# Longest Common Subsequence

Sunday, November 7, 2021 2:37 PM

Q1: What is the difference between `[[0] * m * n]` and `[[0] * m for _ in range(n)]`? Why does the former update all the rows of that column when I try to update one particular cell ?
A1: `[[0] * m * n]` creates `n` references to the exactly same list objet: `[0] * m`; In contrast: `[[0] * m for _ in range(n)]` creates `n` different list objects that have same value of `[0] * m`.
**End of Q & A**

Please refer to my solution Java/Python 3 2 Clean DP codes of O(m * n) & O(min(m, n)) space w/ breif explanation and analysis of a similar problem: 1458. Max Dot Product of Two Subsequences
More similar LCS problems:
1092. Shortest Common Supersequence and Solution
1062. Longest Repeating Substring (Premium).
516. Longest Palindromic Subsequence
1312. Minimum Insertion Steps to Make a String Palindrome

Find LCS;
Let `X` be `"XMJYAUZ"` and `Y` be `"MZJAWXU"`. The longest common subsequence between `X` and `Y` is `"MJAU"`. The following table shows the lengths of the longest common subsequences between prefixes of `X` and `Y`. The `ith` row and `jth` column shows the length of the LCS between `X_{1..i}` and `Y_{1..j}`.

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| | | Ø | M | Z | J | A | W | X | U |
| 0 | Ø | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | X | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | M | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | J | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| 4 | Y | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| 5 | A | 0 | 1 | 1 | 2 | 3 | 3 | 3 | 3 |
| 6 | U | 0 | 1 | 1 | 2 | 3 | 3 | 3 | 4 |
| 7 | Z | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 |

you can refer to here for more details.
**Method 1:**

```
publicintlongestCommonSubsequence(String s1, String s2){
    int[][] dp = newint[s1.length() + 1][s2.length() + 1];
    for(inti = 0; i < s1.length(); ++i)
        for(intj = 0; j < s2.length(); ++j)
            if(s1.charAt(i) == s2.charAt(j)) dp[i + 1][j + 1] = 1+ dp[i][j];
            elsedp[i + 1][j + 1] =  Math.max(dp[i][j + 1], dp[i + 1][j]);
    returndp[s1.length()][s2.length()];
}
deflongestCommonSubsequence(self, text1: str, text2: str) -> int:dp = [[0] * (len(text2) + 1) for_ inrange(len(text1) + 1)]
    fori, c inenumerate(text1):
        forj, d inenumerate(text2):
            dp[i + 1][j + 1] = 1+ dp[i][j] ifc == d elsemax(dp[i][j + 1], dp[i + 1][j])
    returndp[-1][-1]
```

**Analysis:**
Time & space: O(m * n)

**Method 2:**

***Space Optimization***

Obviously, the code in method 1 only needs information of previous row to update current row. So we just use a **two-row** 2D array to save and update the matching results for chars in `s1` and `s2`.

Note: use `k ^ 1` and `k ^= 1` to switch between `dp[0] (row 0)` and `dp[1] (row 1)`.

```
publicintlongestCommonSubsequence(String s1, String s2){
    intm = s1.length(), n = s2.length();
    if(m < n)  returnlongestCommonSubsequence(s2, s1);
    int[][] dp = newint[2][n + 1];
    for(inti = 0, k = 1; i < m; ++i, k ^= 1)
        for(intj = 0; j < n; ++j)
            if(s1.charAt(i) == s2.charAt(j)) dp[k][j + 1] = 1+ dp[k ^ 1][j];
            elsedp[k][j + 1] = Math.max(dp[k ^ 1][j + 1], dp[k][j]);
    returndp[m % 2][n];
}
```

