

Empirical Complexities of Longest Common Subsequence Algorithms

Daniel Kunkle

Computer Science Department
College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14623
drk4633@rit.edu
<http://www.rit.edu/~drk4633>

June 12, 2002

Abstract

The empirical time complexities of several algorithms for finding the Length of Longest Common Subsequence (LLCS) and the actual Longest Common Subsequence (LCS) of two sequences are examined. These algorithms include a naive recursive algorithm, a recursive method with memoization, dynamic programming, and the Hirschberg LCS algorithm [3]. The empirical time complexities are compared to theoretical time complexities in order to find the “hidden” constant ignored by the theoretical limits. Further, the effects of sequence structure and alphabet size on empirical time complexity are examined.

1 Longest Common Subsequence

Informally, a subsequence of a sequence is simply that sequence with zero or more elements left out. Formally, as defined by [1], given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$ another sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ is a *subsequence*

of X is there exists a strictly increasing sequence of $\langle i_1, i_2, \dots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \dots, k$, we have $x_{i_j} = z_j$.

Given two sequences X and Y , a third sequence Z is a *common subsequence* of X and Y if it is a subsequence of both X and Y . Z is the *longest common subsequence* (LCS) of X and Y if there is no other common subsequence of X and Y longer than Z .

The *longest common subsequence problem* is as follows: given two sequences X and Y , find the LCS of X and Y . The length of the LCS is referred to as the LLCS throughout this paper. As will be shown later, the problem of finding the LCS of two sequences is significantly more difficult than simply finding the LLCS of two sequences.

For example: given the sequences $X = \langle A, C, G, T, G, A, C, T \rangle$ and $Y = \langle G, A, C, T, A, G, T \rangle$, the LLCS of X and Y is 5 and the an LCS of X and Y is $\langle A, C, T, G, T \rangle$. Note that two sequences may have more than one LCS. The sequence $\langle A, C, T, A, T \rangle$ is also an LCS of X and Y having a length of 5.

The LCS of two sequences is one measure of how similar the two sequences are to each other. Algorithms solving the LCS problem are often used in applications that require a measure of similarity, such as:

- Biological applications comparing DNA strands, which are sequences of the four bases represented by A, C, T, and G.
- The Unix utility *diff*, which reports the differences between two files, expressed as a minimal list of line changes to bring either file into agreement with the other [4]. Here, the lines are treated as complex elements in a sequence of lines.

2 Algorithms to Compute the LLCS

All algorithms for computing the LCS of two sequences are based on the following theorem (from [1]).

Theorem 1 (optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .

2. If $x_m \neq y_n$ then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .
3. If $x_m \neq y_n$ then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1} .

2.1 Naive Recursive Algorithm

The most straightforward algorithm based on Theorem 1 is a recursive one, corresponding the following formula.

Given two sequences X and Y , let X_i and Y_j be prefixes of X and Y containing i and j elements, respectively. Let $c[i, j]$, the LLCS of X_i and Y_j be defined as follows:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

A simple implementation of this function will yield an exponential number of recursive calls and therefore an exponential time algorithm, with time complexity $O(2^{mn})$, or $O(2^{n^2})$ if $m = n$. An improvement can be made by *memoizing*, or remembering, the results of the recursive sub-problems.

2.2 Recursion with Memoization

Because the recursive function takes as arguments i and j , with ranges of $[0, m]$ and $[0, n]$ respectively, there are $(m + 1)(n + 1)$ possible subproblems. If the recursive algorithm is computing on the order of 2^{mn} subproblems many are computed several times. By creating a lookup table to store the answers, these re-computations can be avoided. Whenever the function is called recursively it will simply check the lookup table to see if the current subproblem has been computed before. If it has the value is returned, if not the value is calculated, stored in the lookup table, and returned.

The memoizing LLCS algorithm is called once initially and at most twice for each possible subproblem, yielding at most $2(m + 1)(n + 1) + 1$ calls and time $O(mn)$ [2].

This polynomial time upper bound is significantly better than the exponential time upper bound of the naive recursive algorithm.

