# Sequence Match

Sequence match is one of the common patterns in the dynamic programming. The original task is to get the longest common sequence between two string, the solution is to establish a two dimension matrix, use one string as row, another as column, compare the accumulated state with a position in string A, against a position in string B. The result in the last right and bottom corner is normally the answer.

## 1143. Longest Common Subsequence

Medium

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.

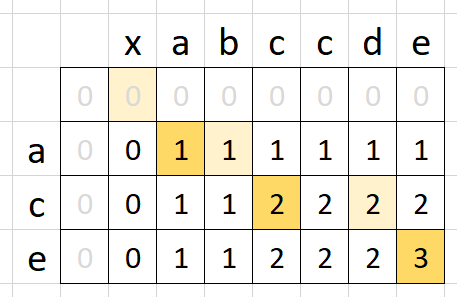
**Constraints:**

* 1 <= text1.length <= 1000
* 1 <= text2.length <= 1000
* The input strings consist of lowercase English characters only.

### Analysis

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.  


/// <summary>

/// Leet code #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.

///

///

/// Constraints:

/// 1. 1 <= text1.length <= 1000

/// 2. 1 <= text2.length <= 1000

/// 3. The input strings consist of lowercase English characters only.

/// </summary>

int LeetCodeDP::longestCommonSubsequence(string text1, string text2)

{

vector<vector<int>> dp(text1.size(), vector<int>(text2.size()));

for (size\_t i = 0; i < text1.size(); i++)

{

for (size\_t j = 0; j < text2.size(); j++)

{

if (i == 0 || j == 0)

{

if (text1[i] == text2[j]) dp[i][j] = 1;

else if (i > 0) dp[i][j] = dp[i-1][j];

else if (j > 0) dp[i][j] = dp[i][j - 1];

else dp[i][j] = 0;

}

else

{

if (text1[i] == text2[j]) dp[i][j] = dp[i - 1][j - 1] + 1;

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

}

}

}

return dp[text1.size() - 1][text2.size() - 1];

}

## 1092. Shortest Common Supersequence

Hard

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.

*(A string S is a subsequence of string T if deleting some number of characters from T (possibly 0, and the characters are chosen anywhere from T) results in the string S.)*

**Example 1:**

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

**Output:** "cabac"

**Explanation:**

str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".

str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".

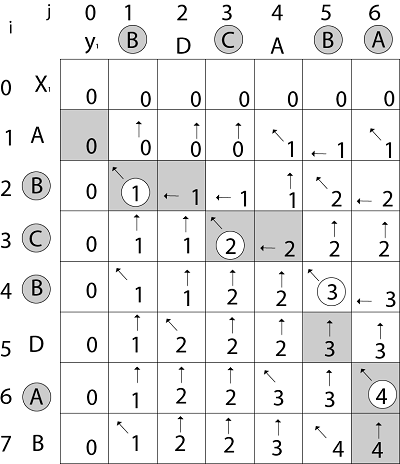
The answer provided is the shortest such string that satisfies these properties.

**Note:**

1. 1 <= str1.length, str2.length <= 1000
2. str1 and str2 consist of lowercase English letters.

### Analysis

1. Find LCS;  
   e.g, s1 = "XABCBDAB", s2 = "yBDCABA", for LCS DP computation, please refer to the following picture, which will also be helpful to understand the building process for the final result.



1. Reversely append the chars to StringBuilder, if the char is among the LCS, choose either one between the two strings.  
   a) start from i = m - 1 and j = n - 1, check if the corresponding chars are equal, that is, s1.charAt(i) == s2.charAt(j); if yes, append either of them; if no, append the char with larger dp value.  
   b) whether s1 or s2 reaches left end first, continue to append remaining chars in the other string.

/// <summary>

/// Leet code #1092. Shortest Common Supersequence

///

/// 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.

///

/// (A string S is a subsequence of string T if deleting some number of

/// characters from T (possibly 0, and the characters are chosen anywhere

/// from T) results in the string S.)

///

/// Example 1:

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

/// Output: "cabac"

/// Explanation:

/// str1 = "abac" is a substring of "cabac" because we can delete the

/// first "c".