Note: use `1 - i % 2` and `i % 2` to switch between `dp[0] (row 0)` and `dp[1] (row 1)`.

```
deflongestCommonSubsequence(self, text1: str, text2: str) -> int:m, n = len(text1), len(text2)
    ifm < n:
        returnself.longestCommonSubsequence(text2, text1)
    dp = [[0] * (n + 1) for_ inrange(2)]
    fori, c inenumerate(text1):
        forj, d inenumerate(text2):
            dp[1- i % 2][j + 1] = 1+ dp[i % 2][j] ifc == d elsemax(dp[i % 2][j + 1], dp[1- i % 2][j])
    returndp[m % 2][-1]
```

***Further Space Optimization to save half space*** - credit to **@survive and @lenchen1112**.

Obviously, the above code in method 2 only needs information of previous and current columns of previous row to update current row. So we just use a **1-row** 1D array and `2` variables to save and update the matching results for chars in `text1` and `text2`.

```
publicintlongestCommonSubsequence(String text1, String text2){
    intm = text1.length(), n = text2.length();
    if(m < n) {
        returnlongestCommonSubsequence(text2, text1);
    }
    int[] dp = newint[n + 1];
    for(inti = 0; i < text1.length(); ++i) {
        for(intj = 0, prevRow = 0, prevRowPrevCol = 0; j < text2.length(); ++j) {
            prevRowPrevCol = prevRow;
            prevRow = dp[j + 1];
            dp[j + 1] = text1.charAt(i) == text2.charAt(j) ? prevRowPrevCol + 1: Math.max(dp[j], prevRow);
        }
    }
    returndp[n];
}
deflongestCommonSubsequence(self, text1: str, text2: str) -> int:m, n = map(len, (text1, text2))
    ifm < n:
        returnself.longestCommonSubsequence(text2, text1)
    dp = [0] * (n + 1)
    forc intext1:
        prevRow, prevRowPrevCol = 0, 0forj, d inenumerate(text2):
            prevRow, prevRowPrevCol = dp[j + 1], prevRow
            dp[j + 1] = prevRowPrevCol + 1ifc == d elsemax(dp[j], prevRow)
    returndp[-1]
```

**Analysis:**

Time: O(m * n). space: O(min(m, n)).

From <https://leetcode.com/problems/longest-common-subsequence/discuss/351689/JavaPython-3-Two-DP-codes-of-O(mn)-and-O(min(m-n))-spaces-w-picture-and-analysis>

C++ with Pictures

# Intuition

LCS is a well-known problem, and there are similar problems here:

- 1092. Shortest Common Supersequence
- 1062. Longest Repeating Substring
- 516. Longest Palindromic Subsequence

Bottom-up DP utilizes a matrix `m` where we track LCS sizes for each combination of `i` and `j`.

- If `a[i] == b[j]`, LCS for i and j would be `1` plus LCS till the i-1 and j-1 indexes.
- Otherwise, we will take the largest LCS if we skip a charracter from one of the string (`max(m[i - 1][j], m[i][j - 1])`).

This picture shows the populated matrix for `"xabccde", "ace"` test case.

| | | x | a | b | c | c | d | e |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| c | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 2 |
| e | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 |

# Solution

```
intlongestCommonSubsequence(string&a, string&b){
 vector<vector<short>> m(a.size() + 1, vector<short>(b.size() + 1));
 for(autoi = 1; i <= a.size(); ++i)
  for(autoj = 1; j <= b.size(); ++j)
   if(a[i - 1] == b[j - 1]) m[i][j] = m[i - 1][j - 1] + 1;
   elsem[i][j] = max(m[i - 1][j], m[i][j - 1]);
```

```
 returnm[a.size()][b.size()];
}
```

## Complexity Analysis

Runtime: O(nm), where n and m are the string sizes.
Memory: O(nm).

# Memory Optimization

You may notice that we are only looking one row up in the solution above. So, we just need to store two rows.

```
intlongestCommonSubsequence(string&a, string&b){
 if(a.size() < b.size()) returnlongestCommonSubsequence(b, a);
 vector<vector<short>> m(2, vector<short>(b.size() + 1));
 for(autoi = 1; i <= a.size(); ++i)
  for(autoj = 1; j <= b.size(); ++j)
   if(a[i - 1] == b[j - 1]) m[i % 2][j] = m[(i -1) % 2][j - 1] + 1;
   elsem[i % 2][j] = max(m[(i - 1) % 2][j], m[i % 2][j - 1]);
 returnm[a.size() % 2][b.size()];
}
```

## Complexity Analysis

Runtime: O(nm), where n and m are the string sizes.
Memory: O(min(n,m)).