2.3 Dynamic Programming

The memoizing recursive approach is considered “top-down” in that it waits until a result is needed to compute and store the value. A “bottom-up” dynamic programming approach will compute all of the subproblems, starting with the simplest and working towards the final solution.

A table, c , of size $(m + 1)$ by $(n + 1)$ holds the LLCS of all X_i and Y_j pairs. Each of the cells is computed in row order according to Theorem 1. So, the value of any cell, $c[i, j]$ depends only on three other cells, $c[i - 1, j - 1]$, $c[i - 1, j]$, and $c[i, j - 1]$. The value at $c[m, n]$ will hold the LLCS of X and Y .

When finding only the LLCS of the two sequences, the dynamic programming method can easily be made to run in linear space. Because the computation of one row of the dynamic programming table depends only on the previous row, any row can be discarded after the next row is computed. However, this task is more difficult when computing the LCS of two sequences. In this case, a second table, b , is used to define a “path” from $b[m, n]$ to a cell on $b[0, n]$ or $b[m, 0]$. The path from a cell, $b[i, j]$, will lead to the subproblem that its value was computed from, either $b[i - 1, j - 1]$ in the case of a match between X_i and Y_j , or $b[i - 1, j]$ or $b[i, j - 1]$ in the case that they do not match. Because this full table is needed for LCS recovery the dynamic programming method will use $\theta(mn)$ space.

2.4 Hirschberg’s LCS Algorithm

Hirschberg’s LCS Algorithm [3] runs in time $O(mn)$ and space $O(m + n)$. This is achieved using methods significantly more complicated than the previous three LLCS and LCS algorithms, but achieves an important reduction in space complexity without an asymptotically significant increase in time complexity. There is an increase in the practical time needed for Hirschberg’s LCS algorithm over the dynamic programming LCS algorithm, but for large inputs the reduction in space usage becomes necessary.

3 Empirical Results

3.1 Environment

All code was written in C++ and compiled with `cc` using the `-O3` flag. All experiments were executed on a Sun Ultra-80 with four US-II 450 MHz processors and 4096 Megabytes of main memory (only one processor was utilized at any time). All time measurements are for the actual CPU time used by a process in computing an LLCS or LCS.

3.2 Experiments

Each of the four algorithms presented previously were implemented to find both the LCS and LLCS of two given sequences. Each method, for both LCS and LLCS, was tested for input sizes ranging from 10 to 20,000. Those methods that could not complete large inputs, such as the naive recursive algorithm, were constrained to relatively small inputs.

To better achieve an “average case” running time, each method was tested at each size for five different random input sequences and the average over these five runs was recorded. This averaging is particularly important with the naive recursive algorithm due to the dramatic affect the structure of the input sequences has on computation time.

Further, to improve the accuracy of timing on very small inputs, those experiments that took less than 1 second to complete were repeated until a full second of computing time was consumed and the average time to complete one LCS or LLCS was recorded.

The empirical results here offer the following:

1. A comparison of the time complexities of each method.
2. A comparison of each method’s observed time complexity with their theoretical time complexity, yielding a value for the constant multiplier not provided by asymptotic formulas.
3. An examination of the affect of various alphabet sizes on the observed time complexity of each method.
4. An examination of the differences in time complexities when computing either the LCS or LLCS of two sequences for each method.

5. An examination of running times when using sequences of differing lengths.
6. An examination of the relationship between the number of recursive calls made and the running time in the naive recursive algorithm.
7. A measure of the increase in time complexity when using Hirschberg's LCS algorithm to decrease space complexity to $O(m + n)$.

3.3 One Minute/Hour Marks

As an intuitive measure of the time efficiency of each algorithm, figure 1 shows the largest inputs an LLCS can be found for in one minute and one hour by each method. The input size listed applies to both inputs, X and Y , which are of equal length in this experiment.

LLCS Method	One Minute	One Hour
Naive Recursion	32	42
Memoization	11,300	87,560
Hirschberg's	17,000	131,720
Dynamic Programming	25,340	196,270

Figure 1: Largest inputs an LLCS can be found for in one minute and one hour by each method. Input sizes are for both input sequences.