/// str2 = "cab" is a substring of "cabac" because we can delete the last

/// "ac".

/// The answer provided is the shortest such string that satisfies these

/// properties.

///

/// Note:

/// 1. 1 <= str1.length, str2.length <= 1000

/// 2. str1 and str2 consist of lowercase English letters.

/// </summary>

string LeetCode::shortestCommonSupersequence(string str1, string str2)

{

vector<vector<int>> dp(str1.size(), vector<int>(str2.size()));

for (size\_t i = 0; i < str1.size(); i++)

{

for (size\_t j = 0; j < str2.size(); j++)

{

if (str1[i] == str2[j])

{

dp[i][j] = 1;

if (i > 0 && j > 0) dp[i][j] = dp[i - 1][j - 1] + 1;

}

else

{

dp[i][j] = 0;

if (i > 0) dp[i][j] = dp[i - 1][j];

if (j > 0)dp[i][j] = max(dp[i][j], dp[i][j - 1]);

}

}

}

int i = str1.size() - 1;

int j = str2.size() - 1;

string result;

while (i >= 0 || j >= 0)

{

if (i < 0)

{

result.push\_back(str2[j]);

j--;

}

else if (j < 0)

{

result.push\_back(str1[i]);

i--;

}

else if (str1[i] == str2[j])

{

result.push\_back(str1[i]);

i--;

j--;

}

else if (j > 0 && dp[i][j] == dp[i][j - 1])

{

result.push\_back(str2[j]);

j--;

}

else if (i > 0 && dp[i][j] == dp[i - 1][j])

{

result.push\_back(str1[i]);

i--;

}

else

{

result.push\_back(str1[i]);

i--;

}

}

std::reverse(result.begin(), result.end());

return result;

}

## 1062. Longest Repeating Substring

Medium

Given a string S, find out the length of the longest repeating substring(s). Return 0 if no repeating substring exists.

**Example 1:**

**Input:** "abcd"

**Output:** 0

**Explanation:** There is no repeating substring.

**Example 2:**

**Input:** "abbaba"

**Output:** 2

**Explanation:** The longest repeating substrings are "ab" and "ba", each of which occurs twice.

**Example 3:**

**Input:** "aabcaabdaab"

**Output:** 3

**Explanation:** The longest repeating substring is "aab", which occurs 3 times.

**Example 4:**

**Input:** "aaaaa"

**Output:** 4

**Explanation:** The longest repeating substring is "aaaa", which occurs twice.

**Note:**

1. The string S consists of only lowercase English letters from 'a' - 'z'.
2. 1 <= S.length <= 1500

### Analysis

We can consider the self string match (no sequence match, so only compare on major diagonal line). A i-th position the string should only compare with the position before it.

/// <summary>

/// Leet code #1062. Longest Repeating Substring

///

/// Given a string S, find out the length of the longest repeating

/// substring(s). Return 0 if no repeating substring exists.

///

/// Example 1:

/// Input: "abcd"

/// Output: 0

/// Explanation: There is no repeating substring.

///

/// Example 2:

/// Input: "abbaba"

/// Output: 2

/// Explanation: The longest repeating substrings are "ab" and "ba", each of

/// which occurs twice.

///

/// Example 3:

/// Input: "aabcaabdaab"

/// Output: 3

/// Explanation: The longest repeating substring is "aab", which occurs 3

/// times.

///

/// Example 4:

///

/// Input: "aaaaa"

/// Output: 4

/// Explanation: The longest repeating substring is "aaaa", which occurs twice.

///

/// Note:

///

/// 1. The string S consists of only lowercase English letters from 'a' - 'z'.

/// 2. 1 <= S.length <= 1500

/// </summary>

int LeetCode::longestRepeatingSubstring(string S)

{

vector<vector<int>> dp(S.size(), vector<int>(S.size()));

int result = 0;

for (size\_t i = 0; i < S.size(); i++)

{

for (size\_t j = 0; j < i; j++)

{

if (S[i] == S[j])

{

if (i == 0 || j == 0)

{

dp[i][j] = 1;

}

else

{

dp[i][j] = dp[i - 1][j - 1] + 1;

}

}

result = max(result, dp[i][j]);

}

}

return result;

}

## 1147. Longest Chunked Palindrome Decomposition

Hard

Return the largest possible k such that there exists a\_1, a\_2, ..., a\_k such that:

* Each a\_i is a non-empty string;
* Their concatenation a\_1 + a\_2 + ... + a\_k is equal to text;
* For all 1 <= i <= k,  a\_i = a\_{k+1 - i}.

