<https://leetcode.com/problems/interleaving-string/description/>

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

An **interleaving** of two strings s and t is a configuration where s and t are divided into n and m substrings respectively, such that:

s = s1 + s2 + ... + sn

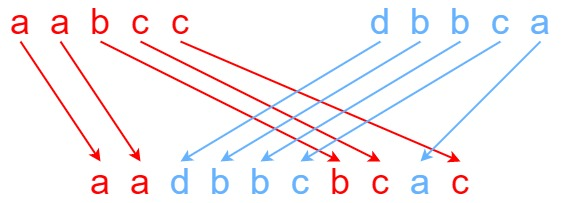
t = t1 + t2 + ... + tm

|n - m| <= 1

The interleaving is s1 + t1 + s2 + t2 + s3 + t3 + ... or t1 + s1 + t2 + s2 + t3 + s3 + ...

Note: a + b is the concatenation of strings a and b.

**Example 1:**



**Input:** s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"

**Output:** true

**Explanation:** One way to obtain s3 is:Split s1 into s1 = "aa" + "bc" + "c", and s2 into s2 = "dbbc" + "a".Interleaving the two splits, we get "aa" + "dbbc" + "bc" + "a" + "c" = "aadbbcbcac".Since s3 can be obtained by interleaving s1 and s2, we return true.

**Example 2:**

**Input:** s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"

**Output:** false

**Explanation:** Notice how it is impossible to interleave s2 with any other string to obtain s3.

**Example 3:**

**Input:** s1 = "", s2 = "", s3 = ""

**Output:** true

**Constraints:**

0 <= s1.length, s2.length <= 100

0 <= s3.length <= 200

s1, s2, and s3 consist of lowercase English letters.

**Follow up:** Could you solve it using only O(s2.length) additional memory space?

**Attempt 1: 2024-01-03**

**Wrong Solution**

Below recursive solution is wrong, failed on input as s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac", expect result is true, but below wrong code return false

class Solution {

public boolean isInterleave(String s1, String s2, String s3) {

int len1 = s1.length();

int len2 = s2.length();

int len3 = s3.length();

if(len1 + len2 != len3) {

return false;

}

return helper(s1, 0, s2, 0, s3);

}

private boolean helper(String s1, int i, String s2, int j, String s3) {

if(i == s1.length() && j == s2.length()) {

return true;

}

if(i == s1.length() || j == s2.length()) {

return false;

}

for(int m = i + 1; m <= s1.length(); m++) {

for(int n = j + 1; n <= s2.length(); n++) {

String s11 = s1.substring(0, m);

String s12 = s1.substring(m);

String s21 = s2.substring(0, n);

String s22 = s2.substring(n);

String s31 = s3.substring(0, m + n);

String s32 = s3.substring(m + n);

if((s11 + s21).equals(s31)) {

return helper(s12, 0, s22, 0, s32);

} else if((s21 + s11).equals(s31)) {

return helper(s22, 0, s12, 0, s32);

} else {

return false;

}

}

}

return true;

}

}

The issue with your current solution lies in the nested loops within the **helper** method. The loops iterate over all possible split positions in both strings **s1** and **s2**, but the early return inside the loops might not cover all possibilities.

In this corrected version below, the recursive calls are made based on the characters matching at the current positions in **s1**, **s2**, and **s3**. This ensures that all possible interleavings are considered, and it avoids premature return statements that might lead to incorrect results.

**Solution 1: Native DFS (30min)**

**无论以什么方式写helper方法都绕不过拿s1或者s2的最后一个字母和s3最后一个字母比较这个关键的递归逻辑基础，为何这个比较是不可或缺？**

**原理如下：**

<https://leetcode.com/problems/interleaving-string/solutions/2249509/python-simple-solution-w-explanation-recursion-dp/>

We don't know the size of each substring or the number of substrings beforehand. So, we can take all possible substrings of s1 and s2 and check if s3 can be formed by interleaving them. **At each step, we have two options: choose a character from s1 or s2**. Let's call our recursive function dfs(i, j). Then the two choices can be represented as:

**dfs(i + 1, j): Choose a character at ith index from s1**

**dfs(i, j + 1): Choose a character at jth index from s2**

Actually, we can make this choice more smartly. Instead of considering all possibilities, **we can make either/both choice(s) only when it matches the character at the i + jth index of s3.**

