DP-LCS

1143. Longest Common Subsequence

```python
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        # bottom up, O(mn)
        m, n = len(text1), len(text2)
        dp = [[0] * (n + 1) for _ 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[-1][-1]

        # top down, O(mn)
        m, n = len(text1), len(text2)

        @lru_cache(None)
        def dfs(i, j):
            if i == m or j == n:
                return 0

            if text1[i] == text2[j]:
                return dfs(i + 1, j + 1) + 1
            else:
                return max(dfs(i + 1, j), dfs(i, j + 1))

        return dfs(0, 0)
```

Given two strings text1 and text2, return the length of their longest common subsequence.

Input: text1 = "abcde", text2 = "ace"

Output: 3

Explanation: The longest common subsequence is "ace" and its length is 3.

72. Edit Distance

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

`Input: word1 = "horse", word2 = "ros"`

`Output: 3`

`Explanation:`

`horse -> rorse (replace 'h' with 'r')`

`rorse -> rose (remove 'r')`

`rose -> ros (remove 'e')`

```python
class Solutiom:
    def minDistance(self, word1: str, word2: str) -> int:
        # ttop down
        m , n = len(word1), len(word2)
        @lru_cache(None)
        def dfs(i, j):
            if i == m or j == n:
                return n - j + m - i

            if word1[i] == word2[j]:
                res = dfs(i+1, j+1)
            else:
                res = min(dfs(i+1, j+1), dfs(i, j+1), dfs(i+1, j)) + 1
            return res

        return dfs(0, 0)

        # Bottom up 2D DP, time O(m * n), space O(m * n)
        m, n = len(word1), len(word2)
        dp = [[float('inf')] * (n + 1) for _ in range(m + 1)]

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

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if word1[i - 1] == word2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1]
                else:
                    # dp[i][j - 1]: delete, dp[i - 1][j]: insert, dp[i - 1][j - 1]: replace
                    dp[i][j] = min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j - 1]) + 1

        return dp[-1][-1]
```

97. Interleaving String

Given strings `s1`, `s2`, and `s3`, find whether `s3` is formed by an interleaving of `s1` and `s2`.

```python
class Solution2:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
        # 2D Top Down DP, time O(n1*n2), space O(n1*n2)
        n1, n2, n3 = len(s1), len(s2), len(s3)

        if s1 == "": return s2 == s3
        if s2 == "": return s1 == s3
        if n1 + n2 != n3: return False

        @lru_cache(None)
        def dfs(i, j):
            if i == 0 and j == 0:
                return True
            if i == 0 and s2[j - 1] == s3[j - 1]:
                return dfs(i, j - 1)
            if j == 0 and s1[i - 1] == s3[i - 1]:
                return dfs(i - 1, j)

            tmp1, tmp2 = False, False
            if i > 0 and s1[i - 1] == s3[i - 1 + j]:
                tmp1 = dfs(i - 1, j)
            if j > 0 and s2[j - 1] == s3[i - 1 + j]:
                tmp2 = dfs(i, j - 1)
            return tmp1 or tmp2

        return dfs(n1, n2)
```

```python
class Solution1:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
        # 2D Bottom Up DP, time O(n1*n2), space O(n1*n2)
        n1, n2, n3 = len(s1), len(s2), len(s3)

        if s1 == "": return s2 == s3
        if s2 == "": return s1 == s3
        if n1 + n2 != n3: return False

        dp = [[0] * (n2 + 1) for _ in range(n1 + 1)]
        dp[0][0] = 1

        for i in range(1, n1 + 1):
            dp[i][0] = dp[i - 1][0] and s1[i - 1] == s3[i - 1]
        for j in range(1, n2 + 1):
            dp[0][j] = dp[0][j - 1] and s2[j - 1] == s3[j - 1]

        for i in range(1, n1 + 1):
            for j in range(1, n2 + 1):
                dp[i][j] = (s1[i - 1] == s3[i - 1 + j] and dp[i - 1][j]) \
                        or (s2[j - 1] == s3[i - 1 + j] and dp[i][j - 1])

        return dp[-1][-1]
```

115. Distinct Subsequences

```python
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        # dp[i][j]: the number of distinct subsequences for s[0,i-1] and t[0,j-1];
        @lru_cache(None)
        def dfs(i, j):
            if j == 0:
                return 1
            if i == 0:
                return 0

            if s[i - 1] == t[j - 1]:
                # given the match, t can choose or not choose
                res = dfs(i - 1, j - 1) + dfs(i - 1, j)
            else:
                res = dfs(i - 1, j)

            return res

        return dfs(len(s), len(t))
```

Given two strings `s` and `t`, return *the number of distinct subsequences of* `s` *which equals* `t`.

Input: s = "rabbbit", t = "rabbit"

Output: 3

Explanation:

As shown below, there are 3 ways you can generate "rabbit" from S.

rabbbit

rabbbit

rabbbit

583. Delete Operation for Two Strings

Given two words *word1* and *word2*, find the minimum number of steps required to make *word1* and *word2* the same, where in each step you can delete one character in either string.

