Edit Distance

Tuesday, November 16, 2021 11:47 AM

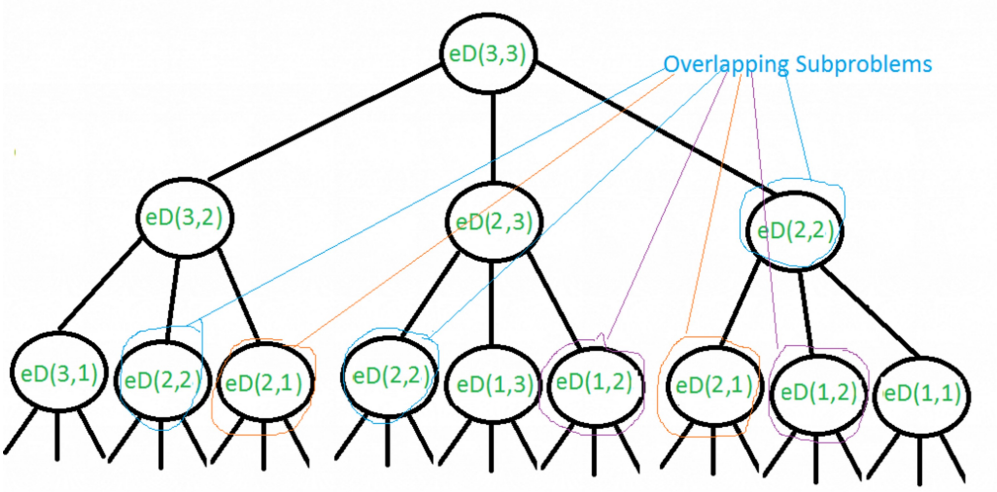
Recursive Solution (Containing both the code mine and)

```
packagedsaProblems;

publicclassedit_Distance{
publicstaticinteditDistance(Stringstr1,Stringstr2){

if(str1.length()==0)
returnstr2.length();
if(str2.length()==0)
returnstr1.length();

intm=str1.length();
intn=str2.length();

if(str1.charAt(m-1)==str2.charAt(n-1))
returneditDistance(str1.substring(0,m-1),str2.substring(0,n-1));

else{
return1+Math.min(
replace(str1,str2),
Math.min(
insert(str1,str2),remove(str1,str2)
)
);

}


}
publicstaticintedit_distance(Stringstr1,Stringstr2,intm,intn){

if(m==0)
returnn;
if(n==0)
```

```
return m;

if(str1.charAt(m-1)==str2.charAt(n-1))
return edit_distance(str1,str2,m-1,n-1);

else
{
return 1+Math.min(edit_distance(str1,str2,m,n-1),
Math.min(edit_distance(str1,str2,m-1,n),
edit_distance(str1,str2,m-1,n-1)));
}


}

private static int replace(String str1,String str2){
char strOne[]=str1.toCharArray();
char strTwo[]=str2.toCharArray();
int m=str1.length();
int n=str2.length();
strTwo[n-1]=strOne[m-1];
str2=new String(strTwo);
return editDistance(str1.substring(0,m-1),str2.substring(0,n-1));
}

private static int insert(String str1,String str2){
int m=str1.length();
int n=str2.length();

str2+=str1.charAt(m-1);
return editDistance(str1.substring(0,m-1),str2.substring(0,n));


}

private static int remove(String str1,String str2){
int m=str1.length();
int n=str2.length();
str2=str2.substring(0,n-1);
return editDistance(str1,str2);

}
```

```
publicstaticvoidmain(String[]args){

Stringstr1="sunday";
Stringstr2="saturday";
intm=str1.length();
intn=str2.length();

intoutput=editDistance(str1,str2);

intSecond_output=edit_distance(str1,str2,m,n);


System.out.println("Output:"+output);
System.out.println("Second_output:"+Second_output);
}
}
```

The time complexity of above solution is exponential. In worst case, we may end up doing $O(3^m)$ operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.



Worst case recursion tree when m = 3, n = 3.
Worst case example str1="abc" str2="xyz"

We can see that many subproblems are solved, again and again, for example, eD(2, 2) is called three times. Since same subproblems are called again, this problem has Overlapping Subproblems property. So Edit Distance problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by

constructing a temporary array that stores results of

From <https://www.geeksforgeeks.org/edit-distance-dp-5/>

subproblems.

- C++
- Java
- Python
- C#
- PHP
- Javascript

```
// A Dynamic Programming based Java program to find minimum
// number operations to convert str1 to str2
class EDIST {
    static int min(int x, int y, int z)
    {
        if (x <= y && x <= z)
            return x;
        if (y <= x && y <= z)
            return y;
        else
            return z;
    }

    static int editDistDP(String str1, String str2, int m,
                          int n)
    {
        // Create a table to store results of subproblems
        int dp[][] = new int[m + 1][n + 1];

        // Fill d[][] in bottom up manner
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                // If first string is empty, only option is
                // to insert all characters of second string
                if (i == 0)
                    dp[i][j] = j; // Min. operations = j

                // If second string is empty, only option is
                // to remove all characters of second string
                else if (j == 0)
                    dp[i][j] = i; // Min. operations = i

                // If last characters are same, ignore last
                // char and recur for remaining string
                else if (str1.charAt(i - 1)
                         == str2.charAt(j - 1))
                    dp[i][j] = dp[i - 1][j - 1];

                // If the last character is different,
                // consider all possibilities and find the
```

```
                // minimum
                else
                    dp[i][j] = 1
                               + min(dp[i][j - 1], // Insert
                                     dp[i - 1][j], // Remove
                                     dp[i - 1]
                                       [j - 1]); // Replace
            }
        }

        return dp[m][n];
    }

    // Driver Code
    public static void main(String args[])
    {
        String str1 = "sunday";
        String str2 = "saturday";
        System.out.println(editDistDP(
            str1, str2, str1.length(), str2.length()));
    }
} /*This code is contributed by Rajat Mishra*/
```