**而且我们采用逐字比较的方式是基于表面上s1和s2在构成s3的时候必须间隔着来，比如最直觉的理解是s3 = "abc"，其中'a'和'c'来自s1，'b'来自s2，这样就实现了间隔着来，但实际上并不是，当前字符的下一个字符完全可以来自同一个字符串，依然是s3 = "abc"，其中'a'来自s1的时候，下一个字符'b'也可以来自s1，然后'c'来自s2，也就是说s3中的下一个字符或多个字符完全可以来自同一个字符串，而并不违背interleaving字符串的构造定义，所以我们的逻辑中没必要强行设定下一个或者多个字符来自另一个字符串，比如当前字符来自s1，没必要设定下一个或者多个字符来自s2，这也反过来要求我们必须对s3进行逐个字符个拆解和比较，每个字符都存在两种可能，要么来自s1，要么来自s2，甚至当用i和j两个指针分别扫描s1和s2时，如果s1[i] = s2[j]，则存在同时需要考虑来自s1和s2的可能性，这也解释了逻辑上是并列的两个if而不是if...else if**

**Style 1: Compare by character only (TLE 105/106)**

class Solution {

    public boolean isInterleave(String s1, String s2, String s3) {

        int len1 = s1.length();

        int len2 = s2.length();

        int len3 = s3.length();

        if (len1 + len2 != len3) {

            return false;

        }

        return helper(s1, 0, s2, 0, s3);

    }

    private boolean helper(String s1, int i, String s2, int j, String s3) {

        if(i == s1.length() && j == s2.length()) {

            return true;

        }

        if(i < s1.length() && s1.charAt(i) == s3.charAt(i + j)) {

            if(helper(s1, i + 1, s2, j, s3)) {

                return true;

            }

        }

        // Parallel condition, not else if, because the when s1's substring

        // last char equals s3's substring last char doesn't block the

        // potential s2's subtring last char equals s3's substring last char

        if(j < s2.length() && s2.charAt(j) == s3.charAt(i + j)) {

            if(helper(s1, i, s2, j + 1, s3)) {

                return true;

            }

        }

        return false;

    }

}

Time Complexity: O(2^(m + n))

At each step, we have two choices, so 2 \* 2 \* 2 ... (m + n) times.

Space Complexity: O(m + n)

Recursion stack space.

**Refer to**

<https://leetcode.com/problems/interleaving-string/solutions/2249509/python-simple-solution-w-explanation-recursion-dp/>

Given three strings s1, s2 and s3, we need to check if s3 can be formed by an interleaving of s1 and s2.

An **interleaving** of two strings, s1 and s2, means that s1 is divided into x and s2 is divided into y contiguous substrings, respectively. Then those substrings are concatenated without changing the order of their occurrence in s1 and s2.

Note that the condition |x - y| <=1 always holds true.

❌ **Solution I: Recursion [TLE]**

We don't know the size of each substring or the number of substrings beforehand. So, we can take all possible substrings of s1 and s2 and check if s3 can be formed by interleaving them. At each step, we have two options: choose a character from s1 or s2. Let's call our recursive function dfs(i, j). Then the two choices can be represented as:

dfs(i + 1, j): Choose a character at ith index from s1

dfs(i, j + 1): Choose a character at jth index from s2

Actually, we can make this choice more smartly. Instead of considering all possibilities, we can make either/both choice(s) only when it matches the character at the i + jth index of s3.

class Solution:

def isInterleave(self, s1: str, s2: str, s3: str) -> bool:

if len(s1) + len(s2) != len(s3):

return False

def dfs(i, j):

if i == len(s1) and j == len(s2):

return True

choose\_s1, choose\_s2 = False, False

if i < len(s1) and s1[i] == s3[i + j]:

choose\_s1 = dfs(i + 1, j)

if j < len(s2) and s2[j] == s3[i + j]:

choose\_s2 = dfs(i, j + 1)

return choose\_s1 or choose\_s2

return dfs(0, 0)

**Java version (convert by chatGPT)**

class Solution {

    public boolean isInterleave(String s1, String s2, String s3) {

        if (s1.length() + s2.length() != s3.length()) {

            return false;