**Example 1:**

**Input:** text = "ghiabcdefhelloadamhelloabcdefghi"

**Output:** 7

**Explanation:** We can split the string on "(ghi)(abcdef)(hello)(adam)(hello)(abcdef)(ghi)".

**Example 2:**

**Input:** text = "merchant"

**Output:** 1

**Explanation:** We can split the string on "(merchant)".

**Example 3:**

**Input:** text = "antaprezatepzapreanta"

**Output:** 11

**Explanation:** We can split the string on "(a)(nt)(a)(pre)(za)(tpe)(za)(pre)(a)(nt)(a)".

**Example 4:**

**Input:** text = "aaa"

**Output:** 3

**Explanation:** We can split the string on "(a)(a)(a)".

**Constraints:**

* text consists only of lowercase English characters.
* 1 <= text.length <= 1000

### Analysis

Although a simple brute force solution also work, but we can do it as DP, we find the match substring position from the start, if there are multiple match, we will always remember the latest. Then skip the matched substring and match the next position, if there is not match position this is the last substring left.

/// <summary>

/// Leet code #1147. Longest Chunked Palindrome Decomposition

///

/// Return the largest possible k such that there exists a\_1, a\_2, ..., a\_k

/// such that:

///

/// Each a\_i is a non-empty string;

/// Their concatenation a\_1 + a\_2 + ... + a\_k is equal to text;

/// For all 1 <= i <= k, a\_i = a\_{k+1 - i}.

///

/// Example 1:

/// Input: text = "ghiabcdefhelloadamhelloabcdefghi"

/// Output: 7

/// Explanation: We can split the string on

/// "(ghi)(abcdef)(hello)(adam)(hello)(abcdef)(ghi)".

///

/// Example 2:

/// Input: text = "merchant"

/// Output: 1

/// Explanation: We can split the string on "(merchant)".

///

/// Example 3:

/// Input: text = "antaprezatepzapreanta"

/// Output: 11

/// Explanation: We can split the string on

/// "(a)(nt)(a)(pre)(za)(tpe)(za)(pre)(a)(nt)(a)".

///

/// Example 4:

/// Input: text = "aaa"

/// Output: 3

/// Explanation: We can split the string on "(a)(a)(a)".

///

/// Constraints:

/// 1. text consists only of lowercase English characters.

/// 2. 1 <= text.length <= 1000

/// </summary>

int LeetCodeDP::longestDecomposition(string text)

{

vector<vector<int>> dp(text.size(), vector<int>(text.size()));

for (int i = 0; i < (int)text.size(); i++)

{

for (int j = i; j < (int)text.size(); j++)

{

if (text[i] == text[j])

{

if (i == 0 || j == 0) dp[i][j] = 1;

else dp[i][j] = dp[i - 1][j - 1] + 1;

}

else

{

dp[i][j] = 0;

}

}

}

int result = 0;

for (int j = text.size() - 1; j >= (int)text.size() / 2;)

{

int start = (int)text.size() - 1 - j;

for (int i = start; i < (int)text.size(); i++)

{

if (i - start + 1 <= dp[i][j])

{

if (i != j) result += 2;

else result++;

j -= i - start + 1;

break;

}

}

}

return result;

}

## 1035. Uncrossed Lines

Medium

We write the integers of A and B (in the order they are given) on two separate horizontal lines.

