;
        int n= nums.length-1;
        for(int i=n;i>=0;i--){
            // 转移方程
            dp_i = Math.max(dp_i_1,dp_i_2+nums[i]);
            dp_i_2 = dp_i_1;
            dp_i_1 = dp_i;
        }
        return dp_i;
```

```
    }
}
```

b.Leetcode213 打家劫舍II

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 。

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

> 解题思路：
>
> 要不从最后一个，不到第一个；
>
> 要不从不从最后一个，到第一个

```java
class Solution {
    public int rob(int[] nums) {
        // 打家劫舍环形链表 最优结果就是最后一个抢还是不抢
        int n = nums.length;
        if(n<=1){
            return nums[0];
        }
        // 最后选择那么第一个不选择 最后一个不选择那么第一个选择
        return Math.max(process(nums,n-1,1),process(nums,n-2,0));
    }
    public int process(int[] nums,int start,int end){
        int dp_i_1 = 0;
        int dp_i_2 = 0;
        // 当前天数
        int dp_i = 0;
        for(int i=start;i>=end;i--){
            dp_i = Math.max(dp_i_1,dp_i_2+nums[i]);
            dp_i_2 = dp_i_1;
            dp_i_1 = dp_i;
        }
        return dp_i;

    }
}
```

c.Leetcode337 打家劫舍III

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 1:

输入: [3,2,3,null,3,null,1]

```
 3
/ \
```

```
2  3
\  \
 3  1
```

输出: 7

> 递归+动态

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    // 二叉树 抢劫头结点或者不抢劫头结点
    public int rob(TreeNode root) {
        int[] res = dps(root);
        return Math.max(res[0],res[1]);
    }

    public int[] dps(TreeNode root){
        if(root==null){
            return new int[]{0,0};
        }
        // 左边抢劫
        int[] left = dps(root.left);
        // 右边抢劫
        int[] right = dps(root.right);

        // 抢劫根节点两边不能抢
        int rob = root.val + left[1] + right[1];
        // 不抢劫根节点 两边可以抢也可以不抢
        int rob_no = Math.max(left[0],left[1]) + Math.max(right[0],right[1]);

        return new int[]{rob,rob_no};
    }
}
```

其他算法问题

a.经典动态规划：正则表达式

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。

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。

```
class Solution {

    public boolean isMatch(String s, String p) {
        int slen=s.length();
        int plen=p.length();
        if(slen==0&&plen==0)return true;
        //if(slen==0||plen==0)return false;

        boolean[][] dp=new boolean[slen+1][plen+1];
        //dp[i][j]表示s的0到i-1和p的0到j-1是否匹配
        dp[0][0]=true;
        //初始化s=0
        for(int j=1;j<=plen;j++){
            //当s为空时，a*b*c*可以匹配
            //当判断到下标j-1是*，j-2是b，b对应f，要看之前的能否匹配
            //比如a*b*下标依次为ftft，b之前的位t，所以j-1也是true
            //即dp[0][j]对应的下标j-1位true
            if(j==1)dp[0][j]=false;
            if(p.charAt(j-1)=='*'&&dp[0][j-2])dp[0][j]=true;
        }

        //for循环当s长度为1时能否匹配，一直到s长度为slen
        for(int i=1;i<=slen;i++){
            for(int j=1;j<=plen;j++){
                //最简单的两种情况    字符相等或者p的字符是‘.'
                if(s.charAt(i-1)==p.charAt(j-1)||p.charAt(j-1)=='.'){
                    dp[i][j]=dp[i-1][j-1];
                }
                //p当前字符是*时，要判断*前边一个字符和s当前字符

                else if(p.charAt(j-1)=='*'){
                    if(j<2)dp[i][j]=false;
                     //如果p的*前边字符和s当前字符相等或者p的字符是‘.'
                     //三种可能
                     //匹配0个，比如aa aaa*也就是没有*和*之前的字符也可以匹配上（在你（a*）没来之前我们(aa)已经能匹配上了）dp[i][j]=dp[i][j-2]
                     //匹配1个，比如aab aab* 也就是*和*之前一个字符只匹配s串的当前一个字符就不看*号了  即 dp[i][j]=dp[i][j-1]
                     //匹配多个，比如aabb aab*  b*匹配了bb两个b  那么看aab 和aab*是否能匹配上就行了，即dp[i][j]=dp[i-1][j]
                     if(p.charAt(j-2)==s.charAt(i-1)||p.charAt(j-2)=='.'){
                        dp[i][j]=dp[i-1][j]||dp[i][j-1]||dp[i][j-2];
                    }
```

```
                    //如果p的*前边字符和s当前字符不相等或者p的字符不是‘.’，那就把*和*前边一个字符都不要了呗
                    //你会发现不管是这种情况还是上边的情况都会有dp[i][j]=dp[i][j-2];所以可以把下边剪枝，不用分开写了
                    //这里分开写是为了好理解
                    else if(p.charAt(j-2)!=s.charAt(i-1)&&p.charAt(j-2)!='.'){
                        dp[i][j]=dp[i][j-2];
                    }
                }
                //其他情况肯定不能匹配上了  直接false  比如 aba abb*c
                else{
                    dp[i][j]=false;
                }
            }
        }
        return dp[slen][plen];

    }
}
```