        }

        return dfs(0, 0, s1, s2, s3);

    }

    private boolean dfs(int i, int j, String s1, String s2, String s3) {

        if (i == s1.length() && j == s2.length()) {

            return true;

        }

        boolean chooseS1 = false, chooseS2 = false;

        if (i < s1.length() && s1.charAt(i) == s3.charAt(i + j)) {

            chooseS1 = dfs(i + 1, j, s1, s2, s3);

        }

        if (j < s2.length() && s2.charAt(j) == s3.charAt(i + j)) {

            chooseS2 = dfs(i, j + 1, s1, s2, s3);

        }

        return chooseS1 || chooseS2;

    }

}

Why have I named the inside function as dfs? Because if we trace our actions, we can observe that it forms a binary tree. **Don't worry** if you are not familiar with this term. The following visualization will help you to understand what I mean.

┏━━━━━━┓

╭─────────────────────┨ 0, 0 ┠──────────────────────╮

│ ┗━━━━━━┛ │

┏━━━┷━━┓ ┏━━┷━━━┓

╭───────┨ 1, 0 ┠─────────╮ ╭─────────┨ 0, 1 ┠─────────╮

│ ┗━━━━━━┛ │ │ ┗━━━━━━┛ │

┏━━━┷━━┓ ┏━━┷━━━┓ ┏━━━┷━━┓ ┏━━┷━━━┓

┃ 2, 0 ┃ ┃ 1, 1 ┃ ┃ 1, 1 ┃ ┃ 0, 2 ┃

┗━━━━━━┛ ┗━━━━━━┛ ┗━━━━━━┛ ┗━━━━━━┛

. . . .

. . . .

. . . .

In dfs, we traverse all the paths one by one. So, here our paths will be:

(0, 0) -> (1, 0) -> (2, 0) -> ...

(0, 0) -> (1, 0) -> (1, 1) -> ...

(0, 0) -> (0, 1) -> (1, 1) -> ...

(0, 0) -> (0, 1) -> (0, 2) -> ...

**Time Complexity:** O(2^(m + n))

At each step, we have two choices, so 2 \* 2 \* 2 ... (m + n) times.

**Space Complexity:** O(m + n)

Recursion stack space.

**Style 2: Rewrite the base condition by adding if(i == s1.length()) and if(j == s2.length()) to do if(i == s1.length() || j == s2.length()) job, and requires substring() and equals() to use (TLE 99/106)**

class Solution {

    public boolean isInterleave(String s1, String s2, String s3) {

        int len1 = s1.length();

        int len2 = s2.length();

        int len3 = s3.length();

        if(len1 + len2 != len3) {

            return false;

        }

        return helper(s1, 0, s2, 0, s3);

    }

    private boolean helper(String s1, int i, String s2, int j, String s3) {

        if(i == s1.length() && j == s2.length()) {

            return true;

        }

        if(i == s1.length()) {

            return s2.substring(j).equals(s3.substring(i + j));

        }

        if(j == s2.length()) {

            return s1.substring(i).equals(s3.substring(i + j));

        }

// Parallel condition, not else if, because the when s1's substring

        // last char equals s3's substring last char doesn't block the

        // potential s2's subtring last char equals s3's substring last char

        boolean result = false;

        if(s1.charAt(i) == s3.charAt(i + j)) {

            result = helper(s1, i + 1, s2, j, s3);

        }

        if(s2.charAt(j) == s3.charAt(i + j)) {

            result |= helper(s1, i, s2, j + 1, s3);

        }

        return result;

    }

}

Time Complexity: O(2^(m + n))

At each step, we have two choices, so 2 \* 2 \* 2 ... (m + n) times.

Space Complexity: O(m + n)

Recursion stack space.

**Refer to**

<https://leetcode.com/problems/interleaving-string/solutions/31907/my-accepted-java-recursive-solution-for-interleaving-string/>

The private method isInterleave is the recursive method. it takes additional i1, i2, i3 as the start indexes of s1, s2, s3, so it solves the substring of s1, s2, s3 with those start indexes.

The recursion starting condition is i1, i2, i3 are set to 0, means it solves the whole string.

in each recursion, it will just check the first character in s3 with s2 and s1, if it equals s1, it will increase i3 and i1 to solve remain, if remain return true, this recursion will also return true. Same logic for s2.

