LeetCode Memo

Xiyuan Yang
2025.10.22

Only programmers who do a LeetCode Hard problem every day deserve to be called programmers :)

Contents

§1. Introduction

Recordings for my LeetCode Journey.

§1.1. Table of Contents

2025/09/17	重新开始 LeetCode 之旅，配置好在 Vscode Ubuntu 中刷 LeetCode 的环境
2025/09/17	开始算法第一个章节：动态规划刷题

§2. Dynamic Programming

§2.1. Classical Problems

§2.1.1. T72 Edit Distance

§2.1.1.1. Problems

- Url: https://leetcode.com/problems/edit-distance/

Problem 2.1.1.1.1 (T72).
Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character
- Example 1:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

- Example 2:

```
Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')
```

- Constraints:
  - 0 <= word1.length, word2.length <= 500
  - word1 and word2 consist of lowercase English letters.

§2.1.1.2. 前缀数组的考虑

Recordings (前缀数组).

- 动态规划的核心在于找到合适的子问题结构并且构建不同参数的子问题之下的动态联系（从图论的角度就是构建一个有向无环图）
- 对于最常见的情况，问题可以被建模为一个数组，尤其是本身的数组问题或者字符串问题，可以考虑前缀数组或者后缀数组作为结构。
  - 因为这样的结构本身包含父子的包含关系，在一些约束中具有良好的结构，可以轻松的写出状态转移方程。

考虑构建前缀和数组，很显然，这里至少需要构造一个二维数组。下面的核心就是 define 好子结构的具体意义是什么？即 `dp[i][j]` 是什么意思。

这里考虑 `dp[i][j]` 代表着 对于字符串 word1 的前 i 个字符的 slice 和字符串 word2 的前 j 个字符的 slice，最少的操作次数是多少。因此创建二维数组的时候长度会比原先字符串的长度多 1 个，可以巧妙的避免空字符串的特判情况。

定义好子结构后，接下来就需要解决两个问题：

- 状态转移方程？
- 初始化？

§2.1.1.3. **1-based** 的情况下初始化的好处

在 1-based 的情况下，初始化很简单，填满 $i = 0$ 和 $j = 0$ 的两条边，因为这也对应着空字符串的情况

§2.1.1.4. 状态转移方程

考虑计算 `dp[i][j]`，我们首先考虑从 `dp[i-1][j-1]` 的状态转移。

- 如果新加入的两个字符是相同的，那不需要做任何操作，并且反证法可以证明这是最优解。
- 如果新加入的两个字符不同，此时就需要状态转移，为了保证最优，我们的操作都对于新加入的字符串操作：
  - 这样简化问题保证正确性，因为这实际上是一个迭代的过程，最优的任何步骤都会在对应的步长时发生在末尾。
  - 在末尾插入：需要考虑 `dp[i-1][j]`
  - 在末尾删除：需要考虑 `dp[i][j-1]`
  - 在末尾替换：需要考虑 `dp[i-1][j-1]`
  - 最后去 min 加 1

§2.1.1.5. Results

```
#include <cmath>
#include <string>
#include <vector>
class Solution {
public:
    int minDistance(std::string word1, std::string word2) {
        int word_length_1 = word1.length();
        int word_length_2 = word2.length();

        std::vector<std::vector<int>> dp(word_length_1 + 1,
std::vector<int>(word_length_2 + 1));

        // initialize
        for (int i = 0; i <= word_length_1; i++) {
            dp[i][0] = i;
        }

        for (int j = 0; j <= word_length_2; j++) {
            dp[0][j] = j;
        }

        if (word_length_1 == 0 && word_length_2 == 0) {
            return dp[word_length_1][word_length_2];
        }


        // start dp!
        for (int i = 1; i <= word_length_1; i++) {
            for (int j = 1; j <= word_length_2; j++) {

                if (word1[i - 1] == word2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    dp[i][j] = 1 + std::min(dp[i - 1][j], std::min(dp[i][j - 1], dp[i
- 1][j - 1]));
                }
            }
```

```
        }

        return dp[word_length_1][word_length_2];
    }
};
```