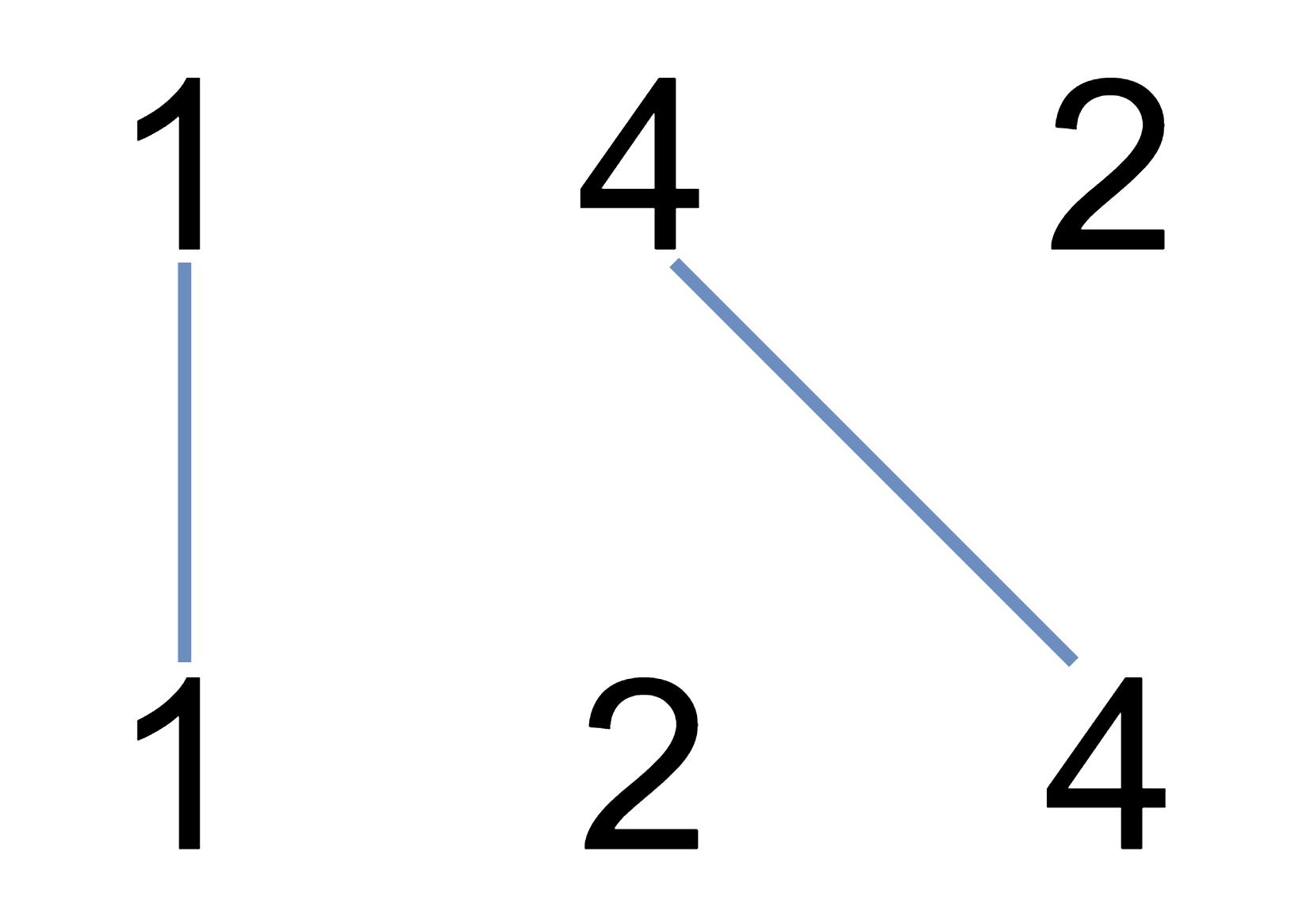
Now, we may draw *connecting lines*: a straight line connecting two numbers A[i] and B[j] such that:

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

Note that a connecting lines cannot intersect even at the endpoints: each number can only belong to one connecting line.

Return the maximum number of connecting lines we can draw in this way.

**Example 1:**



**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.

**Example 2:**

**Input:** A = [2,5,1,2,5], B = [10,5,2,1,5,2]

**Output:** 3

**Example 3:**

**Input:** A = [1,3,7,1,7,5], B = [1,9,2,5,1]

**Output:** 2

**Note:**

1. 1 <= A.length <= 500
2. 1 <= B.length <= 500
3. 1 <= A[i], B[i] <= 2000

### Analysis

The uncrossed lines actually are the common sequence for the two string.

