Dynamic Programming III

SWE2016-44

Edit Distance

Edit Distance

Given two strings $string_1$ and $string_2$ and below operations that can be performed on $string_1$. Find minimum number of operations or edits required to convert $string_1$ to $string_2$.

1. **Insert**
2. **Delete**
3. **Modify**

Assumption: All of the operations are of equal cost.

Edit Distance

Example 1: Input: $string_1$ = “geek” and $string_2$ = “gesek”

Output: 1

We can convert $string_1$ to $string_2$ by inserting ‘s’ in $string_1$.

Edit Distance

Example 2: Input: $string_1$ = “cat” and $string_2$ = “cut”

Output: 1

We can convert $string_1$ to $string_2$ by modifying ‘a’ to ‘u’.

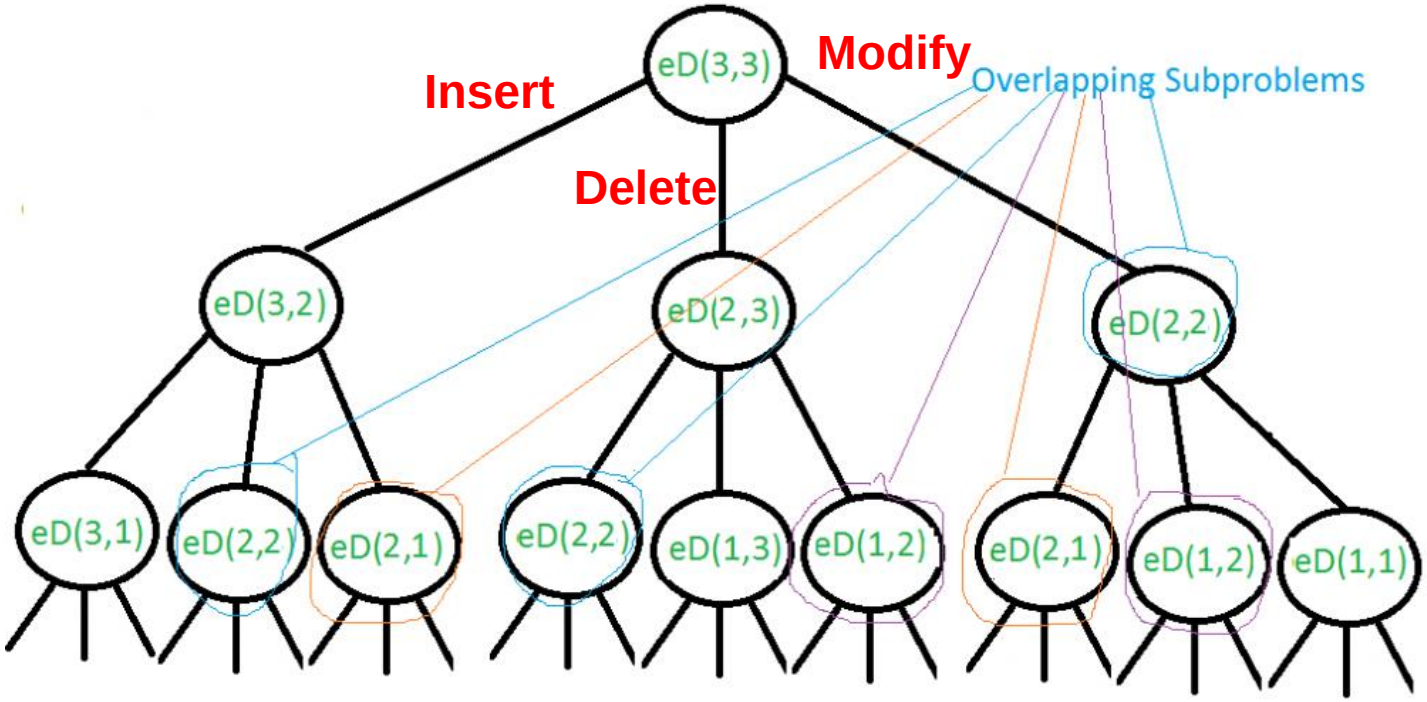
Edit Distance

Approach: Given two strings of length m and n

1. **If the last characters of two strings match, we do not change anything and recur for length m-1 and n-1.**

