上岸进行时！

1. **Dynamic programming**

- Leetcode 518 零钱兑换II
- 0Leetcode 493 目标和
- 优化：dp
- Leetcode 64 最小路径和
- Leetcode198 打家劫舍
- 优化版：不需要dp数组
- Leetcode213 打家劫舍II
- 213. 打家劫舍 II
- Leetcode 337 打家劫舍III
- 力扣46 全排列
    - 两数相加
- presum
- 区间加法

2. **回溯算法**

    1. 目标和Java/python)
    2. 全排列（Java/python）
    3. 力扣22 括号生成
    4.

3. **ListNode**

    1. [两数相加](# 两数相加)
    2. 合并两个有序链表
    3. Leetcode86 分割链表（Java/python）
    4. 力扣23 合并K个链表（Java/python）
    5. 剑指22 倒数第k个数（Java/python）
    6. 力扣19 删除倒数k个数（Java/python）
    7. 力扣876 链表的中点（Java/python）
    8. 力扣141 环形链表（Java/python）
    9. 剑指II 22 环形链表起点（Java）
    10. 力扣160 相交链表的起点（Java/python）
    11. 力扣83 删除重复的（Java）
    12. 力扣206 翻转链表（Java/python）

4. **other algorithms**

    1. [presum](# presum)
    2. 力扣384 打乱数组

5. **array**

1. 力扣26 删除数组中重复的（Java/python）
2. 力扣27 删除数组中的x（Java）
3. 力扣283 移动0（Java）

6. **sliding window**

1. 力扣167 两数之和II（Java）
2. 力扣344 反转字符串（Java/python）
3. 力扣5 最长回文子串（Java）
4. 力扣76 最小覆盖子串（cpp）
5. 力扣567 字符串排列
6. 力扣3 无重复最长子串
7. 力扣15 三数之和
8. 力扣209 长度最小子数组
9.

7. **stack**

1. 力扣20 有效的括号

8. **dfs**

1. 力扣200 岛屿数量
2. 力扣1254 封闭岛屿的数量
3. 力扣695 岛屿的最大面积
4. 力扣1905 统计子岛屿
5. 力扣694 不同岛屿数量

9. **bfs**

1. 力扣111 二叉树最小深度

10. **Binary tree**

1. 力扣100 相同的树
2. 力扣572 另一棵树的子数
3. 力扣102 二叉树层序遍历
4. 力扣103 二叉树的矩形层序遍历
5. 力扣1161 最大层内元素和
6. 力扣1302 层数最深的叶子结点之和
7. 力扣1609 奇偶树
8. 力扣637 二叉树层的平均值
9. 力扣958 二叉树完全性验证
10. 力扣104 二叉树最大深度
11. 力扣114 前序遍历
12. 力扣543 二叉树直径
13. 力扣559 N叉树最大深度
14. 力扣105 从前序和中序生成二叉树

15. 力扣106 中后序生成二叉树
16. 力扣654 最大二叉树
17. 力扣111 二叉树最小深度
18. 力扣114 将二叉树展开为链表
19. 力扣116 填充每一个节点的右侧节点
20. 力扣226 反转二叉树
21. 力扣117 为每一个节点填充nextII
22. 力扣145 后序遍历
23. 力扣222 完全二叉树节点个数
24. 力扣297 二叉树序列化和反序列化
25. 力扣124 二叉树中最大路径和
26. 力扣687 最长相同路径
27. 力扣814 二叉树剪枝
28. 力扣1325 删除给定值的叶子结点
29. 力扣589 N叉树前序遍历
30. 力扣652 寻找重复的子树
31. 力扣965 单值二叉树
32. 力扣255 验证前序遍历是否是BST
33. 力扣450 删除BST中节点
34. 力扣700 BST搜索
35. 力扣701 插入BST
36. 力扣98 验证BST
37. 力扣1038 从BST得到累加树
38. 力扣230 BST中第K小的元素
39. 剑指54 BST中第K大的元素
40. 力扣530 BST最小绝对差
41. 力扣270 最接近的BST的值
42. 力扣285 BST的中序后继
43.

