DFS collection

Derrick Guo

February 2020

1 5. Longest Palindromic Substring

Given a string s, find the longest palindromic substring in s. You may assume that the maximum length of s is 1000.

Example 1:

Input: "babad" Output: "bab" Note: "aba" is also a valid answer. Example 2:

Input: "cbbd" Output: "bb"

Idea:

We can use dp here, but there is another way that can also achieve n^2 time but only constant space.

First, let's see how the dp solution work. Usually in this kind of substring or subsequences problem, we need a 2d dp array, where one demension indicating the starting index and the other one indicating the ending index. The initial value of the dp array is all False because there is no way we can say anything is palindromic before looking at the actual string.

There are 3 cases in the process:

1. l == r, which means the substring only contains 1 character. In this case dp[l][r] is True.
2. r == l+1, which means the substring has 2 characters in it, then the value of dp[l][r] depends on if s[l] == s[r]
3. else case. The substring now has more than 3 characters in it, and the value of dp[l][r] not only depends on s[l] == s[r], but also dp[l+1][r-1], which is indicating if s[l+1, r-1] is a substring. If s[l+1, r-1] is a substring and s[l] == s[r], we can say s[l,r] is also a substring. (here both l, r are included in the substring for simplicity)

Then, our process of building up the dp array needs to build dp[l+1][r-1] before getting to dp[l][r], which means we can't do a regular for loop like **for l in range(len(s)): for r in range(l, len(s))**. **we need to move l backwards from r, which allows us to have dp[l+1][r] computed before dp[l][r]**.

```
1 class Solution(object):
2     def longestPalindrome(self, s):
3         """
```

```
        :type s: str
        :rtype: str
        """
        lmax, rmax = 0, 0
        d = [[False] * len(s) for _ in range(len(s))]
        for r in range(len(s)):
            l = r
            while l >= 0:
                if l == r:
                    d[l][r] = True
                elif r == l + 1:
                    d[l][r] = s[l] == s[r]
                elif d[l+1][r-1] and s[l] == s[r]:
                    d[l][r] = True
                if d[l][r] and r - l > rmax - lmax:
                    lmax = l
                    rmax = r
                l -= 1
        return s[lmax: rmax+1]
```

Another way of solving this is to expand the substring from middle directly, without using the dynamic array. This saves the n^2 space with the 2d array. The idea is to get the longest palindromic substring with s[i] as the middle character, for all i in s. Then return the longest one. There are 2 cases since the length of the longest palindromic substring could be even or odd, so we calculate both and take the max.

```
class Solution(object):
    def longestPalindrome(self, s):
        """
        :type s: str
        :rtype: str
        """
        n = len(s)

        def getLen(s, l, r):
            while l >= 0 and r < n and s[l] == s[r]:
                l -= 1
                r += 1
            return r - l - 1

        start = 0
        maxlen = 0
        for i in range(n):
            curlen = max(getLen(s, i, i), getLen(s, i, i+1))
            if curlen > maxlen:
```

```
                start = i - (curlen-1)//2
                maxlen = curlen
        return s[start: start+maxlen]
```

2 72. Edit Distance

Given two words word1 and word2, find the minimum number of operations required to convert word1 to word2.

You have the following 3 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')

idea:

This question has 2 input strings, and we need to do analysis on their substrings. Then we can use a 2d dynamic array, one dimension representing index (or substring) of word1 and the other dimension representing index (or substring) of word2.

Let d[i][j] represents the min # of operations it takes for word1[0,i] to become word2[0,j]

There are 4 cases.

1. word1 is empty. Then it takes len(word2) operations (addtions) for word1 to become word2. So for each substring of word2, word2[0,j], we need j addtions.

2. word2 is empty. Then it takes len(word1) deletions for word1 to become word2. So for each substring of word1, word1[0, i], we need i deletions to get to word2.

3. word1[i] == word2[j]. There is no edit on this step, the number of operation it takes is exactly the same as d[i-1][j-1].

4. word1[i] != word2[j]. Then we try to solve this with what we've seen. we can delete a char from word1[0, i], or we can insert a char to word[0, i], which is essentailly the same as deleting the last char from word2[0, j], or we can replace word1[i] with word2[j].

```
class Solution(object):
    def minDistance(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """
```

```
        dp = [[0] * (len(word2)+1) for _ in range(len(word1) + 1)]

        for i in range(len(word1)+1):
            dp[i][0] = i

        for j in range(len(word2)+1):
            dp[0][j] = j

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

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

3 1143. Longest Common Subsequence

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

A subsequence of a string is a new string generated from the original string with some characters(can be none) deleted without changing the relative order of the remaining characters. (eg, "ace" is a subsequence of "abcde" while "aec" is not). A common subsequence of two strings is a subsequence that is common to both strings.

If there is no common subsequence, return 0.

Example 1:

Input: text1 = "abcde", text2 = "ace" Output: 3 Explanation: The longest common subsequence is "ace" and its length is 3. Example 2:

Input: text1 = "abc", text2 = "abc" Output: 3 Explanation: The longest common subsequence is "abc" and its length is 3. Example 3:

Input: text1 = "abc", text2 = "def" Output: 0 Explanation: There is no such common subsequence, so the result is 0.

idea:

We have 2 strings. So as usual we need 2d array, with each dimension representing one input string. The first row and col are all 0s, since empty string has a longest common subsequence of length 0 with any other strings.

Then, we start constructing our dynamic array. There are 2 cases when we have a pair of word1[i], word2[j].

1. word1[i] == word2[j], then this char can be in the longest subsequence, but not guaranteed. There could be another subsequence that are longer but doesn't have this char in it. So in this case, dp[i][j] = max(dp[i-1][j-1] + 1, dp[i-1][j], dp[i][j-1])

2. word1[i] != word2[j]. Then this pair doesn't exist in the longest common subsequence because they don't match. Then dp[i][j] = max(dp[i-1][j-1], dp[i-1][j], dp[i][j-1])