/// <summary>

/// Leet code #1035. Uncrossed Lines

///

/// We write the integers of A and B (in the order they are given) on two

/// separate horizontal lines.

///

/// Now, we may draw a straight line connecting two numbers A[i] and B[j]

/// as long as A[i] == B[j], and the line we draw does not intersect any

/// other connecting (non-horizontal) line.

///

/// Return the maximum number of connecting lines we can draw in this way.

///

/// Example 1:

/// 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.

///

/// Example 2:

/// Input: A = [2,5,1,2,5], B = [10,5,2,1,5,2]

/// Output: 3

///

/// Example 3:

/// Input: A = [1,3,7,1,7,5], B = [1,9,2,5,1]

/// Output: 2

/// </summary>

int LeetCode::maxUncrossedLines(vector<int>& A, vector<int>& B)

{

vector<vector<int>> dp(A.size() + 1, vector<int>(B.size() + 1));

dp[0][0] = 0;

for (size\_t i = 0; i < A.size(); i++)

{

for (size\_t j = 0; j < B.size(); j++)

{

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

if (A[i] == B[j]) dp[i + 1][j + 1] =

max(dp[i + 1][j+1], dp[i][j] + 1);

}

}

return dp[A.size()][B.size()];

}

## 72. Edit Distance

Hard

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:

1. Insert a character
2. Delete a character
3. 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')

### Analysis

To apply DP, we define the state dp[i][j] to be the minimum number of operations to convert word1[0..i) to word2[0..j).

For the base case, that is, to convert a string to an empty string, the mininum number of operations (deletions) is just the length of the string. So we have dp[i][0] = i and dp[0][j] = j.

For the general case to convert word1[0..i) to word2[0..j), we break this problem down into sub-problems. Suppose we have already known how to convert word1[0..i - 1) to word2[0..j - 1) (dp[i - 1][j - 1]), if word1[i - 1] == word2[j - 1], then no more operation is needed and dp[i][j] = dp[i - 1][j - 1].

If word1[i - 1] != word2[j - 1], we need to consider three cases.

1. **Replace** word1[i - 1] by word2[j - 1] (dp[i][j] = dp[i - 1][j - 1] + 1);
2. If word1[0..i - 1) = word2[0..j) then **delete** word1[i - 1] (dp[i][j] = dp[i - 1][j] + 1);
3. If word1[0..i) + word2[j - 1] = word2[0..j) then **insert** word2[j - 1] to word1[0..i) (dp[i][j] = dp[i][j - 1] + 1).

So when word1[i - 1] != word2[j - 1], dp[i][j] will just be the minimum of the above three cases.

/// <summary>

/// Leet code #72. Edit Distance

/// Given two words word1 and word2, find the minimum number of steps

/// required to convert word1 to word2. (each operation is

/// counted as 1 step.)

/// You have the following 3 operations permitted on a word:

/// a) Insert a character

/// b) Delete a character

/// c) Replace a character

/// </summary>

int LeetCode::minDistance(string word1, string word2)

{

int distance = 0;

vector<vector<int>> distance\_map;

// The matrix should be [s.size() + 1][p.size() + 1]

// This is because the first row and first column

// if left for empty string.

for (size\_t i = 0; i <= word1.size(); i++)

{

distance\_map.push\_back(vector<int>(word2.size() + 1));

}

for (size\_t i = 0; i <= word1.size(); i++)

{

for (size\_t j = 0; j <= word2.size(); j++)

{

// when word1 is empty string, the distance is to insert every

/// character in word2

if (i == 0)

{

distance\_map[i][j] = j;

}

// when word2 is empty string, the distance is to delete every

// character of word1

else if (j == 0)

{

distance\_map[i][j] = i;

}

else

{

// if the character match, we inherit the edit size from

// the subset

if (word1[i - 1] == word2[j - 1])

{

distance\_map[i][j] = distance\_map[i - 1][j - 1];

}

// otherwise, we increase the size by 1 from the subset,

// by replacing a character

else

{

distance\_map[i][j] = distance\_map[i - 1][j - 1] + 1;

