Survey of Approaches to Solve the Longest Common Subsequence Problem

Computer Algorithms Project
4005-800-70
6 November 2012

Eitan Romanoff
Mark Thill

ABSTRACT

The Longest Common Subsequence problem is a classic problem in computer science, to which there exists many different manners of obtaining the solution. The problem is interesting in that the solutions are extensions of classical algorithmic strategies, such as memoization, and dynamic programming. The problem is also of interest to many fields beyond the scope of computer science for its application in biological genome sequencing, and other similar areas. Our project consisted of a thorough exploration and investigation of these various approaches to the Longest Common Sequence problem, including the implementation and analytical comparison of each.

LONGEST COMMON SUBSEQUENCE

A subsequence of a given sequence (hereby represented as a general string of a particular alphabet) is a “child” sequence of elements in which all elements are contained in the “parent” sequence, following the original ordering of sequenced elements. For example, one possible subsequence of the string S: **{100101}** is the substring $S_{2,4}$: **{01}**. *Subsequences*, however, are not required to be *substrings*. In fact, they are quite different, as a sequence S may have elements between particular elements of subsequence S’. For example, another subsequence of S defined above is S’: **{111}**.

The Longest Common Subsequence (LCS), then, is the longest of all possible subsequences present in sequence S. This is trivial when only one sequence is in question, as the LCS of a sequence is itself, but the problem becomes more interesting when two sequences are being compared to one another. The LCS of two sequences S1 and S2, then, is the longest subsequence contained within the set of all subsequences of S1, and the set of all subsequences S2.

APPROACHES TO THE LCS PROBLEM

Perhaps the most obvious way to implement the LCS problem is to use the naive recursive definition of the LCS between two sequences X and Y. The LCS is recursively defined as follows:

$$LCS\left(X_i, Y_j\right) = \begin{cases} \emptyset & \text{if } \; i = 0 \text{ or } j = 0 \\ \left(LCS\left(X_{i-1}, Y_{j-1}\right), x_i\right) & \text{if } x_i = y_j \\ \text{longest}\left(LCS\left(X_i, Y_{j-1}\right), LCS\left(X_{i-1}, Y_j\right)\right) & \text{if } x_i \neq y_j \end{cases}$$

This algorithm recursively walks through both sequences and compares element by element as it walks through. As noted above, the longest common subsequence between two sequences of zero elements is an empty sequence. Otherwise, should the two elements at any point in the algorithm match, it will add the letter to the subsequence, and step on each sequence. Otherwise, if the two elements differ, the algorithm is recursively called with two branches - one branch with one sequence taking a step, and the other branch with the other sequence taking a step. The longest common subsequence of those two branches is defined as the longest one that eventually returns from the recursion.

This is, unsurprisingly, quite slow. For two strings of length n, the runtime of this approach is 2^n, because in the worst case, there are no matching comparisons, and the recursion branches off with a step on the first, or the second strings. It should be noted that the true runtimes are dependant on lengths of both sequences, as they may differ, but we will assume that the sequences are of the same length.

The LCS problem, by its recursive definition, will create a conceptual binary tree (assuming two sequences are being compared) of potential LCS candidates at any particular depth, with the leaves be the comparison of empty sequences. However, the subtrees represented in this model are not necessarily unique. Clearly, the longest common subsequence of any two sequences is invariable given two sequences X and Y, no matter when the LCS of X and Y are evaluated during the recursion. This allows one to think of the LCS problem in terms of subproblems, and one can abuse this property to create faster algorithms to find the LCS.

One such technique that can be applied is *memoization*, where the subproblems can be stored in a dictionary-like structure to be recalled later. Thus, repeated sub-problems are not re-calculated, but rather recalled in O(constant) instructions, saving much computational complexity. The implementation of the memoized approach simply adds a check prior to each recursive call. It should be noted that there is a constant overhead to the checking associated with the storage and recall step.

The runtime of this approach using memoization for optimizing is $O(n^2)$ which is clearly visible when one considers the storage of these results. Should the storage be in an N by N table of all possible subsequence comparisons, each value would, at worst case, be computed one time, and then subsequently looked up.

While memoization is often seen as a “top-down” approach, the LCS problem and its properties also lend itself well to the “bottom-up” approach. Dynamic Programming techniques can be applied to the LCS problem, where results are stored in a table, and the result is computed through a bottom-up trace through the table. The benefits here are twofold. Firstly, the constructed common subsequence at every iteration does not need to be stored - only the length is necessary, effectively reducing the storage space. Secondly, the algorithm reduces the number of computations over memoization because lookup checks are not necessary. While the two approaches are the same asymptotically, the DP solution is expected to run faster.