b.经典动态规划：高楼扔鸡蛋

给你 k 枚相同的鸡蛋，并可以使用一栋从第 1 层到第 n 层共有 n 层楼的建筑。

已知存在楼层 f ，满足 0 <= f <= n ，任何从 高于 f 的楼层落下的鸡蛋都会碎，从 f 楼层或比它低的楼层落下的鸡蛋都不会破。

每次操作，你可以取一枚没有碎的鸡蛋并把它从任一楼层 x 扔下（满足 1 <= x <= n）。如果鸡蛋碎了，你就不能再次使用它。如果某枚鸡蛋扔下后没有摔碎，则可以在之后的操作中 重复使用 这枚鸡蛋。

请你计算并返回要确定 f 确切的值 的 最小操作次数 是多少?

示例 1：

输入：k = 1, n = 2
输出：2
解释：
鸡蛋从 1 楼掉落。如果它碎了，肯定能得出 f = 0 。
否则，鸡蛋从 2 楼掉落。如果它碎了，肯定能得出 f = 1 。
如果它没碎，那么肯定能得出 f = 2 。
因此，在最坏的情况下我们需要移动 2 次以确定 f 是多少。

> 超时：递归+动态规划

```
class Solution {
    // K个鸡蛋，N层楼
    private Integer[][] res;
    public int superEggDrop(int K, int N) {
        // 递归截止条件
        if(K==1){
            return N;
        }
        if(N==1){
            return 1;
        }
```

```
        if(res==null){
            res = new Integer[K+1][N+1];
        }
        if(res[K][N]!=null){
            return res[K][N];
        }

        // 状态选择
        // 结果
        int res_temp = N;
        for(int i=1;i<=N;i++){
            res_temp = Math.min(res_temp,Math.max(superEggDrop(K-1,i-
1),superEggDrop(K,N-i))+1);
        }
        return res[K][N]=res_temp;


    }
}
```

c.经典动态规划：高楼扔鸡蛋（进阶）（二分算法）

在上述的基础上用二分法来改进，就是全部for循环改成while循环

```
class Solution {
    // K个鸡蛋，N层楼
    private Integer[][] res;
    public int superEggDrop(int K, int N) {
        // 递归截止条件
        if(K==1){
            return N;
        }
        if(N==1){
            return 1;
        }

        if(res==null){
            res = new Integer[K+1][N+1];
        }
        if(res[K][N]!=null){
            return res[K][N];
        }

        // 状态选择
        // 结果
        int res_temp = N;

        // 不对所有的N的都遍历
        int lo = 1;
        int hi = N+1;
        while(lo<hi){
```

```
            int m = lo + (hi-lo)/2;
            // 左边
            int left = superEggDrop(K-1,m-1);
            int right = superEggDrop(K,N-m);
            res_temp = Math.min(res_temp,Math.max(left,right)+1);
            // 移动
            if(left<right){
                lo = m +1;
            }else if(left>right){
                hi = m;
            }else{
                break;
            }
        }

        return res[K][N]=res_temp;


    }
}
```

