Timothy Young

Title: *LCS Algorithm Analysis*

Dynamic vs Brute Force, CS-350

Table of Contents

1 Intro

1.1 Abstract

Finding the LCS(Longest Common Substring), has a wide variety of applications and supporting theoretical assumptions/pseudocode to be ran under empirical analysis. This report found that for small input sizes, the Brute Force LCS algorithm experienced a faster runtime. When these input sizes were increased significantly, an exponential increase in the runtime was experienced for the BF LCS. For these large input sizes(n=24), the Dynamic Programming LCS algorithm was 169% faster than the BF LCS algorithm.

1.2 Introduction

Dynamic programming is a popular topic in the computer science industry, along with pattern matching algorithms such as the LCS. The vast amounts of documentation on this problem in particular makes it fairly appealing to hold under empirical analysis. Using Python 3.8, a BF LCS algorithm and a DP LCS algorithm for finding the length of the longest common subsequence was implemented. Modules like the Timeit library allowed us to record some experimental values for the runtime experienced in the Python. A main test file in the Repl.it IDE environment imports the two algorithms implemented in Python, testing them against each other with results found later in the report.

1.3 Background

The contents of this section covers the known theoretical complexities of the brute force LCS and the DP LCS algorithms. Further optimizations will be covered later in the report within section 7.2.

1.3.1 Brute Force Theoretical Complexities

The brute force LCS requires space on the stack to perform its functionality for finding the length of the LCS. Listed are the relevant time complexities, as well as a recursion tree supporting the number of calls made to the stack:

1.3.1.1 Time

Base Case(Default Best Case Scenario): A string or sequence of either of the sequences(x,y) is equal to 1 and is therefore already the longest subsequence when considering distinct sequences x,y. This will just evaluate once(constant):

$$Brute\ Force\ Base\ Case: O(mn) = O(1)$$

Average Case: There is a common subsequence present in either of the sequences x,y. This will still double the number of recursive calls to check for the longest common subsequence, but only up to the length of the smallest sequence. This will have a time complexity of:

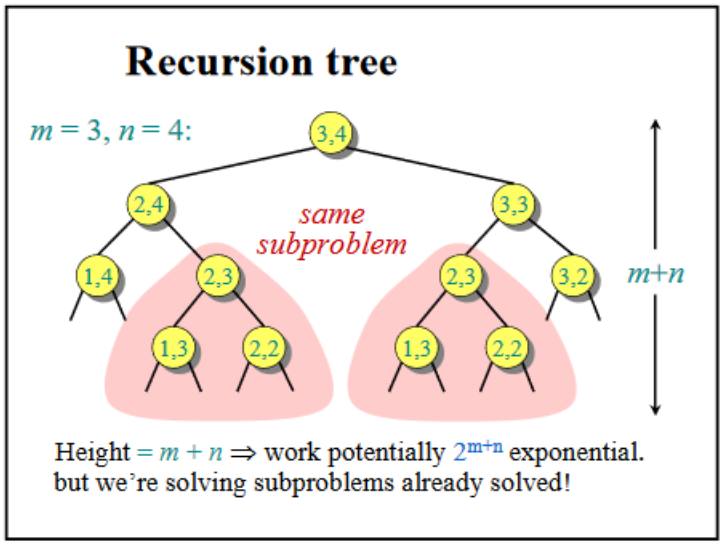
$$Brute\ Force\ Worst\ Case: O(mn) = O(2^{m+n}) = O(2^n)$$

Worst Case: There is no common subsequence present in either of the sequences x,y. This will double the number of recursive calls to check for the longest common subsequence. This will have a time complexity of:$Brute\ Force\ Avg.Case: O(mn) = O(2^n)\ O\ (2^m) = O(2^{mn})\ or\ O(2^{n^2})$

Ronald I. Greenberg (2003-08-06). "Bounds on the Number of Longest Common Subsequences". arXiv:cs.DM/0301030.