§2.1.2. T121 Best Time to Buy Stocks

Problem 2.1.2.1 (T121).
You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

§2.1.2.1. Method1 Using Dynamic Programming

为了简化问题和状态转移方程，如果能给 dp 数组降维就需要降维。同样，对于离散的状态标记，也可以增加一个维度来存储，这不会提高时间复杂度。

考虑 L[i][j]:

- i 代表天数: i = 0,1,2,...,n-1
- j 代表状态，即当天是否持有股票（因为只允许交易一次股票，即一次买入和一次卖出）
- 这个值代表该天持有或者不持有股票可能会产生的最大利润。
- 最终状态，在最终，我们肯定是需要卖掉股票，这样才可以回本，因此我们需要计算 L[n-1][0]

状态转移方程:

- 如果在第 i 天持有:
  - dp[i][1] = max(dp[i-1][1], -prices[i])
- 如果在第 i 天不持有:
  - dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])

§3. Divide and Conquer

Recordings (Divide and Conquer).
- 可以通过需要达到的时间复杂度倒退主定理递推式，再分析具体如何做递归
- 有时候数组的切分是比较重要的，但是有时候数据的合并是比较重要的
- 和动态规划类似，核心思想是在递归中不断的有效减小子问题的规模，直到达到基准问题，进而实现复杂度的优化。

§3.1. Median of two sorted arrays

Example (Median of two sorted arrays).
Given two sorted arrays, return the median number for the merged sorted arrays.

For the most simple intuition, we can merge the two arrays with time complexity $O(m+n)$, then find the median number in constant time. Total time complexity: $O(m+n)$.

We can optimize this problem using divide and conquer.

Recordings (Median of the two sorted arrays).

- 更加一般的子问题：寻找两个数组合并结果的第 k 小元素
  - 寻找中位数只是这种情况的特殊情况
- 如何求解这个更加一般的子问题
  - 我们希望通过原先数组的有序性，来避免遍历查找，而是尽可能访问更少的数组元素。

因此，每个数组选择 $\lfloor \frac{k}{2} \rfloor$ 的元素，通过比较最后一个数组元素，就可以排除掉 $\lfloor \frac{k}{2} \rfloor$ 的数组元素。

时间复杂度分析：

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

$$\text{Time Complexity} : O(\log(m+n))$$

```
#
# @lc app=leetcode id=4 lang=python3
#
# [4] Median of Two Sorted Arrays
#

# @lc code=start
from typing import List


class Solution:
    def findMedianSortedArrays(self, nums1: list[int], nums2: list[int]) -> float:
        m = len(nums1)
        n = len(nums2)
        total_len = m + n

        if total_len % 2 == 1:
            k = total_len // 2 + 1
            return self._findKth(nums1, 0, nums2, 0, k)
        else:
            k1 = total_len // 2
            k2 = total_len // 2 + 1
            median1 = self._findKth(nums1, 0, nums2, 0, k1)
            median2 = self._findKth(nums1, 0, nums2, 0, k2)
            return (median1 + median2) / 2.0

    def _findKth(self, nums1: list[int], i: int, nums2: list[int], j: int, k: int) ->
int:
        # i and j means the starting index
        # len1 and len2 are the length of the sub arrays
        len1 = len(nums1) - i
        len2 = len(nums2) - j

        if len1 == 0:
            return nums2[j + k - 1]
        if len2 == 0:
            return nums1[i + k - 1]
```

```
        if k == 1:
            return min(nums1[i], nums2[j])

        half_k = k // 2

        # 确定要比较的元素索引
        idx1 = i + min(len1, half_k) - 1
        idx2 = j + min(len2, half_k) - 1
        val1 = nums1[idx1]
        val2 = nums2[idx2]

        if val1 <= val2:
            # 排除 nums1 中的 [i, idx1] 部分
            new_k = k - (idx1 - i + 1)
            return self._findKth(nums1, idx1 + 1, nums2, j, new_k)
        else: # val2 < val1
            new_k = k - (idx2 - j + 1)
            return self._findKth(nums1, i, nums2, idx2 + 1, new_k)

# @lc code=end
```

§4. Conclusion