Leetcode 931

```
class Solution {
    public int minFallingPathSum(int[][] matrix) {
            //dp[]: record the minFallingPathSum from last level of matrix
            int m = matrix.length, n = matrix.length;
            int [][] dp = new int[m][m];

            //fill the first level of dp[][]: the original value of matrix
            for(int i = 0; i < m; i ++){
                dp[0][i] = matrix[0][i];
            }
            //fill th rest level of dp[][], we have three cases
            for(int i = 1; i < m; i ++){
                for (int j = 0; j < n; j ++){
                    //first case, left column of matrix
                    if(j == 0){
```

```
                        dp[i][j] = Math.min((dp[i-1][j] + matrix[i][j]),
(dp[i-1][j+1])+matrix[i][j] )  ;
                    }
                    //second case: right column of matrix
                    else if(j == n-1){
                        dp[i][j] =Math.min((matrix[i][j]+dp[i-1][j-1]),
(matrix[i][j]+dp[i-1][j]));
                    }
                    //normal case: in the middle: min of three
                    else{
                        dp[i][j] = min_value((matrix[i][j]+dp[i-1][j-1]),
(matrix[i][j]+dp[i-1][j]),(matrix[i][j]+ dp[i-1][j+1]));
                    }
                }
            }

            //answer is the min of last level of dp
            int res = Integer.MAX_VALUE;
            for(int i = 0; i <m; i ++){
                res = Math.min(res,dp[n-1][i]);
            }
            return res;


    }

    int min_value(int a, int b, int c){
        int res = Math.min(a,b);
        return Math.min(res,c);
    }
}
```

更优的解法:

```
class Solution {
    public int minFallingPathSum(int[][] A) {
        int n = A.length;
        int m = A[0].length;
        int[] dp = new int[m];

        // 初始化 dp 数组第一行
        for(int j = 0; j < m; j++){
            dp[j] = A[0][j];
        }

        for(int i = 1; i < n; i++){
          //每次都有一个新的dp数组
            int[] newDp = new int[m];
            for(int j = 0; j < m; j++){
                if(j == 0){
                  //从dp[]里面拿数据类似于上面二维dp的上一层
                    newDp[j] = A[i][j] + Math.min(dp[j], dp[j+1]);
```

```
                } else if(j == m-1){
                    newDp[j] = A[i][j] + Math.min(dp[j-1], dp[j]);
                } else {
                    newDp[j] = A[i][j] + Math.min(dp[j-1], Math.min(dp[j],
dp[j+1]));
                }
            }
            dp = newDp;
        }

        int minSum = Integer.MAX_VALUE;
        for(int j = 0; j < m; j++){
            minSum = Math.min(minSum, dp[j]);
        }
        return minSum;
    }
}
```

在这个解法中，我们使用一个长度为 m 的一维数组 dp 来记录到达当前行每个位置时的最小路径和，初始化 dp 数组为矩阵 A 的第一行。在遍历每一行时，我们使用一个新的一维数组 newDp 来记录更新后的最小路径和，然后将其赋值给 dp 数组，继续处理下一行。在遍历完 dp 数组最后一行时，我们遍历 dp 数组，找出最小值即为所求的最小路径和。

由于我们只需要保存前一行的 dp 值来更新当前行的 dp 值，因此在遍历每一行时，我们可以使用一个新的一维数组来保存更新后的 dp 值。这样我们就可以将空间复杂度降低到 O(m)。

优化后的算法时间复杂度为 $O(nm)$，其中 n 是矩阵的行数，m 是矩阵的列数。

Leetcode 72 编辑距离

题目描述：

给你两个单词 word1 和 word2，请你找到使得 word1 转换成 word2 所使用的最少操作数的操作次数。

你可以对一个单词进行如下三种操作：

插入一个字符 删除一个字符 替换一个字符

示例 1：

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

示例 2：

输入：word1 = "intention", word2 = "execution" 输出：5 解释： intention -> inention (删除 't') inention -> enention (将 'i' 替换为 'e') enention -> exention (将 'n' 替换为 'x') exention -> exection (将 'n' 替换为 'c') exection -> execution (插入 'u')

```
class Solution {
    public int minDistance(String word1, String word2) {
        //dp[]: 记录从前往后的最少编辑距离
        int m = word1.length();
        int n = word2.length();
        int[][] dp = new int[m+1][n+1];

        //把dp的第一行和第一列填满，填上word1和word2对应的length
        for(int i =0; i <= n; i ++){
            dp[0][i] = i;
        }
        for(int i = 0; i <= m; i ++){
            dp[i][0] = i;
        }

        //fill in dp
        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]; //因为不用变了
                                }
                else{
                    dp[i][j] = Math.min(dp[i-1][j-1],Math.min(dp[i-1]
[j],dp[i][j-1]))+1;
                }
            }
        }
        return dp[m][n];

    }
}
```

> 将word1放在dp的第一行， word2放到dp的第一列。注意一开始放的时候条件要是<=。 如果遍历word1和word2相等，那就从左上角去找dp的值。不然的话就要从三个方向去找最小值。

Leetcode 300

题目描述：

给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。

示例 1：

输入：nums = [10,9,2,5,3,7,101,18] 输出：4 解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。