1.3.1.2 Space

The BF algorithm utilizes recursion to solve every subproblem *i,j* corresponding to lengths *m,n.* The space complexity of recursive algorithms is typically analyzed as a tree, with the depth of the tree aligning with the number of calls to the stack. In this case, the depth of the tree is equal to m+n:



$$Brute\ Force\ Space\ Complexity: O(mn) = O(m + n)$$

Charles Leiserson, https://www2.cs.arizona.edu/classes/cs545/fall09/Dynamic-Prog.prn.pdf

1.3.2 Dynamic Programming Theoretical Complexities

The DP algorithm to find the length of the LCS requires allocating space for a matrix to hold the subsequences generated. Listed are relevant time and space complexities:

1.3.2.1 Time

Average Case Runtime: Given that this implementation is optimized to find the length of the LCS, the dynamic programming approach has an average runtime of:

$$DP\ Avg.Case: O(mn) = O(m * n)\ .$$

This is supported by Wagner/Fischer's "string-to-correction-problem" presented in the Journal of the ACM. *"The string-to-string correction problem". Journal of the ACM. **21** (1): 168–173. CiteSeerX 10.1.1.367.5281. doi:10.1145/321796.321811*

1.3.2.2 Space

This algorithm requires allocating space for a matrix to hold the calculated subsequences for each element within distinct sequences x,y.

$$DP\ Avg.Space\ Complexity: O(mn) = O(m * n)\ .$$

2 Test Plan

2.1 Data Analysis Process or Procedure

Python 3.8 has a library available named Timeit. This library provides a way to run a function a number of times(n), and returns the minimum amount of time after running the testable function n times. This is important because the runtime environment of the testing suite(Repl.it), is not static. A better approach for data analysis is to run the function a large number of times to record the perceived minimum value. This will ensure better accuracy than simply calculating the difference or average of values recorded using the Time library.

When analyzing the DP algorithm for finding the LCS, the average case can be viewed as the worst-case. Therefore, if we are comparing the two algorithms for data analysis, it would be appropriate to compare their worst-case time complexities using the Timeit library and Python 3.8.

Furthermore, we can also compare the theoretical complexities of the BF & DP algorithms vs their experimental timing values recorded using the Timeit library. Using the theoretical values as a control can tell us more about the actual implementations in Python 3.8, allowing us to analyze the best-worst case scenarios These values will be calculated and recorded in a table, and then ran through empirical analysis in section 3.2.

Utilizing a TI-84, I calculated percent differences between the experimental values recorded in this test analysis.
Formula for calculating the percent difference between the cases:

$$\%\, difference = \frac{|x2 - x1|}{\frac{x2 + x1}{2}} * 100\%$$

2.2 Test Cases:

These are test cases for simple inputs and bases cases when comparing the two algorithms against each other and their known theoretical values. These could be taken further with a larger variation in inputs to really test the implementation.

2.2.1 Best Case

It may be suitable to analyze the abstract base case for the DP algorithm as a best-case scenario. This would imply that one of the inputs is equal to length 1, with its one element being a subsequence of the larger sequence.

Inputs:

xBest = ['A','B']

yBest = ['A']

2.2.2 Average Case

This average/worst case input for the DP algorithm in finding the LCS does not differ much in terms of complexity. The BF algorithm however has a much faster time complexity for the average case when compared to the worst case. For the average case, there is a subsequence between the two sequences.

//First Trial Input:

xAvg1 = ['A','G','G','A','T']

yAvg1 = ['G','A','C']

//Increasing Input Analysis:

xAvg = ['A','G','G']

yAvg = ['G','A']

xAvg2 = ['A','G','G','A','G','T']

yAvg2 = ['G','A','G','A']

xAvg4 = ['A','G','G','A','G','T','A','G','G','A','G','T']

yAvg4 = ['G','A','G','A','G','A']

xAvg3 = ['A','G','G','A','G','A','G','A','T','A','G','T','A','G','G','A','G','A','G','A','T','A','G','T']

yAvg3 = ['G','A','G','A','G','A','G','A','T','G','A','G','A','G','A','G','A','T']

2.2.3 Worst Case

This worst-case input does not alter the time complexity of the DP algorithm for finding the LCS when compared to the listed average case, but it does have an affect on the BF algorithm. When a subsequence is not shared between the two sequences, the BF algorithm is required to make twice as many recursive calls and essentially double the work.

