Longest Common Subsequence | Introduction & LCS Length

The longest common subsequence (LCS) problem is the problem of finding the longest subsequence that is present in given two sequences in the same order. i.e. find a longest sequence which can be obtained from the first original sequence by deleting some items, and from the second original sequence by deleting other items.

The problem differs from problem of finding common substrings. Unlike substrings, subsequences are not required to occupy consecutive positions within the original

Browse

Adobe Algorithm Amazon BFS Binary Search Bit Hacks DFS FIFO Google Greedy Hashing Intro JSON LCS LIFO Maze Memoized Microsoft Must Know Priority Queue Probability Recursive Searching Sliding Window Tabulation Tricky Trie

```
X: ABCBDAB
Y: BDCABA


The length of LCS is 4
LCS are BDAB, BCAB and BCBA
```

A Naive solution would be to check if every subsequence of `X[1..m]` to see if it is also a subsequence of `Y[1..n]`. As there are 2^m subsequences possible of X, the complexity of this solution would be $O(n.2^m)$.

The LCS problem has an optimal substructure. That means the problem can be broken down into smaller, simple "subproblems", which can be broken down into yet simpler subproblems, and so on, until, finally, the solution becomes trivial.

1. Let us consider two sequences X and Y of length m and n that both end in the same element.

    To find their LCS, shorten each sequence by removing the last element, find the LCS of the shortened sequences, and to that LCS append the removed element. So we can say that

```
1]) + X[m]   if X[m] =
Y[n]
```

2. Now suppose that the two sequences do not end in the same symbol.

Then the LCS of X and Y is the longer of the two sequences `LCS(X[1..m-1], Y[1..n])` and `LCS(X[1..m], Y[1..n-1])`. To understand this property, let's consider the two following sequences

```
X: ABCBDAB (n elements)
Y: BDCABA  (m elements)
```

The LCS of these two sequences either ends with a B (the last element of sequence X) or does not.

Case 1: If LCS ends with a B, then it cannot end with a A and we can remove the A from sequence Y and the problem reduces to `LCS(X[1..m], Y[1..n-1])`.

Case 2: If LCS does not end with a B, then we can remove B from the sequence X and the problem reduces to `LCS(X[1..m-1], Y[1..n])`. For example,

```
                    LCS(
‘ABCBDAB’, ‘BDCAB’))

LCS(‘ABCBDA’,
‘BDCABA’)
= LCS(‘ABCBD’,
‘BDCAB’) + ‘A’
LCS(‘ABCBDAB’,
‘BDCAB’) =
LCS(‘ABCBDA’,
‘BDCA’) + ‘B’

LCS(‘ABCBD’,
‘BDCAB’) = maximum
(LCS(‘ABCB’,
‘BDCAB’),
LCS(‘ABCBD’,
‘BDCA’))
LCS(‘ABCBDA’,
‘BDCA’) =
LCS(‘ABCBD’, ‘BDC’)
+ ‘A’

and so on..
```

Below solution finds the length of LCS of sequences `X[0..m-1]` and `Y[0..n-1]` recursively by using optimal substructure property of LCS problem.

C++

```
#include <iostream>
#include <string>
using namespace std;

// Function to find len
// sequences X[0..m-1]
int LCSLength(string X,
{
    // return if we hav
```

```
16
17      // else if last cha
18      return max(LCSLengt
19  }
20
21  // Longest Common Subse
22  int main()
23  {
24      string X = "ABCBDAB
25
26      cout << "The length
27              LCSLength(X
28
29      return 0;
30  }
```

Downloa d Run Code

Output:

```
The length of LCS is 4
```

Java

```
1   class LCS
2   {
3       // Function to find
4       // sequences X[0..m
5       public static int L
6       {
7           // return if we
8           if (m == 0 || n
9               return 0;
10          }
11
12          // if last char
13          if (X.charAt(m
14              return LCSL
15          }
16
17          // else if last
18          return Integer.
19
20      }
21
22      // main function
23      public static void
24      {
25          String X = "ABC
```

Downloa d Run Code

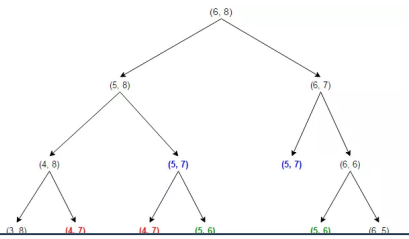
Output:

```
The length of LCS is 4
```

The worst case time complexity of above solution is $O(2^{(m+n)})$. The worst case happens when there is no common subsequence present in X and Y (i.e. LCS is `0`) and each recursive call will end up in two recursive calls.