示例 2：

输入：nums = [0,1,0,3,2,3] 输出：4

示例 3：

输入：nums = [7,7,7,7,7,7,7] 输出：1

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        vector<int> dp(nums.size(),1);
        for(int i = 0; i <dp.size(); i++ ){
            //如果发现nums里面在递增的话，去dp里面找
            for(int j = 0; j < i; j++ ){
                if(nums[j] < nums[i]){
                    dp[i] = max(dp[i], dp[j] + 1);
                }
            }
        }
        return *max_element(dp.begin(), dp.end());
    }
};
```

> 每次loop到i位的时候，要回过头去找最大的dp值
>
> 时间复杂度为 O(n^2)，空间复杂度为 O(n)，其中 n 是数组 nums 的长度。

1Leetcode 53

问题描述： 给定一个整数数组nums，找到具有最大和的连续子数组（至少包含一个元素）并返回其和。

示例： 输入：nums = [-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。

```
class Solution {
    public int maxSubArray(int[] nums) {
        //dp: 维护两个变量
        int currSum = nums[0];//当前元素和dp【i-1】，比较出最大值
        int maxSum = nums[0];//最大值

        for(int i = 1;i < nums.length; i ++){
            currSum = Math.max(nums[i], currSum + nums[i]);
            maxSum = Math.max(currSum,maxSum);
        }

        return maxSum;
    }
}
```

Leetcode 1143 最长公共子序列

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
```

```
        int m = text1.length();  // 计算 text1 的长度
        int n = text2.length();  // 计算 text2 的长度

        int[][] dp = new int[m + 1][n + 1];  // 创建一个二维数组 dp，其中
dp[i][j] 表示 text1 前 i 个字符和 text2 前 j 个字符的最长公共子序列的长度

        // 计算最长公共子序列的长度
        for (int i = 1; i <= m; i++) {  // 遍历 text1 的每个字符
            for (int j = 1; j <= n; j++) {  // 遍历 text2 的每个字符
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {  // 如果
字符相等 注意这里是i-1！！！
                    dp[i][j] = dp[i - 1][j - 1] + 1;  // 当前位置的最长公共子
序列长度为左上角的值加 1
                } else {  // 如果字符不相等
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);  // 当
前位置的最长公共子序列长度为上方和左方的最大值
                }
            }
        }

        return dp[m][n];  // 返回最长公共子序列的长度
    }
}
```

> 代码中使用动态规划思想，首先定义一个 `dp` 数组，其中 `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。然后进行循环，如果 `text1[i-1]` 和 `text2[j-1]` 相等，那么当前位置的最长公共子序列长度就是 `dp[i-1][j-1]` 加上 1，否则就是 `dp[i-1][j]` 和 `dp[i][j-1]` 中较大的那个。最后返回 `dp[m][n]` 就是 `text1` 和 `text2` 的最长公共子序列的长度。
>
> 这个算法的时间复杂度为 $O(mn)$，其中 m 和 n 分别是两个字符串的长度。这是因为我们需要遍历两个字符串的所有字符，并且对于每个字符，需要进行一次常数时间的比较和状态转移操作。
>
> 这个算法的空间复杂度为 $O(mn)$，因为我们需要创建一个 $m+1$ 行、$n+1$ 列的二维数组来保存状态值。在实际的算法实现中，我们可以使用滚动数组或者原地 DP 的方法来将空间复杂度优化到 $O(\min(m, n))$。

```
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m, n = len(text1), len(text2)

        # 创建二维列表，用于存储动态规划的中间结果
        # dp[i][j] 表示 text1[0:i] 和 text2[0:j] 的最长公共子序列的长度
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        # 遍历 text1 和 text2 的所有子串，进行动态规划
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if text1[i - 1] == text2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                else:
```

```
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

        # 返回 text1 和 text2 的最长公共子序列的长度
        return dp[m][n]
```

Leetcode 583 两个字符串的删除操作

给定两个单词 word1 和 word2 ，返回使得 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 1:

输入: word1 = "sea", word2 = "eat" 输出: 2 解释: 第一步将 "sea" 变为 "ea" ，第二步将 "eat "变为 "ea" 示例 2:

输入：word1 = "leetcode", word2 = "etco" 输出：4

```
class Solution(object):
    def minDistance(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """
        lcs = self.LCS(word1,word2)
        return len(word1) - lcs + len(word2) - lcs


    def LCS(self,text1,text2) -> int:
        m, n = len(text1), len(text2)
        dp = [[0]*(n+1) for i in range (m+1)]
        for i in range(1, m+1):
            for j in range(1, n+1):
                if text1[i-1] == text2[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
        return dp[m][n]
```