d.经典动态规划：戳气球（区间DP）

最值问题，就是可以for循环，遍历得到所有结果；

而遍历得到所有结果，首先可以for循环来解决，要不就是回溯法；当然优化就是动态规划

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。戳破第 i 个气球，你可以获得 nums[i - 1] * nums[i] * nums[i + 1] 枚硬币。 这里的 i - 1 和 i + 1 代表和 i 相邻的两个气球的序号。如果 i - 1或 i + 1 超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

示例 1：
输入：nums = [3,1,5,8]
输出：167
解释：
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins = *3*15 + *3*5*8 + *1*3*8 + *1*8*1 = 167

> 解题思路：正方形dp，从下往上走 从左往右
>
> 区间问题，多了个中间的k

```
class Solution {
    public int maxCoins(int[] nums) {
        // 给其两端增加
        int n = nums.length;
        int[] new_num = new int[n+2];
        new_num[0] = 1;
        new_num[n+1] = 1;
        for(int i=1;i<=n;i++){
            new_num[i] = nums[i-1];
```

```
        }
        // 开始动态规划 // 记录两个区间之间的值
        int[][] dp = new int[n+2][n+2];
        // 初始化对角线都为0
        // 转移方程两个区间
        for(int i=n;i>=0;i--){
            for(int j=i+1;j<n+2;j++){
                // 在两个区间中
                for(int k=i+1;k<j;k++){
                    // 别的两个区间
                    dp[i][j] = Math.max(dp[i][j],dp[i][k]+dp[k]
[j]+new_num[i]*new_num[j]*new_num[k]);
                }
            }
        }
        return dp[0][n+1];

    }
}
```

奇怪的打印机

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'。

> 区间dp

```
class Solution {
    public int strangePrinter(String s) {
        // 奇怪的打印机
        char[] arr = s.toCharArray();
        int n = arr.length;
        if(n==0){
            return 0;
        }
        // 动态规划，从i到j区间内
        int[][] dp = new int[n][n];
        // 初始化
        for(int i=0;i<n;i++){
            dp[i][i] = 1;
        }
        // 开始
```

```
        for(int i=n-2;i>=0;i--){
            for(int j=i+1;j<n;j++){
                dp[i][j] = j-i+1;// 最坏情况，所有字符都不一样
                // 分割
                for(int k=i;k<j;k++){
                    int total = dp[i][k] + dp[k+1][j];
                    if(arr[k]==arr[j]){
                        total--;
                    }
                    dp[i][j] = Math.min(dp[i][j],total);
                }
            }
        }
        return dp[0][n-1];
    }
}
```

移除盒子

给出一些不同颜色的盒子，盒子的颜色由数字表示，即不同的数字表示不同的颜色。

你将经过若干轮操作去去掉盒子，直到所有的盒子都去掉为止。每一轮你可以移除具有相同颜色的连续 k 个盒子（k >= 1），这样一轮之后你将得到 k * k 个积分。

当你将所有盒子都去掉之后，求你能获得的最大积分和。

示例 1：

输入：boxes = [1,3,2,2,2,3,4,3,1]
输出：23
解释：
[1, 3, 2, 2, 2, 3, 4, 3, 1]
----> [1, 3, 3, 4, 3, 1] (3*3=9 分)
----> *[1, 3, 3, 3, 1]* (*1*1=1 分)
----> [1, 1] (3*3=9 分)
----> *[]* (*2*2=4 分)

