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

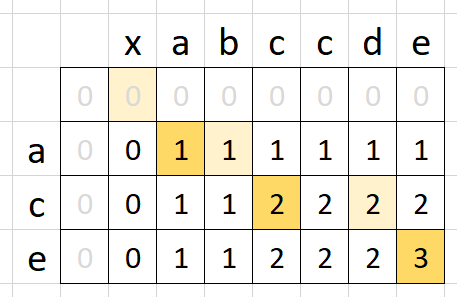
Intuition

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

* [1092. Shortest Common Supersequence](https://leetcode.com/problems/shortest-common-supersequence/)
* [1062. Longest Repeating Substring](https://leetcode.com/problems/longest-repeating-substring/)
* [516. Longest Palindromic Subsequence](https://leetcode.com/problems/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.  


Solution

int longestCommonSubsequence(string &a, string &b) {

vector<vector<short>> m(a.size() + 1, vector<short>(b.size() + 1));

for (auto i = 1; i <= a.size(); ++i)

for (auto j = 1; j <= b.size(); ++j)

if (a[i - 1] == b[j - 1]) m[i][j] = m[i - 1][j - 1] + 1;

else m[i][j] = max(m[i - 1][j], m[i][j - 1]);

return m[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.

int longestCommonSubsequence(string &a, string &b) {

if (a.size() < b.size()) return longestCommonSubsequence(b, a);

vector<vector<short>> m(2, vector<short>(b.size() + 1));

for (auto i = 1; i <= a.size(); ++i)

for (auto j = 1; j <= b.size(); ++j)

if (a[i - 1] == b[j - 1]) m[i % 2][j] = m[(i -1) % 2][j - 1] + 1;

else m[i % 2][j] = max(m[(i - 1) % 2][j], m[i % 2][j - 1]);

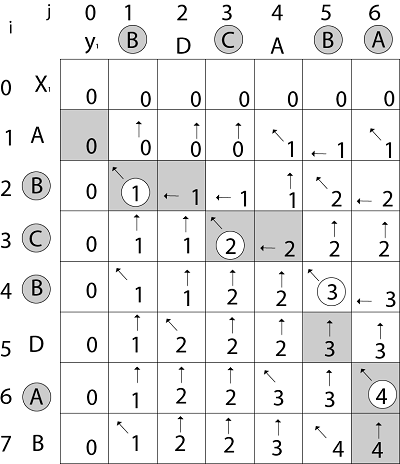
return m[a.size() % 2][b.size()];

}

Complexity Analysis

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

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

For LCS DP computation, e.g, s1 = "XABCBDAB", s2 = "yBDCABA", please refer to the following picture:  


**Method 1:**

public int longestCommonSubsequence(String s1, String s2) {

int[][] dp = new int[s1.length() + 1][s2.length() + 1];

for (int i = 0; i < s1.length(); ++i)

for (int j = 0; j < s2.length(); ++j)

if (s1.charAt(i) == s2.charAt(j)) dp[i + 1][j + 1] = 1 + dp[i][j];

else dp[i + 1][j + 1] = Math.max(dp[i][j + 1], dp[i + 1][j]);

return dp[s1.length()][s2.length()];

}

**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).

public int longestCommonSubsequence(String s1, String s2) {

int m = s1.length(), n = s2.length();

if (m < n) return longestCommonSubsequence(s2, s1);

int[][] dp = new int[2][n + 1];

for (int i = 0, k = 1; i < m; ++i, k ^= 1)

for (int j = 0; j < n; ++j)

if (s1.charAt(i) == s2.charAt(j)) dp[k][j + 1] = 1 + dp[k ^ 1][j];

else dp[k][j + 1] = Math.max(dp[k ^ 1][j + 1], dp[k][j]);

return dp[m % 2][n];

}

**Analysis:**

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