> 这道题就是上一道题的变种，先求出两个str的LCS就可以得到答案

Leetcode 712 最小 ASCII 删除和

给定两个字符串s1 和 s2，返回 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。

```
class Solution {
    public int minimumDeleteSum(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();
        int[][] dp = new int [m+1][n+1];
        //fill int the first row and column
        for(int i = 1; i <= m; i ++){
            dp[i][0] = dp[i-1][0]+s1.codePointAt(i-1);
        }
        for(int j = 1; j <=n; j ++){
            dp[0][j] = dp[0][j-1]+s2.codePointAt(j-1);
        }
        for(int i = 1; i <=m; i ++){
            for(int j = 1; j <=n; j ++){
                if(s1.charAt(i-1) == s2.charAt(j-1)){
                    dp[i][j] = dp[i-1][j-1];
                }
                else{
                    dp[i][j] = Math.min((dp[i-1][j] +s1.codePointAt(i-1)),
(dp[i][j-1] + s2.codePointAt(j-1)));
                }
            }
        }
        return dp[m][n];

    }
}
```

> 和上一道题不同要注意的是，我们第一行第一列要储存s1和s2到i位置的ascii值，左上角是空的。
>
> 其次，我们发现不想等的时候，应该是dp前面的值加上s1/s2的ascii值。
>
> 用到了Java codepointat

```
class Solution:
    def minimumDeleteSum(self, s1: str, s2: str) -> int:
        m ,n= len(s1),len(s2)
        dp = [[0]* (n+1) for i in range (m+1)]
        #initialize dp
        for i in range (1,m+1):
            dp[i][0] = dp[i-1][0] + ord(s1[i-1])
        for j in range (1,n+1):
            dp[0][j] = dp[0][j-1] + ord(s2[j-1])

        for i in range (1, m+1):
            for j in range (1, n+1):
                if s1[i-1] == s2[j-1]:
```

```
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = min((dp[i][j-1] + ord(s2[j-1])),(dp[i-1][j]
+ ord(s1[i-1])))
        return dp[m][n]
```

Leetcode516 最长回文子串

题目描述：

给定一个字符串，找到它的最长回文子序列。可以假设字符串的最大长度为1000。

示例：

输入："bbbab" 输出：4 解释：一个可能的最长回文子序列为 "bbbb"。

输入："cbbd" 输出：2 解释：一个可能的最长回文子序列为 "bb"。

```
class Solution {
    public int longestPalindromeSubseq(String s) {
        //2D dp
        int m = s.length();
        int[][] dp = new int[m][m];
        //fill in dp[][]
        for(int i = 0; i < m; i ++){
            dp[i][i] = 1;//since every single element is a LPS
        }
        //loop from the end
        for(int i = m-2 ; i >=0; i --){
            for(int j = i+1; j <m; j ++){
                //if equals,means both s[i] and s[j] are on the LPS, so we
+2 to the middle of it
                if(s.charAt(i) == s.charAt(j)){
                    dp[i][j] = dp[i+1][j-1] + 2;
                }
                else{
                    //not equal: means either s[i] or s[j] isn't on the
LPS
                    dp[i][j] = Math.max(dp[i+1][j],dp[i][j-1]);
                }
            }
        }
        return dp[0][m-1];

    }
}
```

> 首先要注意，为了避免边界问题，这个loop从后往前，注意一下i和j的定义。

> 状态转移： s【i】 和s【j】做比较从尾部开始。如果想等，说明两人都在LPS上，就把窗口里边一个+2（看代码）。
>
> 如果不想等，说明至少有一个不在LPS上，退回dp【】【】去找。

```
class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        #2d dp
        m = len(s)
        dp = [[0]* m for i in range (m)]
        #fill in dp
        for i in range (m):
            dp[i][i] = 1
        #loop from the end: avoid out of bound error
        for i in range (m-2,-1,-1):
            for j in range (i+1,m):
                if s[i] == s[j]:
                    dp[i][j] = dp[i+1][j-1] + 2
                else:
                    dp[i][j] = max(dp[i+1][j], dp[i][j-1])
        return dp[0][m-1]
```

> Notice: the usgae of for loop!!!

背包问题

N = 3, W = 4 wt = [2, 1, 3] val = [4, 2, 3]

算法返回 6，选择前两件物品装进背包，总重量 3 小于 W，可以获得最大价值 6。

```
public int knapsack(int W, int[] wt, int[] val, int n) {
    int[][] dp = new int[n+1][W+1]; // 创建二维数组，用于记录最大价值
    for (int i = 0; i <= n; i++) {  // 初始化边界条件
        for (int j = 0; j <= W; j++) {
            if (i == 0 || j == 0) { // 背包容量为0或者没有物品时，最大价值都为0
                dp[i][j] = 0;
            } else if (wt[i-1] <= j) { // 当前物品可以放入背包中
                dp[i][j] = Math.max(dp[i-1][j], val[i-1] + dp[i-1][j-wt[i-
1]]); // 选择放入或者不放入物品
            } else { // 当前物品不能放入背包中
                dp[i][j] = dp[i-1][j];
            }
        }
    }
    return dp[n][W]; // 返回最大价值
}
```

> i:背包的第i个元素

> j： 多少重量
>
> 状态转移：如果发现第i个元素不能放到背包（超重）：那么就找i-1元素对应的dp
>
> 如果能放，那么就看Math.max(dp[i-1][j], val[i-1] + dp[i-1][j-wt[i-1]]);