Input: "sea", "eat"

Output: 2

Explanation: You need one step to make "sea" to "ea" and another step to make "eat" to "ea".

```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        # soln 1
        m, n = len(word1), len(word2)
        @lru_cache(None)
        def dfs(i, j):
            if i == m and j == n:
                return 0
            if i == m or j == n:
                return n - j or m - i
            if word1[i] == word2[j]:
                return dfs(i + 1, j + 1)
            else:
                return min(dfs(i + 1, j), dfs(i, j + 1)) + 1

        return dfs(0, 0)

        # soln 2
        @lru_cache(None)
        def dfs(i, j):
            if i == 0 and j == 0:
                return 0
            if i == 0 or j == 0:
                return j or i

            return dfs(i - 1, j - 1) if word1[i - 1] == word2[j - 1] \
                    else min(dfs(i, j - 1), dfs(i - 1, j)) + 1

        return dfs(len(word1), len(word2))

        # bottom up
        m, n = len(word1), len(word2)
        dp = [[0] * (n + 1) for i in range(m + 1)]
        for i in range(m):
            for j in range(n):
                dp[i + 1][j + 1] = max(dp[i][j + 1], dp[i + 1][j], \
                                       dp[i][j] + (word1[i] == word2[j]))
        return m + n - 2 * dp[m][n]
```

712. Minimum ASCII Delete Sum for Two Strings

```python
class Solution2:
    def minimumDeleteSum(self, s1, s2):
        """
        :type s1: str
        :type s2: str
        :rtype: int
        """
        # Top Down 2D DP, time O(n1*n2), space O(n1*n2)
        l1, l2 = [ord(c) for c in s1], [ord(c) for c in s2]

        @lru_cache(None)
        def dfs(i, j):
            if i == 0 and j == 0: return 0
            if i == 0: return dfs(i, j - 1) + l2[j - 1]
            if j == 0: return dfs(i - 1, j) + l1[i - 1]

            if l1[i - 1] == l2[j - 1]:
                return dfs(i - 1, j - 1)
            else:
                return min(dfs(i - 1, j) + l1[i - 1], dfs(i, j - 1) + l2[j - 1])

        return dfs(len(s1), len(s2))
```

Given two strings `s1, s2`, find the lowest ASCII sum of deleted characters to make two strings equal.

`Input: s1 = "sea", s2 = "eat"`

`Output: 231`

`Explanation: Deleting "s" from "sea" adds the ASCII value of "s" (115) to the sum.`

`Deleting "t" from "eat" adds 116 to the sum.`

`At the end, both strings are equal, and 115 + 116 = 231 is the minimum sum possible to achieve this.`

1035. Uncrossed Lines

```python
class Solution:
    def maxUncrossedLines(self, A: List[int], B: List[int]) -> int:
        m, n = len(A), len(B)

        @lru_cache(None)
        def dfs(i, j):
            if i == 0 or j == 0: return 0

            if A[i - 1] == B[j - 1]:
                return dfs(i - 1, j - 1) + 1
            else:
                return max(dfs(i - 1, j), dfs(i, j - 1))

        return dfs(m, n)
```

Now, we may draw *connecting lines*: a straight line connecting two numbers `A[i]` and `B[j]` such that:

- `A[i] == B[j]`;
- The line we draw does not intersect any other connecting (non-horizontal) line.

`Input: A = [1,4,2], B = [1,2,4]`

`Output: 2`

`Explanation: We can draw 2 uncrossed lines as in the diagram.`

`We cannot draw 3 uncrossed lines, because the line from A[1]=4 to B[2]=4 will intersect the line from A[2]=2 to B[1]=2.`

1092. Shortest Common Supersequence

```python
class Solution(object):
    def shortestCommonSupersequence(self, A, B):
        """
        :type str1: str
        :type str2: str
        :rtype: str
        """
        def lcs(A, B):
            n, m = len(A), len(B)
            dp = [["" for _ in range(m + 1)] for _ in range(n + 1)]
            for i in range(n):
                for j in range(m):
                    if A[i] == B[j]:
                        dp[i + 1][j + 1] = dp[i][j] + A[i]
                    else:
                        dp[i + 1][j + 1] = max(dp[i + 1][j], dp[i][j + 1], key=len)
            return dp[-1][-1]

        # print(lcs(A, B))
        res, i, j = "", 0, 0
        for c in lcs(A, B):
            while A[i] != c:
                res += A[i]
                i += 1
            while B[j] != c:
                res += B[j]
                j += 1
            res += c
            i, j = i + 1, j + 1
        return res + A[i:] + B[j:]
```

Given two strings `str1` and `str2`, return the shortest string that has both `str1` and `str2` as subsequences. If multiple answers exist, you may return any of them.