3.4 Observed Running Times for LLCS Algorithms

Figure 2 shows the observed running times for algorithms computing the LLCS of two sequences, including naive recursion, memoization, and dynamic programming. The two quadratic algorithms, memoization and dynamic programming, are obviously superior to the exponential time recursive method, which appears as a near vertical line on the left side of the graph.

3.5 Empirical and Theoretical Time Complexities

Figures 3, 4, 5, and 6 show the observed running times of each LLCS method (along with Hirschberg's LCS method) compared with a closest fit line corresponding to the theoretical time complexities of each algorithm with a

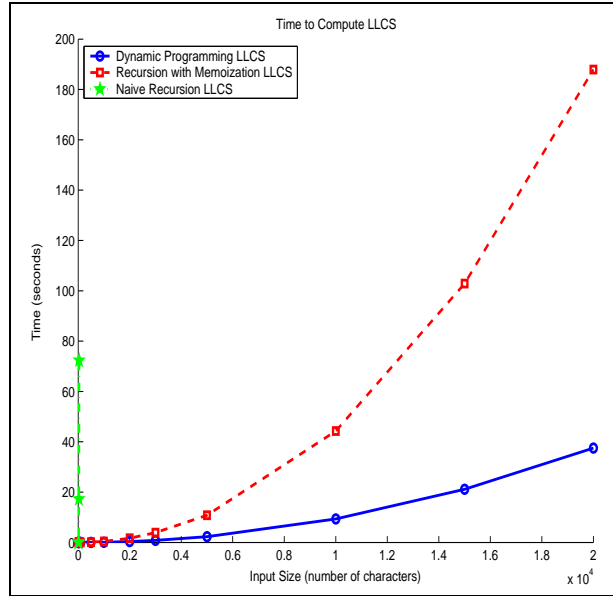


Figure 2: Observed running times for three LLCS algorithms for various input sizes. In each case both inputs are of the same size.

constant multiplier. The observed time complexities for each algorithm are shown in figure 7.

The large error between the theoretical and observed time complexities of the recursive method are due to the fact that the running time of that algorithm depends on the number of recursive calls made, which is a function of the structure of the sequences, not simply the lengths of them. Later on the relationship between the number of recursive calls and the running time will be explored.

3.6 Effects of Alphabet Size on Running Times

In this work, alphabets are assumed to be small, in this case having either 2 or 4 elements. This is because one of the main application area of LCS algorithms at this point is in determining the level of similarity of two DNA