Leetcode416分割等和子集

给你一个 只包含正整数 的 非空 数组 nums 。请你判断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。

示例 1：

输入：nums = [1,5,11,5] 输出：true 解释：数组可以分割成 [1, 5, 5] 和 [11] 。

```
class Solution {
    public boolean canPartition(int[] nums) {
        //将问题转换成，把nums里面的元素值看成重量，放到sum/2的背包中能不能放满
        //compute sum
        int n = nums.length;
        int sum = 0;
        for(int num: nums){
            sum += num;
        }
        if(sum%2 == 1) return false;
        sum = sum/2;
        boolean[][] dp = new boolean[n+1][sum+1];
        //initialize first column in dp: for 0 weight bag,we can fill in
with any element
        for(int i = 0;i <= n; i ++){
            dp[i][0] = true;
        }
        //dp[i][j]看看前i个元素在j的重量下能不能装满
        for(int i = 1; i <=n; i ++){
            for(int j = 1;j <= sum; j ++){
                if(j - nums[i - 1] < 0){
                    //means we cannot put nums[i] into bag
                    dp[i][j] = dp[i-1][j];
                }
                else{
                    //看去掉i元素或者去掉i元素的重量
                    dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i-1]];
                }
            }
        }
        return dp[n][sum];
    }
}
```

> 题目变成：前n个元素能不能把sum/2的背包放满

> 注意：dp的索引

```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        n = len(nums)
        sums = sum(nums)
        if sums % 2 == 1:
            return False

        target = sums // 2
        dp = [[False] * (target + 1) for _ in range(n+1)]

        for i in range(n+1):
            dp[i][0] = True

        for i in range(1, n+1):
            for j in range(1, target+1):
                if j - nums[i-1] >= 0:
                    dp[i][j] = dp[i-1][j] or dp[i-1][j-nums[i-1]]
                else:
                    dp[i][j] = dp[i-1][j]

        return dp[n][target]
```

Leetcode 518 零钱兑换II

```java
class Solution {
    public int change(int amount, int[] coins) {
        //dp[][]背包问题: dp[i][j] use ith-element can get j solutions
        int n = coins.length;
        int[][] dp = new int [n + 1][amount + 1];
        //fill in dp[][]
        for (int i = 0;i <= n; i ++){
            dp[i][0] = 1;//means if we do nothing we can compute value of
0, so solution be 1

        }
        for (int i = 1; i <=n; i ++){
            for(int j = 1;j <= amount; j ++){
                if(coins[i-1] <= j){
                    //means coins[i-1] is possible solution, so dp[i][j]
depends on 扣掉i这个元素能有几种解法 + 前i-1个元素有几种解法
                    dp[i][j] = dp[i-1][j] + dp[i][j - coins[i-1]];
                }
                else{
                    //cannot fill in package
                    dp[i][j] = dp[i-1][j];
                }
            }
```

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

    }
```

> 注意两层for loop中的if条件

```
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        n = len(coins)
        dp = [[0]* (amount + 1) for i in range (n+1)]
        for i in range(n+1):
            dp[i][0] = 1
        for i in range(1,n+1):
            for j in range(1,amount + 1):
                if coins[i-1] <= j:
                    dp[i][j] = dp[i-1][j] + dp[i][j-coins[i-1]]
                else:
                    dp[i][j] = dp[i-1][j]
        return dp[n][amount]
```

0Leetcode 493 目标和

给你一个整数数组 nums 和一个整数 target 。

向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个 表达式 ：

例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，然后串联起来得到表达式 "+2-1" 。 返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。

示例 1:

输入：nums = [1,1,1,1,1], target = 3 输出：5 解释：一共有 5 种方法让最终目标和为 3 。 -1 + 1 + 1 + 1 + 1 = 3 +1 - 1 + 1 + 1 + 1 = 3 +1 + 1 - 1 + 1 + 1 = 3 +1 + 1 + 1 - 1 + 1 = 3 +1 + 1 + 1 + 1 - 1 = 3

```
//回溯算法
class Solution {
    int result = 0;
    public int findTargetSumWays(int[] nums, int target) {
        //use backtrack solution
        backtrack(nums, 0, target);//our goal is change target to 0
        return result;

    }
    void backtrack(int[] nums, int start, int remain){
        //base case
        if(start == nums.length){
            if (remain == 0){//we find a possibile answer
```

```
                result++;

            }
            return;
        }
        remain += nums[start];
        backtrack(nums,start+1,remain);
        remain -= nums[start];

        remain -= nums[start];
        backtrack(nums, start+1, remain);
        remain += nums[start];