> 解题思路：区间问题

```
class Solution {
    public int removeBoxes(int[] boxes) {
        // 用于存储之前计算过的状态，避免重复计算
        int[][][] dp = new int[100][100][100];
        return cal(boxes, dp, 0, boxes.length - 1, 0);
    }

    public int cal(int[] boxes, int[][][] dp, int l, int r, int k) {
        if (l > r) {
            return 0;
        }
        if (dp[l][r][k] != 0) {
            return dp[l][r][k];
        }
        // 计算右边有几个跟最右边一个（boxes[r]）相等，如果相等则把右边界左移到不相同的元素之后一个为止，移动过程中同步改动k
        while (r > l && boxes[r] == boxes[r-1]) {
```

```
            r--;
            k++;
        }
        // 计算把右边k+1个消除时的得分
        dp[l][r][k] = cal(boxes, dp, l, r-1, 0) + (k+1)*(k+1);
        // 从右边界开始向左寻找跟外部k个元素相等的元素，如果相等则剔除掉这些不相等的，让后面一段连起来。
        // 此时得分就是中间消除中间一段不连续部分的得分和剩下来部分的得分
        // 比较这个得分和原来计算过其他方案的得分，去最大值覆盖到状态数组dp中
        for (int i = r-1; i >= l; --i) {
            if (boxes[i] == boxes[r]) {
                dp[l][r][k] = Math.max(dp[l][r][k], cal(boxes, dp, l, i, k+1) + cal(boxes, dp, i+1, r-1, 0));
            }
        }
        return dp[l][r][k];

    }
}
```

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 。

```
class Solution {
    public boolean PredictTheWinner(int[] nums) {
        // 预测赢家
        int n = nums.length;
        int[][] dp = new int[n][n];
        // 初始化
        for(int i=0;i<n;i++){
            // 对角线赋值
            dp[i][i] = nums[i];
        }
```

```
        // 转移方程从下到上
        for(int i=n-2;i>=0;i--){
            for(int j=i+1;j<n;j++){
                dp[i][j] = Math.max(nums[i]-dp[i+1][j],nums[j]-dp[i][j-1]);
            }
        }
        //
        return dp[0][n-1]>=0;
    }
}
```

e.经典动态规划：博弈问题 (小球问题)

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 。

示例：

输入：[5,3,4,5]
输出：true
解释：
亚历克斯先开始，只能拿前 5 颗或后 5 颗石子 。
假设他取了前 5 颗，这一行就变成了 [3,4,5] 。
如果李拿走前 3 颗，那么剩下的是 [4,5]，亚历克斯拿走后 5 颗赢得 10 分。
如果李拿走后 5 颗，那么剩下的是 [3,4]，亚历克斯拿走后 4 颗赢得 9 分。
这表明，取前 5 颗石子对亚历克斯来说是一个胜利的举动，所以我们返回 true

> 泛化
>
> 你和你的朋友面前有一排石头堆，用一个数组 piles 表示，piles[i] 表示第 i 堆石子有多少个。你们轮流拿石头，一次拿一堆，但是只能拿走最左边或者最右边的石头堆。所有石头被拿完后，谁拥有的石头多，谁获胜。
>
> 石头的堆数可以是任意正整数，石头的总数也可以是任意正整数，这样就能打破先手必胜的局面了。比如有三堆石头 `piles = [1, 100, 3]`，先手不管拿 1 还是 3，能够决定胜负的 100 都会被后手拿走，后手会获胜。

一种博弈问题的通用设计框架。

以下是对 dp 数组含义的解释：

`dp[i][j].fir` 表示，对于 `piles[i...j]` 这部分石头堆，先手能获得的最高分数。`dp[i][j].sec` 表示，对于 `piles[i...j]` 这部分石头堆，后手能获得的最高分数。
举例理解一下，假设 `piles = [3, 9, 1, 2]`，索引从 0 开始`dp[0][1].fir = 9` 意味着：面对石头堆 `[3, 9]`，先手最终能够获得 9 分。`dp[1][3].sec = 2` 意味着：面对石头堆 `[9, 1, 2]`，后手最终能够获得 2 分。

我们想求的答案是先手和后手最终分数之差，按照这个定义也就是 `dp[0][n-1].fir - dp[0][n-1].sec`，即面对整个 piles，先手的最优得分和后手的最优得分之差。