From <https://leetcode.com/problems/longest-common-subsequence/discuss/348884/C%2B%2B-with-picture-O(nm)>

## Step -Step By approach

**Why might we want to solve the longest common subsequence problem?**
File comparison. The Unix program "diff" is used to compare two different versions of the same file, to determine what changes have been made to the file. It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed. In this instance of the problem we should think of each line of a file as being a single complicated character in a string.

## Solution

1. Recursive solution

```
classSolution:deflongestCommonSubsequence(self, s1: str, s2: str) -> int:returnself.helper(s1, s2, 0, 0)

    defhelper(self, s1, s2, i, j):ifi == len(s1) orj == len(s2):
            return0ifs1[i] == s2[j]:
            return1+ self.helper(s1, s2, i + 1, j + 1)
        else:
            returnmax(self.helper(s1, s2, i+1, j), self.helper(s1, s2, i, j + 1))
```

If the two strings have no matching characters, so the last line always gets executed, the the time bounds are binomial coefficients, which (if m=n) are close to 2^n.

```
                                                    lcs("AXYT", "AYZX")
```

```
                /          \
       lcs("AXY", "AYZX")         lcs("AXYT", "AYZ")
       /      \                /          \
lcs("AX", "AYZX") lcs("AXY", "AYZ")   lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

2. Recursive solution with Memoization

```
classSolution:deflongestCommonSubsequence(self, s1: str, s2: str) -> int:m = len(s1)
    n = len(s2)
    memo = [[-1for_ inrange(n + 1)] for_ inrange(m + 1)]
    returnself.helper(s1, s2, 0, 0, memo)

  defhelper(self, s1, s2, i, j, memo):ifmemo[i][j] < 0:
      ifi == len(s1) orj == len(s2):
        memo[i][j] = 0elifs1[i] == s2[j]:
        memo[i][j] = 1+ self.helper(s1, s2, i + 1, j + 1, memo)
      else:
        memo[i][j] = max(
          self.helper(s1, s2, i + 1, j, memo),
          self.helper(s1, s2, i, j + 1, memo),
        )
    returnmemo[i][j]
```

Time analysis: each call to subproblem takes constant time. We call it once from the main routine, and at most twice every time we fill in an entry of array L. There are (m+1)(n+1) entries, so the total number of calls is at most 2(m+1)(n+1)+1 and the time is O(mn).

As usual, this is a worst case analysis. The time might sometimes better, if not all array entries get filled out. For instance if the two strings match exactly, we'll only fill in diagonal entries and the algorithm will be fast.

3. Bottom up dynamic programming

We can view the code above as just being a slightly smarter way of doing the original recursive algorithm, saving work by not repeating subproblem computations. But it can also be thought of as a way of computing the entries in the array L. The recursive algorithm controls what order we fill them in, but we'd get the same results if we filled them in in some other order. We might as well use something simpler, like a nested loop, that visits the array systematically. The only thing we have to worry about is that when we fill in a cell L[i,j], we need to already know the values it depends on, namely in this case L[i+1,j], L[i,j+1], and L[i+1,j+1]. For this reason we'll traverse the array backwards, from the last row working up to the first and from the last column working up to the first.