    }
}
```

> 注意base case 中： 不管我们remain是不是等于0，都要return。
>
> 下面的撤销选择！

```
class Solution:
    result = 0
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        # revise backtrack
        self.backtrack(nums, 0,target)
        return self.result

    def backtrack(self,nums:List[int], i : int, remain: int):
        if i == len(nums):
            if remain == 0:
                self.result += 1
            return
        remain += nums[i]
        self.backtrack(nums, i + 1, remain)
        remain -= nums[i]

        remain -= nums[i]
        self.backtrack(nums, i + 1, remain)
        remain += nums[i]
```

> Caution: using self.

优化：dp

```
class Solution {
    //首先，如果我们把 nums 划分成两个子集 A 和 B，分别代表分配 + 的数和分配 - 的数，
那么他们和 target 存在如下关系：
    /*sum(A) - sum(B) = target
```

```
        sum(A) = target + sum(B)
        sum(A) + sum(A) = target + sum(B) + sum(A)
        2 * sum(A) = target + sum(nums) */
        public int findTargetSumWays(int[] nums, int target) {
            //问题变成了，nums里有多少个子集A能装满target + sum(nums)/2的背包
            int sum = 0;
            for(int num: nums){
                sum += num;
            }
            //two conditions we have no answer
            if(sum < Math.abs(target) || (target + sum)%2 == 1) return 0;
            return subset(nums,(target + sum)/2);
        }
        int subset(int[] nums, int sum){
            //dp
            int n = nums.length;
            int[][] dp = new int[n+1][sum+1];
            for(int i = 0; i <= n ; i ++){
                dp[i][0] = 1;//do nothing count as one solution
            }
            for(int i = 1; i <=n; i ++){
                for(int j = 1; j <= sum; j ++){
                    if(nums[i-1] > j){
                        //means nums[i-1] cannot fill bag
                        dp[i][j] = dp[i-1][j];
                    }
                    else{
                        dp[i][j] = dp[i-1][j] + dp[i-1][j-nums[i-1]];//如果可以
装进背包。我们看去掉这个选项的答案和去掉这个重量的答案
                    }
                }
            }
            return dp[n][sum];
        }
    }
```

> 理解 dp[i][j] = dp[i-1][j] + dp[i-1][j-nums[i-1]];：如果我当前元素是可以放的，那么我就去看前i-1个元素能放进去的解决方法和凑出重量-当前元素重量的解决方法
>
> 比如[1, 2, 3] target= 5
>
> 我现在前两个元素一共有三种方法凑出5， 加了3之后，结果 = 前两元素凑出5 + 前两元素凑出 5-3
>
> 时间复杂度：该算法使用了一个二维数组dp进行动态规划，需要遍历整个数组，因此时间复杂度为O(n * sum)，其中n是nums的长度，sum是nums数组中所有元素的和。
>
> 空间复杂度：该算法使用了一个二维数组dp进行动态规划，其大小为(n+1) * (sum+1)，因此空间复杂度为O(n * sum)

Leetcode 64 最小路径和

给定一个包含非负整数的 m x n 网格 grid ，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

示例 1:

输入：grid = [[1,3,1],[1,5,1],[4,2,1]] 输出：7 解释：因为路径 1→3→1→1→1 的总和最小。

```
class Solution {
    public int minPathSum(int[][] grid) {
        //the variables we need
        int m = grid.length, n = grid[0].length;
        //do it with one dimension dp
    int[] dp = new int[n];
    Arrays.fill(dp,Integer.MAX_VALUE);
    dp[0] = 0;
    //loop from level 2
    for(int i = 0; i < m; i ++){
        dp[0] += grid[i][0];
        for(int j = 1; j < n; j++ ){
            dp[j] = Math.min(dp[j], dp[j-1]) + grid[i][j];
        }

    }
    return dp[n-1];

    }
}
```

> 注意： 为了第一波能找到min，我们要先拿max填满dp；然后要把第一个元素设成0.
>
> 注意状态转移。

```
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        m ,n = len(grid), len(grid[0])
        dp = [0] * n
        dp[0] = grid[0][0]
        for i in range (1,n):
            dp[i] = dp[i-1] + grid[0][i]
        for i in range (1, m):
            dp[0] += grid[i][0]
            for j in range(1,n):
                dp[j] = grid[i][j] + min(dp[j],dp[j-1])
        return dp[-1]
```

> python貌似没有max_value这个东西。

Leetcode198 打家劫舍

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 。