`Input: str1 = "abac", str2 = "cab"`

`Output: "cabac"`

Top Down TLE ;(

1216. Valid Palindrome III

```python
class Solution:
    def isValidPalindrome(self, s: str, k: int) -> bool:
        @lru_cache(None)
        def dfs(i, j):
            if i >= j:
                return 0
            if s[i] == s[j]:
                return dfs(i + 1, j - 1)
            else:
                return min(dfs(i + 1, j), dfs(i, j - 1)) + 1

        return dfs(0, len(s) - 1) <= k
```

Given a string `s` and an integer `k`, find out if the given string is a *K-Palindrome* or not.

A string is K-Palindrome if it can be transformed into a palindrome by removing at most `k` characters from it.

`Input: s = "abcdeca", k = 2`

`Output: true`

`Explanation: Remove 'b' and 'e' characters.`

1312. Minimum Insertion Steps to Make a String Palindrome

```python
class Solution:
    def minInsertions(self, s: str) -> int:
        # Top down DP, time O(n), space O(n)
        @lru_cache(None)
        def dfs(i, j):
            if i >= j: return 0
            # if i == j + 1: return s[i] == s[j]
            return dfs(i + 1, j - 1) if s[i] == s[j] else min(dfs(i, j - 1), dfs(i + 1, j)) + 1

        return dfs(0, len(s) - 1)

        # Bottom up DP, time O(n^2), space O(n^2)
        n = len(s)
        dp = [[0] * n for _ in range(n)]
        for j in range(n):
            for i in range(j - 1, -1, -1):
                dp[i][j] = dp[i + 1][j - 1] if s[i] == s[j] else min(dp[i + 1][j], dp[i][j - 1]) + 1
        return dp[0][n - 1]
```

Given a string `s`. In one step you can insert any character at any index of the string.

Return *the minimum number of steps* to make `s` palindrome.

A Palindrome String is one that reads the same backward as well as forward.

1458. Max Dot Product of Two Subsequences

```python
class Solution:
    def maxDotProduct(self, nums1: List[int], nums2: List[int]) -> int:
        n1, n2 = len(nums1), len(nums2)
        # dont init with 0 because nums could be negative
        dp = [[float('-inf')] * (n2 + 1) for _ in range(n1 + 1)]

        for i in range(1, n1 + 1):
            for j in range(1, n2 + 1):
                prod = nums1[i - 1] * nums2[j - 1]
                # 4 cases, ignore nums1, ignore nums2, consider both, ignore neither
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1] + prod, prod)

        return dp[-1][-1]

class Solution2:
    def maxDotProduct(self, nums1: List[int], nums2: List[int]) -> int:
        @lru_cache(None)
        def dfs(i, j):
            if i == 0 or j == 0: return float('-inf')
            prod = nums1[i - 1] * nums2[j - 1]
            return max(dfs(i - 1, j), dfs(i, j - 1), dfs(i - 1, j - 1) + prod, prod)

        return dfs(len(nums1), len(nums2))
```

Given two arrays `nums1` and `nums2`.

Return the maximum dot product between non-empty subsequences of nums1 and nums2 with the same length.

`Input: nums1 = [2,1,-2,5], nums2 = [3,0,-6]`

`Output: 18`

`Explanation: Take subsequence [2,-2] from nums1 and subsequence [3,-6] from nums2.`

`Their dot product is (2*3 + (-2)*(-6)) = 18.`

1771. Maximize Palindrome Length From Subsequences

```python
class Solution:
    def longestPalindrome(self, word1: str, word2: str) -> int:
        # Top down, O((M+N)^2)/O((M+N)^2)
        s = word1 + word2
        m, n, self.res = len(word1), len(word2), 0

        @lru_cache(None)
        def dfs(i, j):
            if i > j: return 0
            if i == j: return 1

            if s[i] == s[j]:
                tmp = dfs(i+1, j-1) + 2
                if i < m and j >= m:
                        self.res = max(self.res, tmp)
                return tmp
            else:
                return max(dfs(i, j-1), dfs(i+1, j))
        dfs(0, m + n - 1)
        return self.res

        # DP bottom up, O((M+N)^2)
        s = word1 + word2
        m, n, res = len(word1), len(word2), 0

        dp = [[0] * (m + n) for _ in range(m + n)]
        for j in range(m + n):
            dp[j][j] = 1
            for i in range(j - 1, -1, -1):
                if s[i] == s[j]:
                    dp[i][j] = 2 if i + 1 == j else dp[i+1][j-1] + 2
                    # if the palindrome includes both string consider it as a valid
                    if i < m and j >= m:
                        res = max(res, dp[i][j])
                else:
                    dp[i][j] = max(dp[i+1][j], dp[i][j-1])

        return res
```

You are given two strings, `word1` and `word2`. You want to construct a string in the following manner:

- Choose some non-empty subsequence `subsequence1` from `word1`.
- Choose some non-empty subsequence `subsequence2` from `word2`.
- Concatenate the subsequences: `subsequence1 + subsequence2`, to make the string.