写状态转移方程很简单，首先要找到所有「状态」和每个状态可以做的「选择」，然后择优。

根据前面对 dp 数组的定义，**状态显然有三个：开始的索引 i，结束的索引 j，当前轮到的人。**

```
dp[i][j][fir or sec]
其中：
0 <= i < piles.length
i <= j < piles.length
```

对于这个问题的每个状态，可以做的**选择有两个：选择最左边的那堆石头，或者选择最右边的那堆石头。** 我们可以这样穷举所有状态：

```
n = piles.length
for 0 <= i < n:
    for j <= i < n:
        for who in {fir, sec}:
            dp[i][j][who] = max(left, right)
```

根据我们对 dp 数组的定义，很容易解决这个难点，**写出状态转移方程：**

```
dp[i][j].fir = max(piles[i] + dp[i+1][j].sec, piles[j] + dp[i][j-1].sec)dp[i][j].fir = max(    选择最左边的石头堆      ,      选择最右边的石头堆      )# 解释：我作为先手，面对 piles[i...j] 时，有两种选择：# 要么我选择最左边的那一堆石头，然后面对 piles[i+1...j]# 但是此时轮到对方，相当于我变成了后手；# 要么我选择最右边的那一堆石头，然后面对 piles[i...j-1]# 但是此时轮到对方，相当于我变成了后手。
if 先手选择左边:    dp[i][j].sec = dp[i+1][j].firif 先手选择右边:    dp[i][j].sec = dp[i][j-1].fir# 解释：我作为后手，要等先手先选择，有两种情况：# 如果先手选择了最左边那堆，给我剩下了 piles[i+1...j]# 此时轮到我，我变成了先手；# 如果先手选择了最右边那堆，给我剩下了 piles[i...j-1]# 此时轮到我，我变成了先手。
```

根据 dp 数组的定义，我们也可以找出 **base case**，也就是最简单的情况：

```
dp[i][j].fir = piles[i]
dp[i][j].sec = 0
其中 0 <= i == j < n
# 解释：i 和 j 相等就是说面前只有一堆石头 piles[i]
# 那么显然先手的得分为 piles[i]
# 后手没有石头拿了，得分为 0
```

```
/* 返回游戏最后先手和后手的得分之差 */
int stoneGame(int[] piles) {
    int n = piles.length;
    // 初始化 dp 数组
    Pair[][] dp = new Pair[n][n];
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++)
            dp[i][j] = new Pair(0, 0);
    // 填入 base case
```

```
    for (int i = 0; i < n; i++) {
        dp[i][i].fir = piles[i];
        dp[i][i].sec = 0;
    }
    // 斜着遍历数组
    for (int l = 2; l <= n; l++) {
        for (int i = 0; i <= n - l; i++) {
            int j = l + i - 1;
            // 先手选择最左边或最右边的分数
            int left = piles[i] + dp[i+1][j].sec;
            int right = piles[j] + dp[i][j-1].sec;
            // 套用状态转移方程
            if (left > right) {
                dp[i][j].fir = left;
                dp[i][j].sec = dp[i+1][j].fir;
            } else {
                dp[i][j].fir = right;
                dp[i][j].sec = dp[i][j-1].fir;
            }
        }
    }
    Pair res = dp[0][n-1];
    return res.fir - res.sec;
}
```

f.经典动态规划：四键键盘

假设你有一个特殊的键盘包含下面的按键：

`Key 1: (A)`：在屏幕上打印一个 'A'。

`Key 2: (Ctrl-A)`：选中整个屏幕。

`Key 3: (Ctrl-C)`：复制选中区域到缓冲区。

`Key 4: (Ctrl-V)`：将缓冲区内容输出到上次输入的结束位置，并显示在屏幕上。

现在，你只可以按键 **N** 次（使用上述四种按键），请问屏幕上最多可以显示几个 'A'呢？

样例 1:

```
输入: N = 3
输出: 3
解释:
我们最多可以在屏幕上显示三个'A'通过如下顺序按键：
A, A, A
```

样例 2:

```
输入: N = 7
输出: 9
解释:
我们最多可以在屏幕上显示九个'A'通过如下顺序按键：
A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V
```