Dan Hirschberg further optimized the dynamic programming strategy with regards to how it can be applied to the longest common subsequence problem, effectively reducing the space requirement from n^2 to linear space, while retaining quadratic time complexity. To do this, he effectively modified the way the problem can be broken into subproblems such that only two rows of the DP matrix are ever needed at one time. Barring recursive calls, the space required is thus O(2n). Hirschberg then proves in his paper that there will be no greater than 2n-1 calls to the overall function (function C).

IMPLEMENTATION AND PROFILING

All four of the LCS implementations, as well as the profiling and time processing, were implemented in python. Python was chosen for its powerful string slicing, which effectively reduced the difficulty of implementation to more closely resemble the algorithms presented in the various papers examined. Furthermore, Python has the added benefit of being platform independent, and very readable by nature. A higher level script that runs the profiled tests was also used, and makes use of python's cProfile module in order to record CPU usage and function calls to be used in the analysis. Global recursion counting hooks were added into each implementation. See the attached README for full usage details.

RESULTS

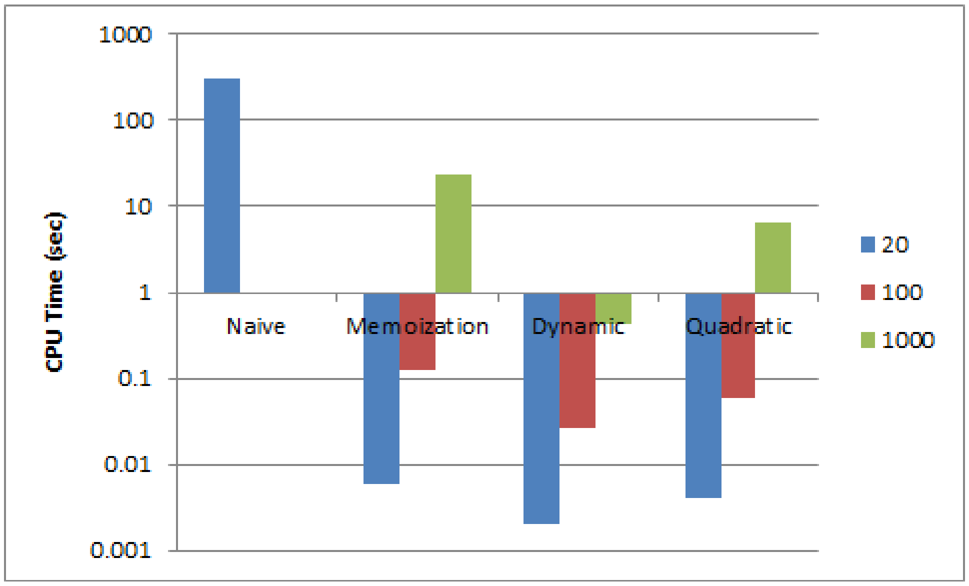
Computability for Length 40,000 Sequences

As a part of the requirements for the project, we were expected to be able to compute the longest common subsequence for two random sequences containing 40,000 elements. While this was not computable for naive (computation time), memoization (memory usage), and DP (memory usage) techniques, our implementation of Hirschberg's Linear Space algorithm was able to find a solution in **2401792064 function calls** resulting in a CPU usage time of **3307.2 seconds**.

Stress test to 10 seconds

For each algorithm, a test was run to see what the maximum string length the algorithm could run in 10 seconds of CPU time. Each test was run on the same computer to keep consistency. The actual test involved iteratively increasing the length of two input strings until time the algorithm takes to compute the solution is over 10 CPU seconds. The input strings chosen were always the same length, and strings were fixed such that the worst case scenario (no matches) was always evaluated. This is because that with random strings, some random strings will be computed much faster (especially with the naive approach) than fixed strings due to the execution of different cases. One example of two fixed strings of length 5 is “00000” and “11111”. The length of these strings was gradually increased until the CPU time exceeded 10 seconds. The lengths and the number of recursive calls (if applicable) is noted in the table below:

Naive	Two strings of length 11
Recursive with memoization	Two strings of length 692
Dynamic	Two strings of length 3133
Quadratic-time linear-space	Two strings of length 1221



Comparison of time to recursive calls
This test involved seeing how the number of recursive calls an algorithm uses affects its runtime. For strings of various lengths, the number of recursive calls is recorded as well as the amount of CPU time the algorithm took to find the solution. The strings used in the experiment were randomly generated, but the same generated strings are used to test all algorithms.

