王争的算法训练营

习题课：匹配问题&其他

重中之重，大厂笔试、面试必考项、学习难但面试不难、掌握模型举一反三

DP专题：

- 专题：适用问题
- 专题：解题步骤
- 专题：最值、可行、计数三种类型　一些特殊小类别：树形DP、区间DP、数位DP
- 专题：空间优化

经典模型：

- 背包问题（0-1、完全、多重、二维费用、分组、有依赖的）
- 路径问题
- 打家劫舍&股票买卖
- 爬楼梯问题
- 匹配问题（LCS、编辑距离）
- 其他（LIS）

配套习题（24）：

背包：

416. 分割等和子集
494. 目标和
322. 零钱兑换
518. 零钱兑换 II

路径问题

64. 最小路径和

剑指 Offer 47. 礼物的最大价值

120. 三角形最小路径和
62. 不同路径
63. 不同路径 II

打家劫舍 & 买卖股票：

198. 打家劫舍
213. 打家劫舍 II
337. 打家劫舍 III (树形DP)
714. 买卖股票的最佳时机含手续
309. 最佳买卖股票时机含冷冻期

爬楼梯问题

70. 爬楼梯
322. 零钱兑换
518. 零钱兑换 II

剑指 Offer 14- I. 剪绳子

剑指 Offer 46. 把数字翻译成字符串

139. 单词拆分

匹配问题

1143. 最长公共子序列
72. 编辑距离

其他

437. 路径总和 III (树形DP)
300. 最长递增子序列

匹配问题

[1143. 最长公共子序列]

[72. 编辑距离]

习题课：动态规划

1143. 最长公共子序列

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。如果不存在 **公共子序列**，返回 `0`。

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。

如果t1[i] == t2[j]，有三种决策：（i+1, j+1）（i+1，j不变）（i不变，j+1）
如果t1[i] != t2[j]，有两种决策：（i+1, j+1）（i+1，j不变）（i不变，j+1）

到达（i, j）这个状态，也就是说：开始匹配t1[i]和t2[j]了，
只有可能从上一个阶段的这几个状态转移过来：（i-1, j）、（i, j-1）、（i-1, j-1）

如果原状态是（i-1，j），那么i+1，j不变，得到（i，j）这个状态
如果原状态是（i，j-1），那么i不变，j+1，得到（i，j）这个状态
如果原状态是（i-1，j-1）那么i+1,j+1，得到（i，j）这个状态

int dp[n+1][m+1];
dp[i][j]表示长度为i的t1子串和长度是j的t2子串的最长公共子序列长度
也就是说：t1[0, i-1]和t2[0, j-1]的最长公共子序列长度
也就是说：开始匹配t1[i]和t2[j]时的最长公共子序列长度

那么：
dp[i][j] = max(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]+1) 如果t1[i-1] == t2[j-1]
dp[i][j] = max(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) 如果t1[i-1] != t2[j-1]

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 。
```

```java
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        int n = text1.length();
        int m = text2.length();
        char[] t1 = text1.toCharArray();
        char[] t2 = text2.toCharArray();

        // dp[i][j] 表示text1[0~i-1](长度为i子串)和text2[0~j-1](长度j的子串)的LCS
        int dp[][] = new int[n+1][m+1];
        for (int j = 0; j <= m; ++j) {
            dp[0][j] = 0;
        }
        for (int i = 0; i <= n; ++i) {
            dp[i][0] = 0;
        }