2. **Else we compute minimum cost of all three operations (insert, delete and modify) and take minimum of these three values.**
   a. **Insert: Recur for m and n-1**
   b. **Delete: Recur for m-1 and n**
   c. **Modify: Recur for m-1 and n-1**

Overlapping Subproblems Property



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

Dynamic Programming

Given two strings “CART” and “MARCH” find its Edit Distance

To convert “CART” into “MARCH” we need to perform 3 operations on string “CART”.

1. **Modify ‘c’ to ‘m’**
2. **Modify ‘t’ to ‘c’**
3. **Insert ‘h’**

Dynamic Programming

editDist	ø	M	A	R	C	H
ø						
C						
A						
R						
T						

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0					
C						
A						
R						
T						

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1				
C						
A						
R						
T						

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2			
C						
A						
R						
T						

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C						
A						
R						
T						

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1					
A						
R						
T						

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1					
A	2					
R						
T						

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1					
A	2					
R	3					
T	4					

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1	1				
A	2					
R	3					
T	4					

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1	1	2			
A	2					
R	3					
T	4					

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1	1	2	3		
A	2					
R	3					
T	4					

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1	1	2	3	3	
A	2					
R	3					
T	4					

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1	1	2	3	3	
A	2					
R	3					
T	4					

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1	1	2	3	3	4
A	2					
R	3					
T	4					

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1	1	2	3	3	4
A	2	2				
R	3					
T	4					

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1	1	2	3	3	4
A	2	2	1			
R	3					
T	4					

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1	1	2	3	3	4
A	2	2	1	2	3	4
R	3	3	2	1	2	3
T	4	4	3	2	2	3

Dynamic Programming

editDist	ø	M	A	R	C	H
ø	0	1	2	3	4	5
C	1	1	2	3	3	4
A	2	2	1	2	3	4
R	3	3	2	1	2	3
T	4	4	3	2	2	3

Recursive Implementation

```
t editDist(string str1 , string str2 , int m ,int n)

  // If first string is empty, the only option is to
  // insert all characters of second string into first
  if (m == 0) return n;

  // If second string is empty, the only option is to
  // remove all characters of first string
  if (n == 0) return m;

  // If last characters of two strings are same, nothing
  // much to do. Ignore last characters and get count for
  // remaining strings.
  if (str1[m-1] == str2[n-1])
      return editDist(str1, str2, m-1, n-1);

  // If last characters are not same, consider all three
  // operations on last character of first string, recursively
  // compute minimum cost for all three operations and take
  // minimum of three values.
  return 1 + min ( editDist(str1,  str2, m, n-1),    // Insert
                   editDist(str1,  str2, m-1, n),   // Remove
                   editDist(str1,  str2, m-1, n-1) // Replace
                 );
```

→ **Time Complexity: $O(3^m)$**

Dynamic Programming

```
int dp[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[i-1] == str2[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];
```

→ **Time Complexity: O(mn)**

Longest Palindromic Subsequence

Longest Palindromic Subsequence

Given a sequence, find the length of the longest palindromic subsequence in it.

For example,

Input: BBABCBCAB

Output: 7 (“BABCBAB”)

Longest Palindromic Subsequence

Let's take the following string

BABCBAB

0 1 2 3 4 5 6

Let's say L(i, j) represents the length of longest palindromic subsequence in a string from index i to j

Thus, L(0, 6) will present the length of 'LPS' for the above string (LPS: Longest Palindromic Subsequence)