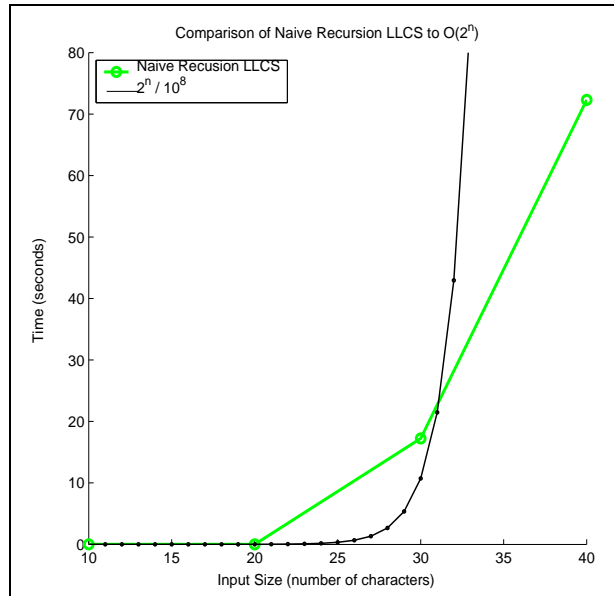


Figure 3: Observed and theoretical time complexities of naive recursive LLCS

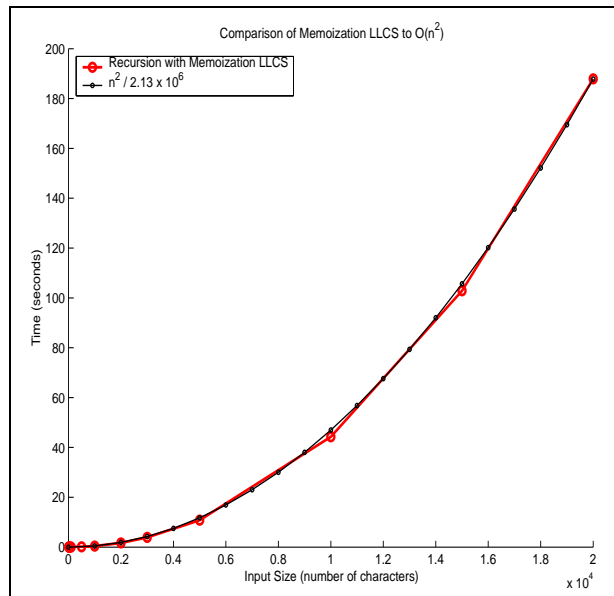


Figure 4: Observed and theoretical time complexities of memoized LLCS

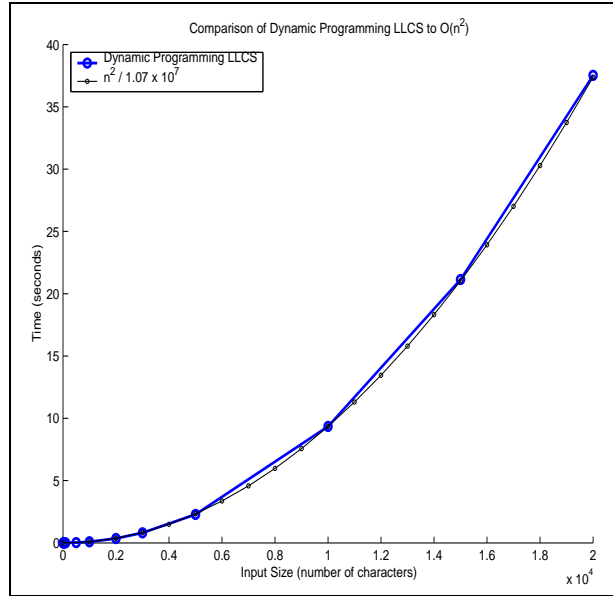


Figure 5: Observed and theoretical time complexities of dynamic programming LLCS

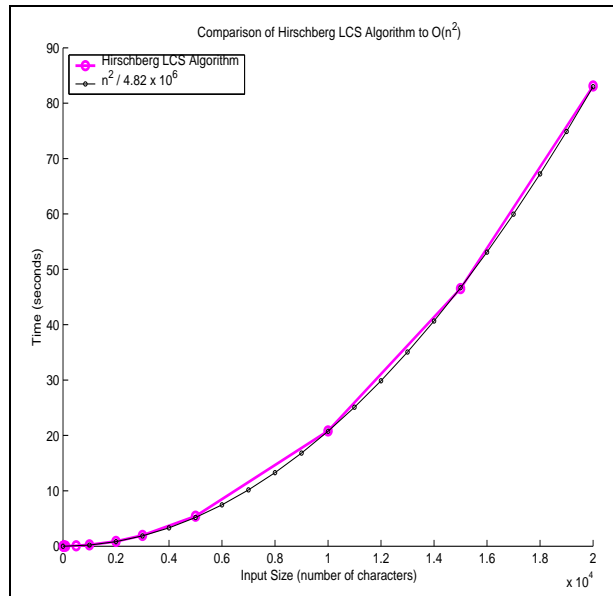


Figure 6: Observed and theoretical time complexities of Hirschberg's LCS

Method	Observed Time Complexity
Naive Recursion LLCS	$2^n/10^8$
Memoization LLCS	$n^2/2.13 \times 10^6$
Dynamic Programming LLCS	$n^2/1.07 \times 10^7$
Hirschberg's LCS	$n^2/4.82 \times 10^6$