        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= m; ++j) {
                if (t1[i-1] == t2[j-1]) {
                    dp[i][j] = max3(dp[i-1][j-1]+1, dp[i-1][j], dp[i][j-1]);
                } else {
                    dp[i][j] = max3(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]);
                }
            }
        }

        return dp[n][m];
    }

    private int max3(int a, int b, int c) {
        int maxv = a;
        if (maxv < b) maxv = b;
        if (maxv < c) maxv = c;
        return maxv;
    }
}
```

72. 编辑距离

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

你可以对一个单词进行如下三种操作：

- 插入一个字符
- 删除一个字符
- 替换一个字符

示例 1：

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

示例 2：

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

如果t1[i] == t2[j]，有三种决策：（i+1, j+1）（i+1，j不变）（i不变，j+1）
如果t1[i] != t2[j]，有两种决策：（i+1，j+1）（i+1，j不变）（i不变，j+1）

到达（i, j）这个状态，也就是说：开始匹配t1[i]和t2[j]了，
只有可能从上一个阶段的这几个状态转移过来：（i-1, j）、（i, j-1）、（i-1, j-1）

如果原状态是（i-1，j），那么i+1，j不变，得到（i，j）这个状态
如果原状态是（i，j-1），那么i不变，j+1，得到（i，j）这个状态
如果原状态是（i-1，j-1），那么i+1,j+1，得到（i，j）这个状态

dp[i][j]表示长度为i的t1子串和长度是j的t2子串的编辑距离
也就是说：t1[0, i-1]和t2[0, j-1]的编辑距离
也就是说：开始匹配t1[i]和t2[j]时的编辑距离

那么：
dp[i][j] = min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]) 如果t1[i-1] == t2[j-1]
dp[i][j] = min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+1) 如果t1[i-1] != t2[j-1]

```java
class Solution {
    public int minDistance(String word1, String word2) {
        int n = word1.length();
        int m = word2.length();
        if (n == 0) return m;
        if (m == 0) return n;
        char[] w1 = word1.toCharArray();
        char[] w2 = word2.toCharArray();
        // dp[i][j]表示w1[0~i-1](长度为i子串)和w2[0~j-1](长度为j的子串)的最小编辑距离
        int[][] dp = new int[n+1][m+1];
        for (int j = 0; j <= m; j++) {
            dp[0][j] = j;
        }
        for (int i = 0; i <= n; i++) {
            dp[i][0] = i;
        }

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (w1[i-1] == w2[j-1]) {
                    dp[i][j] = min3(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]);
                } else {
                    dp[i][j] = min3(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+1);
                }
            }
        }
        return dp[n][m];
    }

    private int min3(int n1, int n2, int n3) {
        return Math.min(n1, Math.min(n2, n3));
    }
}
```

其他

300. 最长递增子序列

437. 路径总和 III (树形DP)

300. 最长递增子序列

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。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列。

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
```

提示：

- `1 <= nums.length <= 2500`
- $-10^4 <= nums[i] <= 10^4$

进阶：

- 你可以设计时间复杂度为 $O(n^2)$ 的解决方案吗?
- 你能将算法的时间复杂度降低到 `O(n log(n))` 吗?

构建多阶段决策模型，每一阶段决策一个数字，是否放入递增子序列中。
1）如果当前数字小于等于已经放入递增子序列中的最后一个数字，那么这个数字只能选择不放入递增子序列。
2）如果当前数字大于已经放入递增子序列中的最后一个数字，可以选择将其放入递增子序列，也可以选择不放入递增子序列。
所以决策阶段都做完之后，就找到了一个递增子序列。利用回溯算法穷举所有的递增子序列，比较出最长的那个

是否能用dp解决呢？记录某个阶段决策完之后的最大递增子序列长度？

dp[i]记录以nums[i]为结尾的最大递增子序列长度
那在这样一个递增子序列中（以nums[i]结尾），
上一个数字为：nums[j] （0<=j<i && nums[j-1]<nums[i]）中的任意一个，
所以：dp[i] = max(dp[j]+1)，其中：0<=j<i && nums[j-1]<nums[i]