Inputs:

xWorst = ['A','G']

yWorst = ['T','C']

2.2.4 Worst Case Analysis(Increasing Inputs):

We can compare the worst case-scenario for both algorithms in which neither sequence shares a subsequence. Furthermore, the time complexity should grow exponentially as stated previously from the known theoretical time complexities for these algorithms. If m=n, then the DP algorithm should grow by:

$$DP(m = n)\text{: } O(mn) = O(m * n) = O(n * n) = O(n^2)$$

The BF algorithm will grow at a greater exponential rate:

$$BF(m = n)\text{: } O(mn) = O(2^{n*n}) \; or \; O(2^{n^2})$$

Inputs:

xWorst = ['A','G']

yWorst = ['T','C']

xWorst2 = ['A','G','G','A']

yWorst2 = ['T','C','C','T']

xWorst3 = ['A','G','G','A','G','A','G','A']

yWorst3 = ['T','C','C','T','C','T','C','T']

2.3Analysis Results

2.3.1 BF vs DP (Best-Worst Cases)

Small Inputs**	Best Case	Average Case	Worst Case
BF	0.0104 fs	0.2922 fs	1.2897 fs
DP	0.1008 fs	0.3244 fs	0.4091 fs
% Difference	882%	10.4%	103%

For the average case, the timing is relatively close within 10%. If we were to increase the input size for one sequence variable in the average case, we would likely see an increase in this calculation. The value of 882% is far too large to be considered for empirical analysis. The percent difference calculated from the worst-case scenarios for the timing complexity follows our theoretical assumptions.

We should see a huge increase in the amount of time taken to compute the length of the LCS given this theoretical assumption. These small input values ran in this analysis are not significant enough to see the superior average case time complexity for the DP LCS algorithm. Please see 2.3.3.

While the value of 882% is far too large, it does follow our theoretic assumption:

$$O(Best\ Case) \rightarrow BF\varepsilon\ O(1) < DP\ \varepsilon\ O(n * m)$$

2.3.2 Worst-Case Analysis(Changing Input Size):

(m=n)	Case1(n =2)	Case2(n=4)	Case3(n=8)
BF	1.2897	2.2081	331.92
DP	0.1327	0.4905	1.2093
Theoretical BF $O(2^{n^2})$	(2^4) x8	(2^{16}) x65536	(2^{64}) x1.8*10^19
Theoretical DP $O(n^2)$	(2^2) x4	(4^2) x16	(8^2) x64

Given increasing input sizes of n, the functions are obviously growing exponentially. Very large sizes of n however require immense amounts of time and begin to lose accuracy within the testing environment for the BF algorithm. This agrees theoretically as it grows at a huge exponential rate, much greater than that of n^2. This worst-case scenario begins to significantly impact runtime speed after inputs greater than n=2.

$$O(m = n) \rightarrow DP\ \varepsilon\ O(n^2) < BF\ \varepsilon\ O(2^{n^2})$$

2.3.3 Average-Case Analysis(Changing Input Size):

	(n = 3, m = 2)	(n = 6, m = 2)	(n = 12, m = 6)	(n = 24, m = 18)
BF	.1210	.2803	.7294	83.80
DP	.2292	.6169	1.829	8.198
% diff	61.7%	74.9%	85.9%	%164

As seen in section 2.3.1, for relatively small input sizes of n, the average time complexity of the two algorithms is fairly close depending on the state of the testing environment. If the size of inputs n, m are greatly increased, a huge speed-up in runtime is experienced when comparing the DP LCS vs the BF LCS algorithms. A similar run not listed in this report increased the sizes of n while keeping m small. A significant change in runtime speed was not experienced, but notably increasing both n, m caused the BF runtime to take 164% longer than the DP LCS. Theoretically, there should be an exponential increase as we increase input sizes.

$$O(n \geq m) \rightarrow DP\ \varepsilon\ O(m * n) < BF\ \varepsilon\ O(n^{n+m})$$

3 Conclusion

3.1 Recommendations

Testing the input sizes for experimental runtime values comparable between the DP & BF LCS algorithms was an insightful exercise. The theoretical time complexities that have been discussed are not significant for small input sizes. Furthermore, when you start to increase the size of n past a certain threshold(n =12 for example), the total run time has a significant increase in the BF LCS algorithm. It grows very quickly, obviously at an exponential rate as outlined by the theoretical assumptions for time complexity. Finding patterns and subsequences has a large number of applications, so it is relevant to understand all of these applications and when certain algorithms are suited better. Overall, for small input sizes of n, the brute force algorithm experienced a faster runtime. For large input sizes of n, the DP algorithm experienced a faster runtime.