}

distance\_map[i][j] = min(distance\_map[i][j], distance\_map[i - 1][j] + 1);

distance\_map[i][j] = min(distance\_map[i][j], distance\_map[i][j - 1] + 1);

}

}

}

return distance\_map[word1.size()][word2.size()];

}

## 115. Distinct Subsequences

Hard

Given a string **S** and a string **T**, count the number of distinct subsequences of **S** which equals **T**.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

**Example 1:**

**Input:** S = "rabbbit", T = "rabbit"

**Output:** 3

**Explanation:**

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

(The caret symbol ^ means the chosen letters)

rabbbit

^^^^ ^^

rabbbit

^^ ^^^^

rabbbit

^^^ ^^^

**Example 2:**

**Input:** S = "babgbag", T = "bag"

**Output:** 5

**Explanation:**

As shown below, there are 5 ways you can generate "bag" from S.

(The caret symbol ^ means the chosen letters)

babgbag

^^ ^

babgbag

^^ ^

babgbag

^ ^^

babgbag

^ ^^

babgbag

^^^

### Analysis

The idea is the following:

* we will build an array mem where mem[i+1][j+1] means that S[0..j] contains T[0..i] that many times as distinct subsequences. Therefor the result will be mem[T.length()][S.length()].
* we can build this array rows-by-rows:
* the first row must be filled with 1. That's because the empty string is a subsequence of any string but only 1 time. So mem[0][j] = 1 for every j. So with this we not only make our lives easier, but we also return correct value if T is an empty string.
* the first column of every rows except the first must be 0. This is because an empty string cannot contain a non-empty string as a substring -- the very first item of the array: mem[0][0] = 1, because an empty string contains the empty string 1 time.

So the matrix looks like this:

S 0123....j

T +----------+

|1111111111|

0 |0 |

1 |0 |

2 |0 |

. |0 |

. |0 |

i |0 |

From here we can easily fill the whole grid: for each (x, y), we check if S[x] == T[y] we add the previous item and the previous item in the previous row, otherwise we copy the previous item in the same row. The reason is simple:

* if the current character in S doesn't equal to current character T, then we have the same number of distinct subsequences as we had without the new character.
* if the current character in S equal to the current character T, then the distinct number of subsequences: the number we had before **plus** the distinct number of subsequences we had with less longer T and less longer S.

An example:  
S: [acdabefbc] and T: [ab]

first we check with a:

\* \*

S = [acdabefbc]

mem[1] = [0111222222]

then we check with ab:

\* \* ]

S = [acdabefbc]

mem[1] = [0111222222]

mem[2] = [0000022244]

And the result is 4, as the distinct subsequences are:

S = [a b ]

S = [a b ]

S = [ ab ]

S = [ a b ]

/// <summary>

/// Leet code #115. Distinct Subsequences

/// Given a string S and a string T, count the number of distinct subsequences

/// of T in S.

/// A subsequence of a string is a new string which is formed from the original

/// string by deleting some (can be none) of the characters without disturbing

/// the relative positions of the remaining characters. (ie, "ACE" is a

/// subsequence of "ABCDE" while "AEC" is not). For example,

/// Here is an example:

/// S = "rabbbit", T = "rabbit"

/// </summary>

int LeetCode::numDistinct(string s, string t)

{

if (s.empty() || t.empty()) return 0;

vector<vector<unsigned int>> dp(t.size(), vector<unsigned int>(s.size()));

for (size\_t i = 0; i < t.size(); i++)

{

for (size\_t j = 0; j < s.size(); j++)

{

if (t[i] == s[j])

{

if (i == 0 && j == 0) dp[i][j] = 1;

else if (i == 0) dp[i][j] = (1 + dp[i][j - 1]);

else if (j == 0) dp[i][j] = 0;

else

{

dp[i][j] = (dp[i - 1][j - 1] + dp[i][j - 1]);

}

}

else

{

if (j > 0) dp[i][j] = dp[i][j - 1];

}

}

}

return dp[t.size() - 1][s.size() - 1];

}

## 10. Regular Expression Matching

Hard

Given an input string (s) and a pattern (p), implement regular expression matching with support for '.' and '\*'.