Figure 7: Observed time complexities of LCS algorithms.

sequences, made up of bases represented by the alphabet $\{A, C, T, G\}$. Figures 8, 9, 10 show the running times of each LLCS algorithm in the cases of using alphabets of size 2 and 4. Because the running time of the recursive and memoized version of the algorithm depend directly on the number of recursive calls made they are more heavily affected by a larger alphabet. This is due to the fact that every time there is a mismatch between element X_i and Y_j two recursive calls are made, whereas when they match only one recursive call is made. With a larger alphabet, when using random sequences as is done here, the number of mismatches will increase as the size of the alphabet increases. The dynamic programming method is not significantly affected by alphabet size because it will compute $O(mn)$ subproblems independent of the structure of the sequences.

3.7 Running Times of LLCS and LCS

Finding the LLCS (length of the longest common subsequence) is an easier task than finding an actual LCS. In terms of implemented algorithms, much of the increase in running time is due to the extra string or sequence manipulation routines necessary to construct the subsequence (this is easily determined by using a code profiler, such as `gprof`).

Additionally, when using dynamic programming, it is relatively easy to construct a linear space algorithm to find the LLCS of two sequences, but considerably more difficult to reconstruct an actual LCS of the sequences in linear space (which is the purpose of Hirschberg's algorithm).

Figures 11, 12, 13 show the increases in running time of the recursive, memoized, and dynamic programming algorithms when reconstructing an LCS instead of simply finding the LLCS.

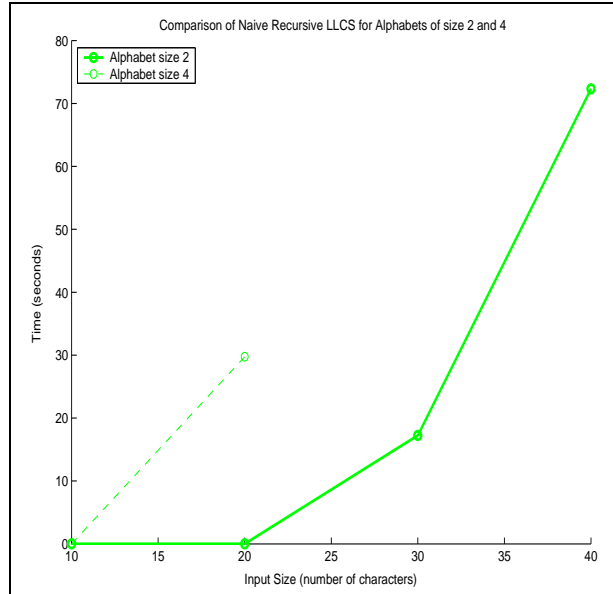


Figure 8: Running time of recursive LLCS algorithm when using an alphabet of size 2 or 4.

3.8 Uneven Input Sequences

Throughout this paper the time complexities of the various LLCS and LCS algorithms has been put in terms of n . This holds true when both input sequences are of the same length. More accurately, the running times are in terms of m and n . For example, the theoretical time complexity of the memoized, dynamic programming, and Hirschberg LCS algorithms are all $O(mn)$. Figure 14 shows the running times of the dynamic programming LLCS algorithm with one input sequence fixed to size 100 while the other input sequence ranges from 10 to 20,000 characters. As expected this yields a linear relationship between n and the running time.

3.9 Recursive Calls and Running Time

As mentioned previously, the running time of the recursive LLCS and LCS algorithm depends heavily on the number of recursive calls made in the process. This, in turn, depends on the structure of the sequences. Figure 15 shows the relationship between the number of recursive calls made and the