The LCS problem exhibits overlapping subproblems. A problem is said to have overlapping subproblems if the recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.

Let us consider recursion tree for two sequences of length 6 and 8 whose LCS is `0`.



As we can see, the same sub-problems (highlighted in same color) are getting computed again and again. We know that problems having optimal substructure and overlapping subproblems can be solved by dynamic programming, in which subproblem solutions are memoized rather than computed again and again. This method is illustrated below in C++ and Java –

C++

```
#include <iostream>
#include <string>
#include <unordered_map
using namespace std;

// Function to find len
// X[0..m-1] and Y[0..n
int LCSLength(string X,
{
    // return if we hav
    if (m == 0 || n ==
        return 0;

    // construct a uniq
    string key = to_str

    // if sub-problem i
    // store its result
    if (lookup.find(key
    {
        // if last char
        if (X[m - 1] ==
            lookup[key]

        else
        // else if last
        lookup[key] = m

    }

    // return the subpr
    return lookup[key];
}

```

```
    unordered_map<strin

    cout << "The length
        << LCSLength(X,

    return 0;
}
```

Downloa d Run Code

Output:

```
The length of LCS is 4
```

Java

```
import java.util.HashMa
import java.util.Map;

class LCS
{
    // Function to find
    // X[0..m-1] and Y[
    public static int L

    {
        // return if we
        if (m == 0 || n
            return 0;

        // construct a
        String key = m

        // if sub-probl
        // store its re
        if (!lookup.con
        {
            // if last
            if (X.charA
                lookup.

            }
            else {
                // else
                lookup.

            }
        }

        // return the c
```

```
40      {
41          String X = "ABC
42
43          // create a map
44          Map<String, Int
45
46          System.out.prin
47                  + LCSLe
48      }
49  }
```

Downloa d Run Code

Output:

```
The length of LCS is 4
```

The time complexity of above solution is `O(mn)` and auxiliary space used by the program is `O(mn)`. Note that we can also use array instead of map. Check implementation here.

Above *Memo*ized version follows the top-down approach, since we first break the problem into subproblems and then calculate and store values. We can also solve this problem in bottom-up manner. In the bottom-up approach, we calculate the smaller values of `LCS(i, j)` first, then build larger values from them.

```
1] +
1                       if
X[i-1] == Y[j-1]
            | longest(LCS[i
- 1][j], LCS[i][j -
1])      if X[i-1] != Y[j-1]
```

Let X be "`XMJYAUZ`" and Y be "`MZJAWXU`". The longest common subsequence between X and Y is "`MJAU`". The table below is generated by the function `LCSLength`, shows the lengths of the longest common subsequences between prefixes of X and Y. The ith row and jth column shows the length of the LCS of substring `X[0..i-1]` and `Y[0..j-1]`.

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	1	1	1	1	1	1	1
3	J	0	1	1	2	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	3	3	3	3
6	U	0	1	1	2	3	3	3	**4**
7	Z	0	1	2	2	3	3	3	**4**

C++

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
```

```
10
11      // lookup table sto
12      // i.e. lookup[i][j
13      // X[0..i-1] and Y[
14      int lookup[m + 1][n
15
16      // first column of
17      for (int i = 0; i <
18          lookup[i][0] =
19
20      // first row of the
21      for (int j = 0; j <
22          lookup[0][j] =
23
24      // fill the lookup
25      for (int i = 1; i <
26      {
27          for (int j = 1;
28          {
29              // if curre
30              if (X[i - 1
31                  lookup[
32
33              // else if
34              else
35                  lookup[
36          }
37      }
38
39      // LCS will be last
40      return lookup[m][n]
41  }
42
43  // Longest Common Subse
44  int main()
45  {
46      string X = "XMJYAUZ
47
48      cout << "The length
49
50      return 0;
51  }
```

Download Run Code

Output:

```
The length of LCS is 4
```

Java

```
5       public static int L
6       {
7           int m = X.lengt
8
9           // lookup table
10          // i.e. T[i][j]
11          // X[0..i-1] an
12          int[][] T = new
13
14          // fill the loo
15          for (int i = 1;
16          {
17              for (int j
18              {
19                  // if c
20                  if (X.c
21                      T[i
22                  }
23                  // else
24                  else {
25                      T[i
26                  }
27              }
28          }
29
30          // LCS will be
31          return T[m][n];
32      }
33
34      // main function
35      public static void
36      {
37          String X = "XMJ
38
39          System.out.prin
40      }
41  }
```

Downloa d Run Code

Output:

```
The length of LCS is 4
```

The time complexity of above solution is `O(mn)` and auxiliary space used by the program is

only the solutions to the current row and the previous row.

Applications of LCS problem:

The longest common subsequence problem forms the basis of data comparison programs such as the diff utility and use in field of bioinformatics. It is also widely used by revision control systems such as Git.

Exercise:

1. Extend the solution for finding length of LCS for K-sequences

2. Write space optimized code for iterative version.

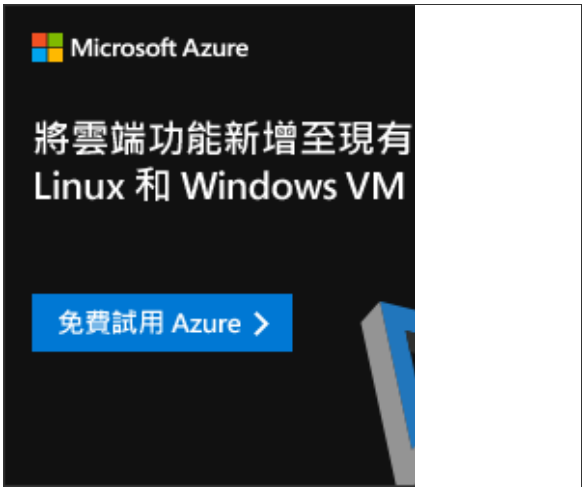
Recommended Read: Longest Common Subsequence (Finding all LCS)

References:
https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

(**16** votes, average: **5.00** out of 5)

Thanks for reading.

and help us grow. Happy coding
🙂



Dynamic Programming, String
Algorithm, Amazon, LCS, Memoized, Recursive, Tabulation

← Introduction to Dynamic Programming

Longest Common Subsequence (LCS) | Space optimized version →

Leave a Reply

b i link b-quote u
ul ol li code
spoiler

Join the

▲ newest ▲ oldest ▲ voted

Geek

Guest

Good post!

4 Reply

2 years ago

md. saiful islam

Guest

really helpful

2 Reply

10 months ago

mohan

Guest

good

2 Reply

2 months ago

Anne

Guest

Thanks! This helped me to get started with dynamic programming.

1 Reply

1 year ago

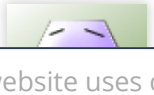
Vik

Guest

Nice

1 Reply

8 months ago



Rahul

dynamic programming.

1 Reply

5 months ago

Andre

Guest

Please make your page copyable so I and people like me can read it with my screen reader.

1 Reply

5 months ago

Andy

Guest

It would of course be more appropriate if they actually did that (since keeping text from being selectable is simply ridiculous ...) but in the meantime, you could install this: https://greasyfork.org/en/scripts/26305-

by-css/code

1

Reply

4 months ago

Guest

hsw

mg very useful extension, ,, thanks

0

Reply

4 days ago

Guest

Ovi Poddar

it's an amazing platform to learn dp ! loved it !

1 Reply

1 month ago

Guest

congtrinh0509

awesome

1 Reply

18 days ago

Guest

Would you like an implementation in Python?

0 Reply

3 days ago

Techie Delight

Author

Sure. You can either share your code here or contact us using this link. Thanks.

0 Reply

3 days ago