第一个状态是剩余的按键次数，用 `n` 表示；第二个状态是当前屏幕上字符 A 的数量，用 `a_num` 表示；第三个状态是剪切板中字符 A 的数量，用 `copy` 表示。

```
def maxA(N: int) -> int:

    # 对于 (n, a_num, copy) 这个状态，
    # 屏幕上能最终最多能有 dp(n, a_num, copy) 个 A
    def dp(n, a_num, copy):
        # base case
        if n <= 0: return a_num;
        # 几种选择全试一遍，选择最大的结果
        return max(
                dp(n - 1, a_num + 1, copy),     # A
                dp(n - 1, a_num + copy, copy), # C-V
                dp(n - 2, a_num, a_num)         # C-A C-C
            )

    # 可以按 N 次按键，屏幕和剪切板里都还没有 A
    return dp(N, 0, 0)
```

g.有限状态机之KMP字符匹配算法

KMP 算法（Knuth-Morris-Pratt 算法）是一个著名的字符串匹配算法，效率很高，但是确实有点复杂。

很多读者抱怨 KMP 算法无法理解，这很正常，想到大学教材上关于 KMP 算法的讲解，也不知道有多少未来的 Knuth、Morris、Pratt 被提前劝退了。有一些优秀的同学通过手推 KMP 算法的过程来辅助理解该算法，这是一种办法，不过本文要从逻辑层面帮助读者理解算法的原理。十行代码之间，KMP 灰飞烟灭。

先在开头约定，本文用 `pat` 表示模式串，长度为 `M`，`txt`** 表示文本串，长度为 `N`**。KMP 算法是在** `txt` **中查找子串** `pat`**，如果存在，返回这个子串的起始索引，否则返回 -1**。

Leetcode28 实现strStr()

实现 strStr() 函数。

给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回 -1。

示例 1:

输入: haystack = "hello", needle = "ll"
输出: 2

示例 2:

输入: haystack = "aaaaa", needle = "bba"
输出: -1

> 暴力破解

```
class Solution {
    public int strStr(String haystack, String needle) {
        // 暴力破解KMP算法
        char[] hay_arr = haystack.toCharArray();
        char[] need_arr = needle.toCharArray();
        // 第一个的值
        int len1 = hay_arr.length;
        int len2 = need_arr.length;
```

```
        // 遍历
        for(int i=0;i<=len1-len2;i++){
            int j;
            for(j=0;j<len2;j++){
                // 不相等则break
                if(hay_arr[i+j]!=need_arr[j]){
                    break;
                }
            }
            // 如果都匹配了
            if(j==len2){
                return i;
            }
        }

        return -1;
    }
}
```

> 解法：KMP算法

```
public class KMP {
    private int[][] dp;
    private String pat;

    public KMP(String pat) {
        this.pat = pat;
        int M = pat.length();
        // dp[状态][字符] = 下个状态
        dp = new int[M][256];
        // base case
        dp[0][pat.charAt(0)] = 1;
        // 影子状态 X 初始为 0
        int X = 0;
        // 构建状态转移图（稍改的更紧凑了）
        for (int j = 1; j < M; j++) {
            for (int c = 0; c < 256; c++)
                dp[j][c] = dp[X][c];
            dp[j][pat.charAt(j)] = j + 1;
            // 更新影子状态
            X = dp[X][pat.charAt(j)];
        }
    }

    public int search(String txt) {
        int M = pat.length();
        int N = txt.length();
        // pat 的初始态为 0
        int j = 0;
        for (int i = 0; i < N; i++) {
            // 计算 pat 的下一个状态
            j = dp[j][txt.charAt(i)];
            // 到达终止态，返回结果
            if (j == M) return i - M + 1;
        }
```

```
        // 没到达终止态，匹配失败
        return -1;
    }
}
```