2 1 6 4 5 8 7 3

第一次遇见确实比较难想到解法，记住！记住！
与其说是一种题解，不如说是一种算法

习题课：动态规划


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

dp[i]记录以nums[i]为结尾的序列的最大长度
那在这样一个递增子序列中（以nums[i]结尾），
上一个数字为：nums[j] （0<=j<i && nums[j-1]<nums[i]）中的任意一个，
所以：dp[i] = max(dp[j]+1)，其中：0<=j<i && nums[j-1]<nums[i]

lisToMinV数组记录当考察到第i个元素时，当前的所有可能的子序列长度，以及每个子序列长度对应的序列中最后一个元素的最小值

2 1 6 4 5 8 7 3

```java
class Solution {
    public int lengthOfLIS(int[] nums) {
        int n = nums.length;

        int[] lisToMinV = new int[n+1];
        int k = 0;

        int[] dp = new int[n];
        for (int i = 0; i < n; ++i) {
            int len = bsearch(lisToMinV, k, nums[i]);
            if (len == -1) {
                dp[i] = 1;
            } else {
                dp[i] = len+1;
            }
            if (dp[i] > k) {
                k = dp[i];
                lisToMinV[dp[i]] = nums[i];
            } else if (lisToMinV[dp[i]] > nums[i]) {
                lisToMinV[dp[i]] = nums[i];
            }
        }

        int result = 0;
        for (int i = 0; i < n; ++i) {
            if (dp[i] > result) result = dp[i];
        }
        return result;
    }

}
```

```java
// 查找最后一个比target小的元素位置
private int bsearch(int[] a, int k, int target) {
    int low = 1;
    int high = k;
    while (low <= high) {
        int mid = (low+high)/2;
        if (a[mid]<target) {
            if (mid==k || a[mid+1]>=target) {
                return mid;
            } else {
                low = mid+1;
            }
        } else {
            high = mid-1;
        }
    }
    return -1;
}
```

437. 路径总和 III

每个节点用一个map记录以此节点为上端点的路径的所有可能的长度(key)对应的路径的个数(value)

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过1000个节点，且节点数值范围是 [-1000000,1000000] 的整数。

示例:

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

返回 3。和等于 8 的路径有:

1.  5 -> 3
2.  5 -> 2 -> 1
3.  -3 -> 11
```

```java
class Solution {
    private int count = 0;
    public int pathSum(TreeNode root, int sum) {
        dfs(root, sum);
        return count;
    }

    // 返回以root为上端点的路径的所有可能的长度（key）对应的路径的个数（value)
    private Map<Integer, Integer> dfs(TreeNode root, int sum) {
        if (root == null) return new HashMap<>();
        Map<Integer, Integer> leftValues = dfs(root.left, sum);
        Map<Integer, Integer> rightValues = dfs(root.right, sum);
        Map<Integer, Integer> rootValues = new HashMap<>();
        rootValues.put(root.val, 1);
        for (Map.Entry<Integer, Integer> entry : leftValues.entrySet()) {
            int newKey = entry.getKey()+root.val;
            int newValue = entry.getValue();
            if (rootValues.containsKey(newKey)) {
                newValue += rootValues.get(newKey);
            }
            rootValues.put(newKey, newValue);
        }
        for (Map.Entry<Integer, Integer> entry : rightValues.entrySet()) {
            int newKey = entry.getKey() + root.val;
            int newValue = entry.getValue();
            if (rootValues.containsKey(newKey)) {
                newValue += rootValues.get(newKey);
            }
            rootValues.put(newKey, newValue);
        }
        for (Map.Entry<Integer, Integer> entry : rootValues.entrySet()) {
            if (entry.getKey() == sum) {
                count += entry.getValue();
            }
        }
        return rootValues;
    }
}
```

```java
class Solution {
    private int count = 0;
    public int pathSum(TreeNode root, int sum) {
        dfs(root, new ArrayList<>(), sum);
        return count;
    }

    private void dfs(TreeNode root, List<Integer> path, int sum) {
        if (root == null) return;
        path.add(root.val);

        int curSum = 0;
        for (int i = path.size()-1; i >= 0; --i) {
            curSum += path.get(i);
            if (curSum == sum) count++;
        }

        dfs(root.left, path, sum);
        dfs(root.right, path, sum);

        path.remove(path.size()-1);
    }
}
```

提问环节