Longest Palindromic Subsequence

BABCBAB

If characters at 0th and 6th index are same

$$L(0, 6) = L(1, 5) + 2$$

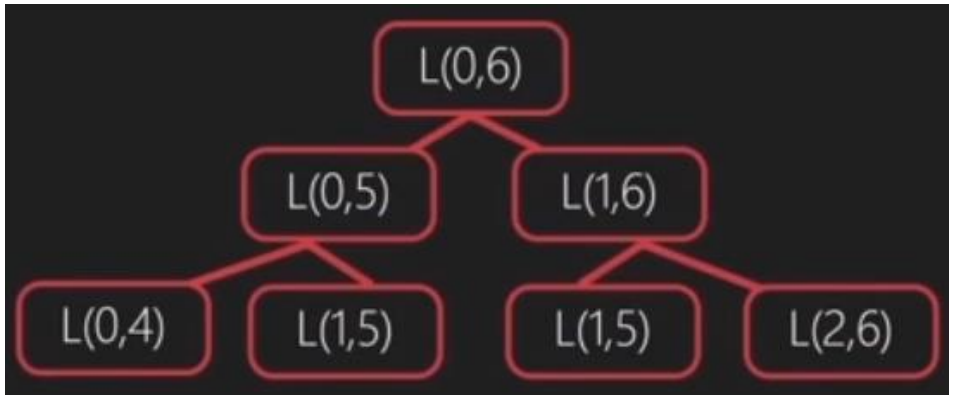
Otherwise,

$$L(0, 6) = \max(L(0, 5), L(1, 6))$$

Overlapping Subproblems Property

BABCBAB

So, our solution will follow the following pattern when implemented recursively



As apparent, there are overlapping subproblems which increase the time complexity of our solution

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i j

Now, for each value of i we will have a row and for each value of j we will have a column

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i j

We will move i and j such that both stay at a constant distance while traversing the string

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i

j

Characters covered by i and j: 1

LPS for 1 character will be 1
thus, L[i][i]=1 for each i in [0, 6]

	0	1	2	3	4	5	6
0							
1							
2							
3							
4							
5							
6							

L[6][6]

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i

j

Characters covered by i and j: 1

LPS for 1 character will be 1
thus, L[i][i]=1 for each i in [0, 6]

	0	1	2	3	4	5	6
0	1						
1							
2							
3							
4							
5							
6							

L[6][6]

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i

j

Characters covered by i and j: 1

LPS for 1 character will be 1
thus, L[i][i]=1 for each i in [0, 6]

	0	1	2	3	4	5	6
0	1						
1		1					
2							
3							
4							
5							
6							

L[6][6]

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

Characters covered by i and j: 1

LPS for 1 character will be 1
thus, L[i][i]=1 for each i in [0, 6]

	0	1	2	3	4	5	6
0	1						
1		1					
2			1				
3							
4							
5							
6							

L[6][6]

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

Characters covered by i and j: 1

LPS for 1 character will be 1
thus, L[i][i]=1 for each i in [0, 6]

	0	1	2	3	4	5	6
0	1						
1		1					
2			1				
3				1			
4					1		
5						1	
6							1

L[6][6]

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i j

Characters covered by i and j: 2

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

	0	1	2	3	4	5	6
0	1						
1		1					
2			1				
3				1			
4					1		
5						1	
6							1

L[6][6]

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i j

Characters covered by i and j: 2

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

	0	1	2	3	4	5	6
0	1	1					
1		1					
2			1				
3				1			
4					1		
5						1	
6							1

L[6][6]

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

	0	1	2	3	4	5	6
0	1	1					
1		1	1				
2			1				
3				1			
4					1		
5						1	
6							1

L[6][6]

Characters covered by i and j: 2

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise, L[i][j]=max(L[i][j-1], L[i+1][j])

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

	0	1	2	3	4	5	6
0	1	1					
1		1	1				
2			1	1			
3				1	1		
4					1	1	
5						1	1
6							1

L[6][6]

Characters covered by i and j: 2

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i j

Characters covered by i and j: 3

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

	0	1	2	3	4	5	6
0	1	1					
1		1	1				
2			1	1			
3				1	1		
4					1	1	
5						1	1
6							1