```
class Solution {
    public int rob(int[] nums) {
        //dp: the max money for nums[i]
        int[] dp = new int[nums.length];
        //base case
        int n = dp.length;
        if(n == 1){
            return nums[0];
        }
        if(n ==2){
            return Math.max(nums[0],nums[1]);
        }
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0],nums[1]);
        for(int i = 2; i < dp.length; i ++){
            dp[i] = Math.max(dp[i-1],dp[i-2] + nums[i]);
        }
        return dp[dp.length-1];
    }
}
```

> 注意：我们for loop从索引2开始，所以要判断两个cornor case；
>
> 状态转移： i索引要么打劫，那么结果就是前前个格子加上nums本身，要么不打劫就是前面一个格子的值。
>
> 该函数的时间复杂度为 $O(n)$，其中 n 是输入数组 $nums$ 的长度，因为需要遍历整个数组一次。
>
> 该函数的空间复杂度为 $O(n)$，因为需要创建一个长度为 n 的数组 dp 来存储每个位置的最大收益。

优化版：不需要dp数组

```
class Solution {
    public int rob(int[] nums) {
        //dont need to use dp[]
        //cornor case
        int n = nums.length;
        if(n == 1) return nums[0];
```

```
        if(n == 2) return Math.max(nums[0], nums[1]);
        int dp_0 = nums[0];
        int dp_1 = Math.max(nums[0], nums[1]);
        int dp_2 = -1;
        for(int i = 2; i < n; i ++){
            dp_2 = Math.max(dp_0 + nums[i],dp_1);
            //update these variables
            dp_0 = dp_1;
            dp_1 = dp_2;
        }
        return dp_2;
    }
}
```

```
class Solution:
    def rob(self, nums: List[int]) -> int:
        n = len(nums)
        #base case
        if n == 1: return nums[0]
        if n == 2: return max(nums[0],nums[1])
        dp_0 = nums[0]
        dp_1 = max(nums[0],nums[1])
        dp_2 = -1
        for i in range(2,n):
            dp_2 = max(dp_0 + nums[i] , dp_1)
            dp_0 = dp_1
            dp_1 = dp_2
        return dp_2
```

Leetcode213 打家劫舍II

213. 打家劫舍 II

难度中等1279收藏分享切换为英文接收动态反馈

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** ，今晚能够偷窃到的最高金额。

示例 1：

```
输入：nums = [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。
```

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 1: return nums[0]
        return max(self.robRange(nums, 0, n - 2), self.robRange(nums, 1, n
- 1))

    def robRange(self, nums: List[int], start: int, end: int) -> int:
        n = len(nums)
        dp_i_1 = dp_i_2 = dp_i = 0
        for i in range(end, start - 1, -1):
            dp_i = max(dp_i_1, nums[i] + dp_i_2)
            dp_i_2 = dp_i_1
            dp_i_1 = dp_i
        return dp_i
```

> 把问题变成，(self.robRange(nums, 0, n - 2), self.robRange(nums, 1, n - 1)) 算这两个区间的rob

Leetcode 337 打家劫舍III

打家劫舍在binary tree里面

```java
/**
 * 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 {
    public int rob(TreeNode root) {
        int[] res = helper(root);
        return Math.max(res[0], res[1]);
    }
    /**
    arr[0] the max value by rob the root
    arr[1] no rob the root */
    int[] helper(TreeNode root){
        if(root == null){
            return new int[]{0,0};
```

```
        }
        int[] left = helper(root.left);
        int[] right = helper(root.right);
        int rob_root = root.val + left[1] + right[1];
        int not_rob = Math.max(left[0],left[1])+
Math.max(right[0],right[1]);
        return new int[]{rob_root,not_rob};
    }
}
```

时间复杂度： 这段代码使用了递归来遍历整棵树，对于每个节点都只访问了一次，因此时间复杂度是O(N)，其中 N 是节点数。

空间复杂度： 这段代码使用了递归来遍历整棵树，因此递归的深度是树的高度 h，最坏情况下为N（退化成链表的情况）。对于每个递归层次，需要使用一个长度为2的整数数组，因此空间复杂度是O(h)。由于 h 最坏情况下为 N，因此空间复杂度是 O(N)。

这段代码是用来解决 "House Robber III" 问题，这个问题要求在二叉树结构的房子里，不能同时抢劫相邻的节点，求最大的抢劫价值。

这段代码的解决思路是采用递归的方式对每个节点进行处理，对于每个节点，有两种情况：

- 抢劫当前节点：由于不能同时抢劫相邻节点，因此抢劫当前节点就必须跳过其子节点，但是可以抢劫其孙子节点。所以当前节点的价值为 root.val + left[1] + right[1]，其中 left[1] 和 right[1] 分别表示左右子节点不抢劫的最大价值。
- 不抢劫当前节点：由于不抢劫当前节点，可以选择抢劫其左右子节点或者不抢劫。所以当前节点的价值为 Math.max(left[0], left[1]) + Math.max(right[0], right[1])。

递归的基准情况是节点为空，此时抢劫价值为0。

最终，将根节点的两种情况的最大值返回即可。

其中，helper() 函数返回一个长度为2的整数数组，arr[0] 表示抢劫当前节点的最大价值，arr[1] 表示不抢劫当前节点的最大价值。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def rob(self, root: Optional[TreeNode]) -> int:
        res = self.helper(root)
        return max(res[0], res[1])

    def helper(self,root: Optional[TreeNode]) -> List[int]:
        if not root:
            return [0,0]
        left = self.helper(root.left)
        right = self.helper(root.right)
```