```
class Solution(object):
    def longestCommonSubsequence(self, text1, text2):
        """
        :type text1: str
        :type text2: str
        :rtype: int
        """
        m = len(text1)
        n = len(text2)
        dp = [[0 for _ in range(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] = max(dp[i-1][j-1] + 1, dp[i-1][j], dp[i][j-1])
                else:
                    dp[i][j] = max(dp[i-1][j-1], dp[i-1][j], dp[i][j-1])
        return dp[-1][-1]
```

4 678. Valid Parenthesis String

Given a string containing only three types of characters: '(', ')' and '*', write a function to check whether this string is valid. We define the validity of a string by these rules:

Any left parenthesis '(' must have a corresponding right parenthesis ')'. Any right parenthesis ')' must have a corresponding left parenthesis '('. Left parenthesis '(' must go before the corresponding right parenthesis ')'. '*' could be treated as a single right parenthesis ')' or a single left parenthesis '(' or an empty string. An empty string is also valid. Example 1: Input: "()" Output: True Example 2: Input: "(*)" Output: True Example 3: Input: "(*))" Output: True

idea:

This can be done by DP, but really complicated. There are 2 alternative way to solve this.

1. use 2 stacks. One for looping forwards to check if there are enough opening brackets to match all the closing brackets, the other one for looping backwards to check if there are enough closing brackets to match all the opening brackets.

```
class Solution(object):
    def checkValidString(self, s):
        """
        :type s: str
```

```
        :rtype: bool
        """
        stack = []
        for c in s:
            if c == '(' or c == '*':
                stack.append(c)
            else:
                if stack:
                    stack.pop()
                else:
                    return False
        stack = []
        for c in s[::-1]:
            if c == ')' or c == '*':
                stack.append(c)
            else:
                if stack:
                    stack.pop()
                else:
                    return False
        return True
```

The second solution is similar, but use 2 counters. one for counting the number of opening brackets, and one for counting the sum of opening brackets and ''. Increase the first counter if we see a '(' and decrease it otherwise. Similarly, increase the second counter when we don't see a ')' and decrease it otherwise. Then if the second counter < 0, we know we have too many closing brackets, return false. And return if the first counter $== 0$ at the end to check if we have too many opening brackets.

```
class Solution(object):
    def checkValidString(self, s):
        """
        :type s: str
        :rtype: bool
        """
        min_op = 0
        max_op = 0
        for c in s:
            if c == '(': min_op += 1
            else: min_op -= 1
            if c != ')': max_op += 1
            else: max_op -= 1
            if max_op < 0: return False
            min_op = max(0, min_op)
        return min_op == 0
```

5 Longest Increasing Subsequence

Given an unsorted array of integers, find the length of longest increasing subsequence.

Example:

Input: [10,9,2,5,3,7,101,18] Output: 4 Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4. Note:

There may be more than one LIS combination, it is only necessary for you to return the length. Your algorithm should run in O(n2) complexity.

idea:

The idea is to let d[i] represent the length of the longest increasing subsequence ending at nums[i], inclusively. Then when I keep build up my longest increasing subsequence, I can compare nums[j] and nums[i], j > i, because I have the knowledge that nums[i] is in the longest increasing subsequence when I check d[i]. Then we just need to build the dp array and find the max of it. Not the last one we want, because the last number in the array can be small, which means the longest increasing subsequence ending at the last number can be short. Thus we want the max.

Note: If a question wants us to find the max length of something, if we assume dp[i] will include nums[i], set the initial value of the dp array to 1.

```
class Solution(object):
    def lengthOfLIS(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if not nums: return 0
        dp = [1 for _ in range(len(nums))]
        for i in range(1, len(nums)):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j] + 1)

        return max(dp)
```

6 322. Coin Change

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

Example 1:

Input: coins = [1, 2, 5], amount = 11 Output: 3 Explanation: 11 = 5 + 5 + 1 Example 2:

Input: coins = [2], amount = 3 Output: -1

Note: You may assume that you have an infinite number of each kind of coin.

idea:

To find the minimum of coins used to make up number k, we set d[i] to be the minimum number of coins used to make up number i. Then we simply return d[k]. To determine the value of d[i], we can use information we gained while building d. We try to use each coin from d[i - coin], and find the minimum number of coins from there.

Note: It makes it easier to sort the coins first, so that we can skip large coins when there is no way to use it (i < coin value)

```
class Solution(object):
    def coinChange(self, coins, amount):
        """
        :type coins: List[int]
        :type amount: int
        :rtype: int
        """
        d = [amount + 1 for _ in range(amount+1)]
        d[0] = 0
        coins.sort()
        for i in range(1, amount+1):
            for c in coins:
                if c > i: break
                d[i] = min(d[i], d[i-c]+1)
        return d[-1] if d[-1] < amount + 1 else -1
```