L[6][6]

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i j

	0	1	2	3	4	5	6
0	1	1	3				
1		1	1				
2			1	1			
3				1	1		
4					1	1	
5						1	1
6							1

L[6][6]

Characters covered by i and j: 3

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i j

	0	1	2	3	4	5	6
0	1	1	3				
1		1	1	1			
2			1	1			
3				1	1		
4					1	1	
5						1	1
6							1

L[6][6]

Characters covered by i and j: 3

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i j

Characters covered by i and j: 3

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

	0	1	2	3	4	5	6
0	1	1	3				
1		1	1	1			
2			1	1	3		
3				1	1		
4					1	1	
5						1	1
6							1

L[6][6]

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

i j

	0	1	2	3	4	5	6
0	1	1	3				
1		1	1	1			
2			1	1	3		
3				1	1	1	
4					1	1	
5						1	1
6							1

L[6][6]

Characters covered by i and j: 3

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

	0	1	2	3	4	5	6
0	1	1	3				
1		1	1	1			
2			1	1	3		
3				1	1	1	
4					1	1	3
5						1	1
6							1

L[6][6]

Characters covered by i and j: 3

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

	0	1	2	3	4	5	6
0	1	1	3	3			
1		1	1	1	3		
2			1	1	3	3	
3				1	1	1	3
4					1	1	3
5						1	1
6							1

L[6][6]

Characters covered by i and j: 4

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

	0	1	2	3	4	5	6
0	1	1	3	3	3		
1		1	1	1	3	5	
2			1	1	3	3	3
3				1	1	1	3
4					1	1	3
5						1	1
6							1

L[6][6]

Characters covered by i and j: 5

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

	0	1	2	3	4	5	6
0	1	1	3	3	3	5	
1		1	1	1	3	5	5
2			1	1	3	3	3
3				1	1	1	3
4					1	1	3
5						1	1
6							1

L[6][6]

Characters covered by i and j: 6

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

	0	1	2	3	4	5	6
0	1	1	3	3	3	5	7
1		1	1	1	3	5	5
2			1	1	3	3	3
3				1	1	1	3
4					1	1	3
5						1	1
6							1

L[6][6]

Characters covered by i and j: 7

If characters at i and j are same, then L[i][j]=L[i+1][j-1] + 2

Otherwise,
L[i][j]=max(L[i][j-1], L[i+1][j])

Dynamic Programming

BABCBAB

0 1 2 3 4 5 6

L[i][j] gives us ‘LPS’ for the string from index i to j

L[0][5] will give us ‘LPS’ for ‘BABCBA’

L[0][5] = 5

	0	1	2	3	4	5	6
0	1	1	3	3	3	5	7
1		1	1	1	3	5	5
2			1	1	3	3	3
3				1	1	1	3
4					1	1	3
5						1	1
6							1

L[6][6]

Dynamic Programming

```
int lps(char *str)
{
   int n = strlen(str);
   int i, j, cl;
   int L[n][n];  // Create a table to store results of subproblems


   // Strings of length 1 are palindrome of lentgh 1
   for (i = 0; i < n; i++)
      L[i][i] = 1;

    // Build the table. Note that the lower diagonal values of table are
    // useless and not filled in the process. The values are filled in a
    // manner similar to Matrix Chain Multiplication DP solution (See
    // https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/). cl is length of
    // substring
    for (cl=2; cl<=n; cl++)
    {
        for (i=0; i<n-cl+1; i++)
        {
            j = i+cl-1;
            if (str[i] == str[j] && cl == 2)
               L[i][j] = 2;
            else if (str[i] == str[j])
               L[i][j] = L[i+1][j-1] + 2;
            else
               L[i][j] = max(L[i][j-1], L[i+1][j]);
        }
    }

    return L[0][n-1];
}
```

→ **Time Complexity: $O(n^2)$**

Reference

- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms”,* September 29, 2004

- https://www.geeksforgeeks.org