Strings	AGGCTTACCGTGC AAGCGAGGATTCTAT TTG	2 strings of length 20	2 strings of length 100	2 strings of length 1000
Naive	2729478 calls 31.908 seconds	26062321 calls 312.904 seconds	N/A*	N/A*
Recursive with memoization	238 calls 0.004 seconds	365 calls 0.006 seconds	7623 calls 0.123 seconds	911195 calls 23.084 seconds
Dynamic	0 calls 0.001 seconds	0 calls 0.002 seconds	0 calls 0.026 seconds	0 calls 0.434 seconds
Quadratic-time linear-space	25 calls 0.002 seconds	35 calls 0.004 seconds	189 calls 0.058 seconds	1875 calls 1.8 seconds

*Naive was not run for the last test because of the extreme length of time it would take to compute

Clearly, the computation time is directly related to the input size of the randomized strings in a manner representative of the asymptotic complexity of the algorithm. The most obvious - the naive solution - quickly becomes uncomputable for even relatively small values of n, as the growth of the algorithm is exponential in nature. The memoization approach ended up being, unsurprisingly, the second worst performer of the four methods tested, but the is still quadratic in nature, which is apparent by the orders of magnitude of growth in the computation time with respect to the length of the input strings. The two DP solutions, on the other hand, run faster than the two naive-esque approaches.

Something else of interest is the comparison of the two DP solutions. Both are in the quadratic time complexity family, but there are different coefficients associated with each of them. As mentioned before, Hirschberg's algorithm has some overhead to minimize the space needed for each recursive call. This constant becomes visible in the resulting CPU metrics, as the general DP method is faster by a factor of 3.

The memory usage metrics are listed as shown below.

Size Sequence	Length 500	Length 1000	Length 1500	Length 2000
Naive	N/A*	N/A*	N/A*	N/A*
Memoized	125 MB	600 MB	1100 MB	N/A**
Dynamic	10 MB	20 MB	30 MB	60 MB
Hirschberg	<5 MB	<5 MB	<5 MB	<5 MB

* Unable to compute due to computation time
** Unable to compute due to memory limitations

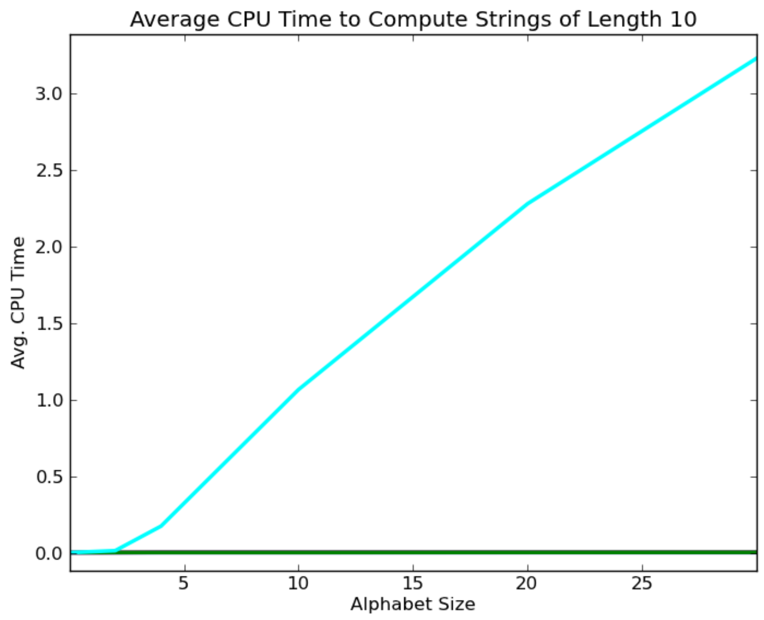
As expected, the recursion depth seen in the memoization technique quickly consumes too much memory for our Python environment to run in. The standard DP algorithm, on the other hand, shows a steady growth (quadratic in nature), of memory usage. This is because it always creates a table of size n^2. While this isn't a huge growth in memory usage, it renders a computation for two sequences of size 40,000 impossible. Hirshberg solves this issue by keeping the memory usage in linear space, and the footprint is hardly noticeable even for extremely large values of n.