'.' Matches any single character.

'\*' Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

**Note:**

* s could be empty and contains only lowercase letters a-z.
* p could be empty and contains only lowercase letters a-z, and characters like . or \*.

**Example 1:**

**Input:**

s = "aa"

p = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Example 2:**

**Input:**

s = "aa"

p = "a\*"

**Output:** true

**Explanation:** '\*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

**Example 3:**

**Input:**

s = "ab"

p = ".\*"

**Output:** true

**Explanation:** ".\*" means "zero or more (\*) of any character (.)".

**Example 4:**

**Input:**

s = "aab"

p = "c\*a\*b"

**Output:** true

**Explanation:** c can be repeated 0 times, a can be repeated 1 time. Therefore, it matches "aab".

**Example 5:**

**Input:**

s = "mississippi"

p = "mis\*is\*p\*."

**Output:** false

### Analysis

Here are some conditions to figure out, then the logic can be very straightforward.

1, If p.charAt(j) == s.charAt(i) : dp[i][j] = dp[i-1][j-1];

2, If p.charAt(j) == '.' : dp[i][j] = dp[i-1][j-1];

3, If p.charAt(j) == '\*':

here are two sub conditions:

1 if p.charAt(j-1) != s.charAt(i) : dp[i][j] = dp[i][j-2] //in this case, a\* only counts as empty

2 if p.charAt(i-1) == s.charAt(i) or p.charAt(i-1) == '.':

dp[i][j] = dp[i-1][j] //in this case, a\* counts as multiple a

or dp[i][j] = dp[i][j-1] // in this case, a\* counts as single a

or dp[i][j] = dp[i][j-2] // in this case, a\* counts as empty

/// <summary>

/// Leet code # 10. Regular Expression Matching

///

/// Implement regular expression matching with support for '.' and '\*'.

///

/// '.' Matches any single character.

/// '\*' Matches zero or more of the preceding element.

///

/// The matching should cover the entire input string (not partial).

/// The function prototype should be:

///

/// bool isMatchRegularExpression(const char \*s, const char \*p)

///

/// Some examples:

/// isMatch("aa","a") -> false

/// isMatch("aa","aa") -> true

/// isMatch("aaa","aa") -> false

/// isMatch("aa", "a\*") -> true

/// isMatch("aa", ".\*") -> true

/// isMatch("ab", ".\*") -> true

/// isMatch("aab", "c\*a\*b") -> true

/// </summary>

bool LeetCode::isMatchRegularExpression(string s, string p)

{

vector<vector<bool>> matrix;

matrix.push\_back(vector<bool>(p.size() + 1));

matrix[0][0] = true;

for (size\_t i = 0; i < p.size(); i++)

{

if ((p[i] == '\*') && (i >= 1))

matrix[0][i + 1] = matrix[0][i - 1];

else

matrix[0][i + 1] = false;

}

for (size\_t i = 0; i < s.size(); i++)

{

matrix.push\_back(vector<bool>(p.size() + 1));

matrix[i + 1][0] = false;

for (size\_t j = 0; j < p.size(); j++)

{

if ((s[i] == p[j]) || (p[j] == '.'))

{

matrix[i + 1][j + 1] = matrix[i][j];

}

else if ((p[j] == '\*') && (j > 0))

{

// if character matches, and the pattern character is \*,

// it can match under 3 conditions.

// 1. if without \*, it matches,

// 2. if without \* and the character before

// (0 occurance for character),

// 3. match until the last character in source,

// This means multiple occurance.

if (((s[i] == p[j - 1]) || (p[j - 1] == '.')))

{

if (matrix[i + 1][j - 1] || matrix[i + 1][j] || matrix[i][j + 1])

{

matrix[i + 1][j + 1] = true;

}

else

{

matrix[i + 1][j + 1] = false;

}

}

else

{

if (matrix[i + 1][j - 1])

{

matrix[i + 1][j + 1] = true;

}

else

{

matrix[i + 1][j + 1] = false;

}

}

}

else

{

matrix[i + 1][j + 1] = false;

}

}

}

return matrix[s.size()][p.size()];

}