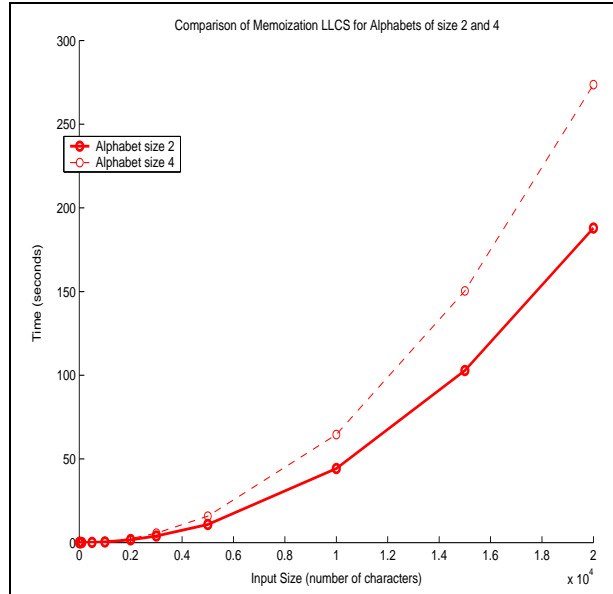


Figure 9: Running time of memoized LLCS algorithm when using an alphabet of size 2 or 4.

running time. As expected, the relationship is linear as each call takes constant time.

3.10 Running Time Cost of Linear Space LCS

Hirschberg’s LCS algorithm provides an important reduction in space requirements over the $O(mn)$ space requirements of the plain dynamic programming LCS methods. The added complexity of the algorithm, however, does increase running time by a constant factor. Figure 16 shows this increase, which was found to be a factor of 1.41. This is certainly an acceptable increase when dealing with very large pairs of sequences that would cause the dynamic programming table to exceed the memory capacities of most machines.

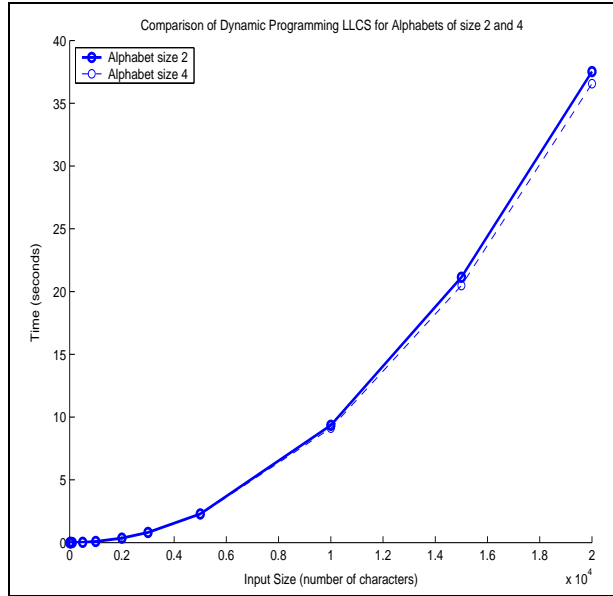


Figure 10: Running time of dynamic programming LLCS algorithm when using an alphabet of size 2 or 4.