Output

3

Time Complexity: O(m x n)

Auxiliary Space: O(m x n)

Space Complex Solution: In the above-given method we require O(m x n) space. This will not be suitable if the length of strings is greater than 2000 as it can only create 2D array of 2000 x 2000. To fill a row in DP array we require only one row the upper row. For example, if we are filling the i = 10 rows in DP array we require only values of 9th row. So we simply create a DP array of 2 x str1 length. This approach reduces the space complexity. Here is the C++ implementation of the above-mentioned problem

From <https://www.geeksforgeeks.org/edit-distance-dp-5/>

```
A Space efficient Dynamic Programming
// based Java program to find minimum
// number operations to convert str1 to str2
import java.util.*;
class GFG
{

static void EditDistDP(String str1, String str2)
{
    int len1 = str1.length();
    int len2 = str2.length();

    // Create a DP array to memoize result
    // of previous computations
```

```
    int [][]DP = new int[2][len1 + 1];


    // Base condition when second String
    // is empty then we remove all characters
    for (int i = 0; i <= len1; i++)
        DP[0][i] = i;

    // Start filling the DP
    // This loop run for every
    // character in second String
    for (int i = 1; i <= len2; i++)
    {

        // This loop compares the char from
        // second String with first String
        // characters
        for (int j = 0; j <= len1; j++)
        {

            // if first String is empty then
            // we have to perform add character
            // operation to get second String
            if (j == 0)
                DP[i % 2][j] = i;

            // if character from both String
            // is same then we do not perform any
            // operation . here i % 2 is for bound
            // the row number.
            else if (str1.charAt(j - 1) == str2.charAt(i - 1)) {
                DP[i % 2][j] = DP[(i - 1) % 2][j - 1];
            }

            // if character from both String is
            // not same then we take the minimum
            // from three specified operation
            else {
                DP[i % 2][j] = 1 + Math.min(DP[(i - 1) % 2][j],
                                            Math.min(DP[i % 2][j - 1],
                                                DP[(i - 1) % 2][j - 1]));
            }
        }
    }

    // after complete fill the DP array
    // if the len2 is even then we end
    // up in the 0th row else we end up
    // in the 1th row so we take len2 % 2
    // to get row
    System.out.print(DP[len2 % 2][len1] +"\n");
}

// Driver program
public static void main(String[] args)
```

```
{
    String str1 = "food";
    String str2 = "money";
    EditDistDP(str1, str2);
}
}

// This code is contributed by aashish1995
```

Output

4

Time Complexity: O(m x n)

Auxiliary Space: O(m)

This is a memoized version of recursion i.e. Top-Down DP:

From <https://www.geeksforgeeks.org/edit-distance-dp-5/>

This is a memoized version of recursion i.e. Top-Down DP:

- C++14
- Java
- Python3
- C#
- Javascript

```
import java.util.*;
class GFG
{

static int minDis(String s1, String s2,
                  int n, int m, int[][]dp)
{

  // If any String is empty,
  // return the remaining characters of other String
  if(n == 0)
    return m;
  if(m == 0)
    return n;

  // To check if the recursive tree
  // for given n & m has already been executed
  if(dp[n][m] != -1)
    return dp[n][m];

  // If characters are equal, execute
  // recursive function for n-1, m-1
  if(s1.charAt(n - 1) == s2.charAt(m - 1))
  {
    if(dp[n - 1][m - 1] == -1)
    {
      return dp[n][m] = minDis(s1, s2, n - 1, m - 1, dp);
```

```
    }
    else
      return dp[n][m] = dp[n - 1][m - 1];
  }

  // If characters are nt equal, we need to

  // find the minimum cost out of all 3 operations.
  else
  {
    int m1, m2, m3;         // temp variables
    if(dp[n-1][m] != -1)
    {
      m1 = dp[n - 1][m];
    }
    else
    {
      m1 = minDis(s1, s2, n - 1, m, dp);
    }

    if(dp[n][m - 1] != -1)
    {
      m2 = dp[n][m - 1];
    }
    else
    {
      m2 = minDis(s1, s2, n, m - 1, dp);
    }

    if(dp[n - 1][m - 1] != -1)
    {
      m3 = dp[n - 1][m - 1];
    }
    else
    {
      m3 = minDis(s1, s2, n - 1, m - 1, dp);
    }
    return dp[n][m] = 1 + Math.min(m1, Math.min(m2, m3));
  }
}

// Driver program
public static void main(String[] args)
{

  String str1 = "voldemort";
  String str2 = "dumbledore";

  int n= str1.length(), m = str2.length();
  int[][] dp = new int[n + 1][m + 1];
  for(int i = 0; i < n + 1; i++)
  Arrays.fill(dp[i], -1);
  System.out.print(minDis(str1, str2, n, m, dp));
}
}
```

```
// This code is contributed by gauravrajput1
```

Output

7

Applications: There are many practical applications of edit distance algorithm, refer Lucene API for sample. Another example, display all the words in a dictionary that are near proximity to a given wordincorrectly spelled word.

From <https://www.geeksforgeeks.org/edit-distance-dp-5/>