```
        rob_root = root.val + left[1] + right[1]
        not_rob = max(left[0],left[1]) + max(right[0], right[1])
        return [rob_root, not_rob]
```

力扣46 全排列

```
class Solution {
    boolean[] used;
    List<List<Integer>> res;
    LinkedList<Integer> track;
    public List<List<Integer>> permute(int[] nums) {
        track = new LinkedList<>();
        res = new LinkedList<>();
        used = new boolean[nums.length];
        backtrack(nums,track);
        return res;

    }
    void backtrack(int[] nums, LinkedList<Integer> track){
        //base case
        if(track.size() == nums.length){
            res.add(new LinkedList(track));
        }
        for(int i = 0; i < nums.length; i ++){
            if(used[i]){
                continue;
            }
            used[i] = true;
            track.add(nums[i]);
            backtrack(nums,track);
            track.removeLast();;
            used[i] = false;
        }

    }
}
```

> track要用LinkedList类型，因为要用到removeLast这个api。
>
> 该算法的时间复杂度为O(N*N!)，*其中N为数组的长度。因为对于每个元素，都需要进行一次回溯，而回溯的总次数是N!，同时还需要在每次回溯时，遍历数组中未被使用的元素，这需要花费O(N)的时间。所以总时间复杂度为O(N*N!)*。
>
> 空间复杂度方面，该算法使用了一个布尔型数组used和一个链表track，以及一个结果列表res。其中used数组和track链表的长度都是N，所以它们的总空间复杂度为O(N)。而结果列表res的长度为N!，因为全排列的总数为N!，所以它的空间复杂度为O(N!)。因此，该算法的总空间复杂度为O(N*N!)。

```python
class Solution:
    #global variables
    def __init__(self):
        self.used = None
        self.track = None
        self.res = None

    def permute(self, nums: List[int]) -> List[List[int]]:
        #initialize variables
        self.track = []
        self.res = []
        self.used = [False] * len(nums)
        self.backtrack(nums, self.track)
        return self.res

    def backtrack(self, nums: List[int], track:List[int]):
        #base case
        n = len(nums)
        if n == len(track):
            self.res.append(track[:])
            return
        for i in range(n):
            if self.used[i]:
                continue
            self.used[i] = True
            track.append(nums[i])
            self.backtrack(nums, track)
            self.used[i] = False
            track.pop()
```

> 注意这里面track在backtrack中不需要self因为函数里面调用的不是全局的track

两数相加

```python
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next

class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        head = None
```

```
        tail = None
        carry = 0
        while l1 or l2:
            n1 = l1.val if l1 else 0
            n2 = l2.val if l2 else 0
            sum  = n1 + n2 + carry
            carry = sum // 10
            i = sum % 10
            curr = ListNode(i)

            if not head:
                head = tail = curr
            else:
                tail.next = curr
                tail = tail.next

            if l1:
                l1 = l1.next
            if l2:
                l2 = l2.next

        if carry > 0:
            tail.next = ListNode(carry)

        return head
```

> 注意我们的逻辑!

presum

```
class NumArray(object):

    def __init__(self, nums):
        """
        :type nums: List[int]
        """
        self.presum = [0]
        for num in nums:
            self.presum.append(self.presum[-1] + num)


    def sumRange(self, left, right):
        """
        :type left: int
        :type right: int
        :rtype: int
        """
        return self.presum[right+1] - self.presum[left]
```

```
# Your NumArray object will be instantiated and called as such:
# obj = NumArray(nums)
# param_1 = obj.sumRange(left,right)
```

区间加法

假设你有一个长度为 n 的数组，初始情况下所有的数字均为 0，你将会被给出 k 个更新的操作。

其中，每个操作会被表示为一个三元组：[startIndex, endIndex, inc]，你需要将子数组 A[startIndex ... endIndex]（包括 startIndex 和 endIndex）增加 inc。

请你返回 k 次操作后的数组。

示例:

输入: length = 5, updates = [[1,3,2],[2,4,3],[0,2,-2]] 输出: [-2,0,3,5,3]

```
class Solution(object):
    def getModifiedArray(self, length, updates):
        """
        :type length: int
        :type updates: List[List[int]]
        :rtype: List[int]
        """
        diff = [0 for i in range(length+1)]
        for update in updates:
            start, end = update[0], update[1]
            diff[start] += update[2]
            if end + 1 < length:
                diff[end + 1] -= update[2]
        num = [0 for i in range(length)]
        num[0] = diff[0]
        for i in range(1,length):
            num[i] = num[i-1] + diff[i]
        return num
```

> 差分数组