The end condition is when remain of either s1 or s2 is empty, then just compare remain of s3 with remain of s1 or s2, if they are equal, it will return true.

A pure recursive solution will cause time limit exceed. We can optimize it by caching the false visited solutions in the visited set. That will short circuit many repeated search path.

public class Solution {

private static Set<Integer> visited; // The combination of i1, i2 has been visited and return false

public static boolean isInterleave(String s1, String s2, String s3) {

if(s3.length() != s1.length() + s2.length())

return false;

visited = new HashSet<Integer>();

return isInterleave(s1, 0, s2, 0, s3, 0);

}

private static boolean isInterleave(String s1, int i1, String s2, int i2, String s3, int i3) {

int hash = i1 \* s3.length() + i2;

if(visited.contains(hash))

return false;

if(i1 == s1.length())

return s2.substring(i2).equals(s3.substring(i3));

if(i2 == s2.length())

return s1.substring(i1).equals(s3.substring(i3));

if(s3.charAt(i3) == s1.charAt(i1) && isInterleave(s1, i1+1, s2, i2, s3, i3+1) ||

s3.charAt(i3) == s2.charAt(i2) && isInterleave(s1, i1, s2, i2+1, s3, i3+1))

return true;

visited.add(hash);

return false;

}

}

<https://leetcode.com/problems/interleaving-string/solutions/31907/my-accepted-java-recursive-solution-for-interleaving-string/comments/1278254>

A little concise version by using boolean array

class Solution {

public boolean isInterleave(String s1, String s2, String s3) {

if (s1.length() + s2.length() != s3.length()) return false;

return dfs(s1, 0, s2, 0, s3, 0, new boolean[s1.length()][s2.length()]);

}

private boolean dfs(String s1, int index1, String s2, int index2, String s3, int index3, boolean[][] visited) {

if (index3 == s3.length()) return true;

if (index1 == s1.length()) return s3.substring(index3).equals(s2.substring(index2));

if (index2 == s2.length()) return s3.substring(index3).equals(s1.substring(index1));

if (visited[index1][index2]) return false;

boolean res = false;

if (s1.charAt(index1) == s3.charAt(index3)) res = dfs(s1, index1 + 1, s2, index2, s3, index3 + 1, visited);

if (s2.charAt(index2) == s3.charAt(index3)) res = res || dfs(s1, index1, s2, index2 + 1, s3, index3 + 1, visited); //dont use else if as we should try both, as we need to check or, as we just need one valid case

visited[index1][index2] = true;

return res;

}

}

**Solution 2: DFS + Memoization (10min)**

**Style 1: Compare by character only**

class Solution {

    public boolean isInterleave(String s1, String s2, String s3) {

        int len1 = s1.length();

        int len2 = s2.length();

        int len3 = s3.length();

        if (len1 + len2 != len3) {

            return false;

        }

        Boolean[][] memo = new Boolean[len1 + 1][len2 + 1];

        return helper(s1, 0, s2, 0, s3, memo);

    }

    private boolean helper(String s1, int i, String s2, int j, String s3, Boolean[][] memo) {

        if(i == s1.length() && j == s2.length()) {

            return true;

        }

        if(memo[i][j] != null) {

            return memo[i][j];

        }

        if(i < s1.length() && s1.charAt(i) == s3.charAt(i + j)) {

            if(helper(s1, i + 1, s2, j, s3, memo)) {

                return memo[i][j] = true;

            }

        }

        // Parallel condition, not else if, because the when s1's substring

        // last char equals s3's substring last char doesn't block the

        // potential s2's subtring last char equals s3's substring last char

        if(j < s2.length() && s2.charAt(j) == s3.charAt(i + j)) {

            if(helper(s1, i, s2, j + 1, s3, memo)) {

                return memo[i][j] = true;

            }

        }

        return memo[i][j] = false;

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(m \* n)

**Style 2: Rewrite the base condition by adding if(i == s1.length()) and if(j == s2.length()) to do if(i == s1.length() || j == s2.length()) job, and requires substring() and equals() to use**

class Solution {

    public boolean isInterleave(String s1, String s2, String s3) {

        int len1 = s1.length();

        int len2 = s2.length();

        int len3 = s3.length();

        if(len1 + len2 != len3) {

            return false;