```
classSolution:deflongestCommonSubsequence(self, s1: str, s2: str) -> int:m = len(s1)
    n = len(s2)
    memo = [[0for_ inrange(n + 1)] for_ inrange(m + 1)]

    forrow inrange(1, m + 1):
      forcol inrange(1, n + 1):
        ifs1[row - 1] == s2[col - 1]:
          memo[row][col] = 1+ memo[row - 1][col - 1]
        else:
          memo[row][col] = max(memo[row][col - 1], memo[row - 1][col])

    returnmemo[m][n]
```

Advantages of this method include the fact that iteration is usually faster than recursion, we save three if statements per iteration since we don't need to test whether L[i,j], L[i+1,j], and L[i,j+1] have already been computed (we know in advance that the answers will be no, yes, and yes). One disadvantage over memoizing is that this fills in the entire array even when it might be possible to solve the problem by looking at only a fraction of the array's cells.

Time complexity: O(mn) and Space complexity: O(mn)

4. Reduced space complexity

```
classSolution:deflongestCommonSubsequence(self, s1: str, s2: str) -> int:m = len(s1)
    n = len(s2)
    ifm < n:
      returnself.longestCommonSubsequence(s2, s1)
    memo = [[0for_ inrange(n + 1)] for_ inrange(2)]
```

```
fori inrange(m):
        forj inrange(n):
            ifs1[i] == s2[j]:
                memo[1- i % 2][j + 1] = 1+ memo[i % 2][j]
            else:
                memo[1- i % 2][j + 1] = max(memo[1- i % 2][j], memo[i % 2][j + 1])
returnmemo[m % 2][n]
```

Time complexity: O(mn) ans Space comlexity: O(min(m, n))

From <https://leetcode.com/problems/longest-common-subsequence/discuss/436719/Python-very-detailed-solution-with-explanation-and-walkthrough-step-by-step.>

# Brute Force

```
classSolution{
    public int longestCommonSubsequence(Stringtext1, Stringtext2){
        returnlongestCommonSubsequence(text1, text2, 0, 0);
    }

    private int longestCommonSubsequence(Stringtext1, Stringtext2, int i, int j){
        if(i == text1.length() || j == text2.length())
            return0;
        if(text1.charAt(i) == text2.charAt(j))
            return1+ longestCommonSubsequence(text1, text2, i + 1, j + 1);
        elsereturnMath.max(
                longestCommonSubsequence(text1, text2, i + 1, j),
                longestCommonSubsequence(text1, text2, i, j + 1)
            );
    }
}
```

# Top-down DP

We might use memoization to overcome overlapping subproblems.
Since there are two changing values, i.e. `i` and `j` in the recursive function `longestCommonSubsequence`, we might apply a two-dimensional array.

```
classSolution{
    private Integer[][] dp;
    public int longestCommonSubsequence(Stringtext1, Stringtext2){
        dp = newInteger[text1.length()][text2.length()];
        returnlongestCommonSubsequence(text1, text2, 0, 0);
    }

    private int longestCommonSubsequence(Stringtext1, Stringtext2, int i, int j){
        if(i == text1.length() || j == text2.length())
            return0;

        if(dp[i][j] != null)
            returndp[i][j];
```

```
    if(text1.charAt(i) == text2.charAt(j))
      returndp[i][j] = 1+ longestCommonSubsequence(text1, text2, i + 1, j + 1);
    elsereturndp[i][j] = Math.max(
        longestCommonSubsequence(text1, text2, i + 1, j),
        longestCommonSubsequence(text1, text2, i, j + 1)
      );
  }
}
```

# Bottom-up DP

For every `i` in text1, `j` in text2, we will choose one of the following two options:

- if two characters match, length of the common subsequence would be 1 plus the length of the common subsequence till the `i-1` and`j-1` indexes
- if two characters doesn't match, we will take the longer by either skipping `i` or `j` indexes

```
classSolution{
  public int longestCommonSubsequence(Stringtext1, Stringtext2){
    int[][] dp = newint[text1.length() + 1][text2.length() + 1];
    for(int i = 1; i <= text1.length(); i++) {
      for(int j = 1; j <= text2.length(); j++) {
        if(text1.charAt(i - 1) == text2.charAt(j - 1))
          dp[i][j] = 1+ dp[i - 1][j - 1];
        elsedp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
    }
    returndp[text1.length()][text2.length()];
  }
}
```

From <https://leetcode.com/problems/longest-common-subsequence/discuss/590781/From-Brute-Force-To-DP>