Variable Alphabet Size

This test has the goal of seeing how the size of the alphabet affects the performance of the algorithms. This done by comparing 5 alphabets of sizes 2,4,10,20, and 30. The test is to find the average run time of randomly generated strings of the same length from each alphabet.

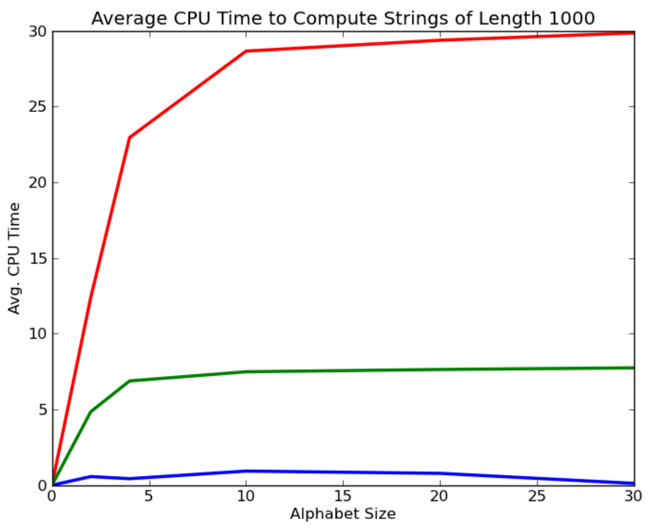
Average CPU Time to Compute Strings of Length 10

Alphabets	Length 2	Length 4	Length 10	Length 20	Length 30
Naive	0.013	0.173	1.064	2.277	3.2266
Memoized	0.001	0.002	0.002	0.002	0.002
Dynamic	0.0004	0.001	0.001	0.001	0.001
Hirschberg	0.001	0.001	0.001	0.0016	0.002



Average CPU Time to Compute Strings of Length 100

Alphabets	Length 2	Length 4	Length 10	Length 20	Length 30
Naive	n/a	n/a	n/a	n/a	n/a
Recursive with memoization	0.097	0.161	0.192	0.195	0.197
Dynamic	0.026	0.034	0.039	0.04	0.041
Quadratic-time linear-space	0.055	0.072	0.081	0.084	0.085



Alphabets	Length 2	Length 4	Length 10	Length 20	Length 30
Naive	n/a	n/a	n/a	n/a	n/a
Recursive with memoization	12.40	22.95	28.66	29.38	29.86
Dynamic	0.569	0.430	0.933	0.783	0.119
Quadratic-time linear-space	4.850	6.883	7.490	7.640	7.747

* Every value is the average of 10 results
** Values marked n/a were not computed because of the extreme amount of time it would take to do so

With regards to the DP approaches, the runtime is not at all dependant on the size of the alphabet used. This is not the case, however, for the naive recursive, and naive with memoization approaches. The reason for this is that, in the case of non-matched elements between the two sequences, the complexity grows exponentially, as there are two recursive calls in this case. By nature of a large alphabet, there is a greater likelihood that there will be longer unique subsequences, and thus, less direct matches between elements. This pattern seems to flatten out over a certain amount of elements in the alphabet used.

Book Exercise 15.4-1
Determine an LCS of (1,0,0,1,0,1,0,1) and (0,1,0,1,1,0,1,1,0).
Answer: 0,1,0,1,0,1

This was solved using our program with the following command:
```
python mainLCS.py m i 10010101 010110110
```

Output:
```
Finding the LCS for Strings:"10010101" "010110110"
       with algorithm:  memoized LCS

===================================================
Configuration for LCS in 0.001 CPU seconds.
  String lengths:
     S1 - 8
     S2 - 9
  First:  10010101
  Second: 010110110
  LCS:    010101
  Recursive Calls:  40
```

CONCLUSION

Just as the LCS problem has many applications, it also has many approaches, and is a good example problem as it allows for many different solutions that fall under different algorithm families. The naive approach quickly becomes unwieldy and slow due to its exponential runtime.

Memoization reduces the runtime to O(n^2), but the memory usage becomes too large due to extreme recursive depths. Dynamic programming solves this problem, and keeps to the quadratic time delivered by memoization, but suffers from quadratic memory growth over large sized sequences. Hirschberg’s algorithm improves on the standard DP approach by keeping the algorithm to linear space, but at the expense of a runtime coefficient of three times that of the standard DP approach. For smaller data sets, we believe the standard DP approach to be superior, due to its speed, until the sequences become so large that they strain the memory allocation of the running environment.