        }

        Boolean[][] memo = new Boolean[len1 + 1][len2 + 1];

        return helper(s1, 0, s2, 0, s3, memo);

    }

    private boolean helper(String s1, int i, String s2, int j, String s3, Boolean[][] memo) {

        if(i == s1.length() && j == s2.length()) {

            return true;

        }

        if(memo[i][j] != null) {

            return memo[i][j];

        }

        if(i == s1.length()) {

            return memo[i][j] = s2.substring(j).equals(s3.substring(i + j));

        }

        if(j == s2.length()) {

            return memo[i][j] = s1.substring(i).equals(s3.substring(i + j));

        }

        // Parallel condition, not else if, because the when s1's substring

        // last char equals s3's substring last char doesn't block the

        // potential s2's subtring last char equals s3's substring last char

        boolean result = false;

        if(s1.charAt(i) == s3.charAt(i + j)) {

            result = helper(s1, i + 1, s2, j, s3, memo);

        }

        if(s2.charAt(j) == s3.charAt(i + j)) {

            result |= helper(s1, i, s2, j + 1, s3, memo);

        }

        return memo[i][j] = result;

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(m \* n)

**Solution 3: 2D DP (10min)**

class Solution {

    public boolean isInterleave(String s1, String s2, String s3) {

        int len1 = s1.length();

        int len2 = s2.length();

        int len3 = s3.length();

        if(len1 + len2 != len3) {

            return false;

        }

        // Top:0,0 -> Bottom:s1.length(),s2.length()

        boolean[][] dp = new boolean[len1 + 1][len2 + 1];

        dp[len1][len2] = true;

        // if(i == s1.length()) {return s2.substring(j).equals(s3.substring(i + j));}

        for(int j = len2 - 1; j >= 0; j--) {

            dp[len1][j] = s2.substring(j).equals(s3.substring(len1 + j));

        }

        // if(j == s2.length()) {return s1.substring(i).equals(s3.substring(i + j));}

        for(int i = len1 - 1; i >= 0; i--) {

            dp[i][len2] = s1.substring(i).equals(s3.substring(i + len2));

        }

        // if(s1.charAt(i) == s3.charAt(i + j)) {result = helper(s1, i + 1, s2, j, s3);}

        // if(s2.charAt(j) == s3.charAt(i + j)) {result |= helper(s1, i, s2, j + 1, s3);}

        for(int i = len1 - 1; i >= 0; i--) {

            for(int j = len2 - 1; j >= 0; j--) {

                // Parallel condition, not else if, because the when s1's substring

                // last char equals s3's substring last char doesn't block the

                // potential s2's subtring last char equals s3's substring last char

                if(s1.charAt(i) == s3.charAt(i + j)) {

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

                }

                if(s2.charAt(j) == s3.charAt(i + j)) {

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

                }

            }

        }

        return dp[0][0];

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(m \* n)

**Just a bit tweak, merge last column initialization into main for loop logic**

class Solution {

    public boolean isInterleave(String s1, String s2, String s3) {

        int len1 = s1.length();

        int len2 = s2.length();

        int len3 = s3.length();

        if(len1 + len2 != len3) {

            return false;

        }

        // Top:0,0 -> Bottom:s1.length(),s2.length()

        boolean[][] dp = new boolean[len1 + 1][len2 + 1];

        dp[len1][len2] = true;

        // if(i == s1.length()) {return s2.substring(j).equals(s3.substring(i + j));}

        for(int j = len2 - 1; j >= 0; j--) {

            dp[len1][j] = s2.substring(j).equals(s3.substring(len1 + j));

        }

        // Merge last column initialization into main for loop logic

        // if(j == s2.length()) {return s1.substring(i).equals(s3.substring(i + j));}

        //for(int i = len1 - 1; i >= 0; i--) {

        //    dp[i][len2] = s1.substring(i).equals(s3.substring(i + len2));