3.2 Further Optimizations

“If only the length of the LCS is required, the matrix can be reduced to a 2xmin(n,m) matrix with ease, or to a min(m,n)+1 vector (smarter) as the dynamic programming approach only needs the current and previous columns of the matrix. Hirschberg's algorithm allows the construction of the optimal sequence itself in the same quadratic time and linear space bounds.”” *Hirschberg, D. S. (1975). "A linear space algorithm for computing maximal common subsequences". Communications of the ACM. **18** (6): 341–343. doi:10.1145/360825.360861*

4 Figures

4.1Appendix 1

4.1.1 BF vs DP (Best-Worst Cases):

```
Best/Base Case Scenarios:
-------------------------
BF Best Case Length:  1
Best Case Brute Force LCS Time:  0.010462262998771621 fractional seconds


Dynamic LCS Best Case Length:  1
Best Case Dynamic LCS Time:  0.10081429300043965 fractional seconds
-----------------------------------------------------------------
Average Case Scenarios:
-------------------------


BF Avg. Case Length:  2
Avg Case Brute Force LCS Time:  0.2922076510003535 fractional seconds


Dynamic LCS Avg. Case Length:  2
Avg. Case Dynamic LCS Time:  0.32442581899886136 fractional seconds
-----------------------------------------------------------------
Worst-Case Scenarios:
-------------------------


BF Worst Case Length:  0
Worst Case Brute Force LCS Time:  1.2897551170026418 fractional seconds


Dynamic LCS Worst Case Length:  0
Worst Case Dynamic LCS Time:  0.4091255759994965 fractional seconds


Note: The average/worst case does not differ for the DP implementation
-----------------------------------------------------------------
```

4.1.2 Worst-Case Analysis(Changing Input Size)

```
Worst-Case Experimental(Change Input Size):
-------------------------


BF Worst Case Length:  0
Worst Case Brute Force LCS Time:  2.2081716939974285 fractional seconds


Dynamic LCS Worst Case Length:  0
Worst Case Dynamic LCS Time:  0.49057419000018854 fractional seconds




BF Worst Case Length:  0
Worst Case Brute Force LCS Time:  331.92209551400083 fractional seconds


Dynamic LCS Worst Case Length:  0
Worst Case Dynamic LCS Time:  1.2093682320009975 fractional seconds
```

4.1.3 Average-Case Analysis(Changing Input Size)

```
Average Case Scenarios:
--------------------------


BF Avg. Case Length:  1
Avg Case Brute Force LCS Time:  0.12108409499887784 fractional seconds


Dynamic LCS Avg. Case Length:  1
Avg. Case Dynamic LCS Time:  0.22923797099974763 fractional seconds


BF Avg. Case Length(n=6,m=2):  3
Avg Case Brute Force LCS Time(n=6,m=2):  0.2803812910005945 fractional seconds


Dynamic LCS Avg. Case Length(n=6,m=2):  3
Avg. Case Dynamic LCS Time(n=6,m=2):  0.6169195279999258 fractional seconds


BF Avg. Case Length(n=12,m=6):  6
Avg Case Brute Force LCS Time(n=12,m=6):  0.7294044709997252 fractional seconds


Dynamic LCS Avg. Case Length(n=12,m=6):  6
Avg. Case Dynamic LCS Time(n=12,m=6):  1.8294738530003087 fractional seconds


BF Avg. Case Length(n=24,m=18): 17
Avg Case Brute Force LCS Time(n=24,m=18):  83.80627718600044 fractional seconds


Dynamic LCS Avg. Case Length(n=24,m=18):  17
Avg. Case Dynamic LCS Time(n=24,m=18):  8.198277392999444 fractional seconds
```