        //}

        // if(s1.charAt(i) == s3.charAt(i + j)) {result = helper(s1, i + 1, s2, j, s3);}

        // if(s2.charAt(j) == s3.charAt(i + j)) {result |= helper(s1, i, s2, j + 1, s3);}

        for(int i = len1 - 1; i >= 0; i--) {

            dp[i][len2] = s1.substring(i).equals(s3.substring(i + len2));

            for(int j = len2 - 1; j >= 0; j--) {

                // Parallel condition, not else if, because the when s1's substring

                // last char equals s3's substring last char doesn't block the

                // potential s2's subtring last char equals s3's substring last char

                if(s1.charAt(i) == s3.charAt(i + j)) {

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

                }

                if(s2.charAt(j) == s3.charAt(i + j)) {

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

                }

            }

        }

        return dp[0][0];

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(m \* n)

**Solution 4: 1D DP (10min)**

class Solution {

    public boolean isInterleave(String s1, String s2, String s3) {

        int len1 = s1.length();

        int len2 = s2.length();

        int len3 = s3.length();

        if(len1 + len2 != len3) {

            return false;

        }

        // Use a single-dimensional array to store intermediate results, remove row dimension

        //boolean[][] dp = new boolean[len1 + 1][len2 + 1];

        //dp[len1][len2] = true;

        boolean[] dp = new boolean[len2 + 1];

        dp[len2] = true;

        for(int j = len2 - 1; j >= 0; j--) {

            //dp[len1][j] = s2.substring(j).equals(s3.substring(len1 + j));

            dp[j] = s2.substring(j).equals(s3.substring(len1 + j));

        }

        // Loop through the rest of the array

        for(int i = len1 - 1; i >= 0; i--) {

            // Update the last column of the array

            dp[len2] = s1.substring(i).equals(s3.substring(i + len2));

            for(int j = len2 - 1; j >= 0; j--) {

                // Parallel condition, similar to the original code

                if(s1.charAt(i) == s3.charAt(i + j)) {

                    dp[j] = dp[j];

                } else {

                    dp[j] = false;

                }

                if(s2.charAt(j) == s3.charAt(i + j)) {

                    dp[j] |= dp[j + 1];

                }

            }

        }

        return dp[0];

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(n)

**Why we need "dp[j] = false;" in 1D DP solution?**

**2D DP Solution Logic:**

if(s1.charAt(i) == s3.charAt(i + j)) {

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

}

In the 2D DP solution, if the current characters in s1 and s3 match at the position (i, i + j), it implies that you can consider this character for an interleaving. The corresponding cell in the 2D array dp is set to the value in the cell below (i + 1, j). This is essentially saying, "If the current character matches, check the next character in s1 and continue with the result from below."

**Conversion to 1D DP Solution:**

if(s1.charAt(i) == s3.charAt(i + j)) {

dp[j] = dp[j];

} else {

dp[j] = false;

}

In the 1D DP solution, we represent the 2D array dp using a 1D array dp. The variable j now represents the column in the original 2D array. When the characters don't match (s1.charAt(i) != s3.charAt(i + j)), we set dp[j] to false. This is because if the current characters don't match, the interleaving is not possible, and we need to update the value in dp[j] to reflect this.

So, essentially, in the 1D DP solution, the dp[j] = false statement is handling the case where the current characters in s1 and s3 don't match at the position (i, i + j). If they don't match, it means we cannot continue with the interleaving, and we set dp[j] to false to indicate that.

**If no "dp[j] = false", we can test out by**

Input: s1 = "db", s2 = "b", s3 = "cbb"

Output: true, Expected: false

**Refer to**

<https://leetcode.com/problems/interleaving-string/solutions/2250195/java-4-solutions-recursion-memoization-dp/>

class Solution {

public boolean isInterleave(String s1, String s2, String s3) {

if(s1.length() + s2.length() != s3.length())

return false;

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

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

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

if (i == 0 && j == 0)

dp[j] = true;

else if (i == 0)

dp[j] = dp[j - 1] && s2.charAt(j - 1) == s3.charAt(i + j - 1);

else if (j == 0)

dp[j] = dp[j] && s1.charAt(i - 1) == s3.charAt(i + j - 1);

else

dp[j] = (dp[j] && s1.charAt(i - 1) == s3.charAt(i + j - 1))

|| (dp[j - 1] && s2.charAt(j - 1) == s3.charAt(i + j - 1));

}

}

return dp[s2.length()];

}

}