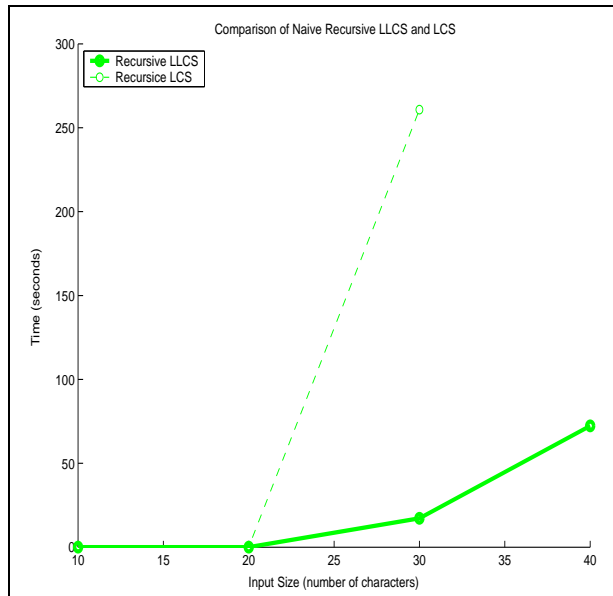


Figure 11: Running times of recursive algorithm for LLCS and LCS

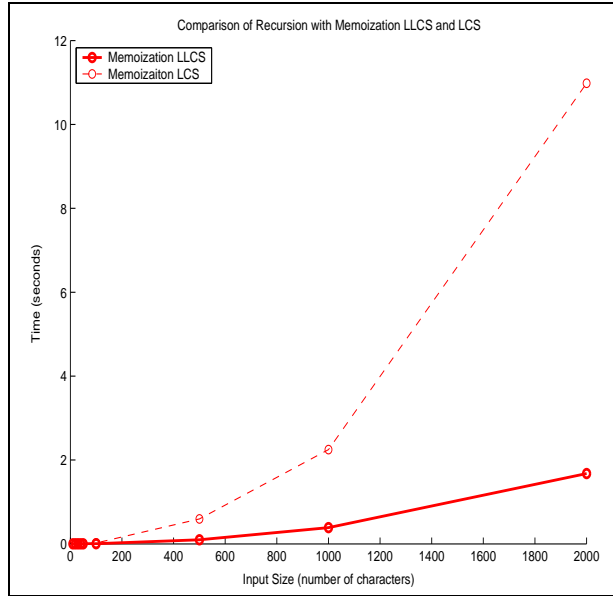


Figure 12: Running times of memoized algorithm for LLCS and LCS

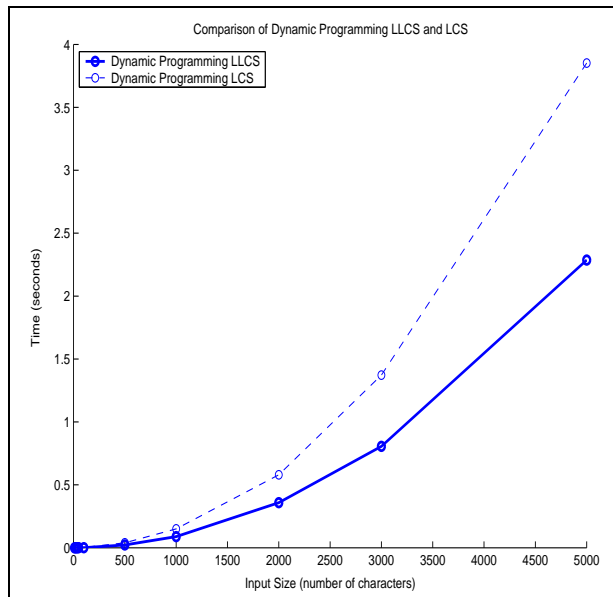


Figure 13: Running times of dynamic programming algorithm for LLCS and LCS

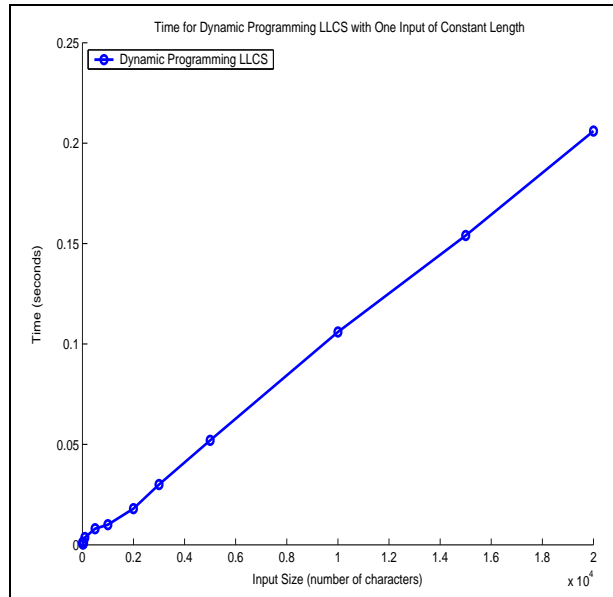


Figure 14: Running times of dynamic programming LLCS algorithm with one input fixed to size 100.

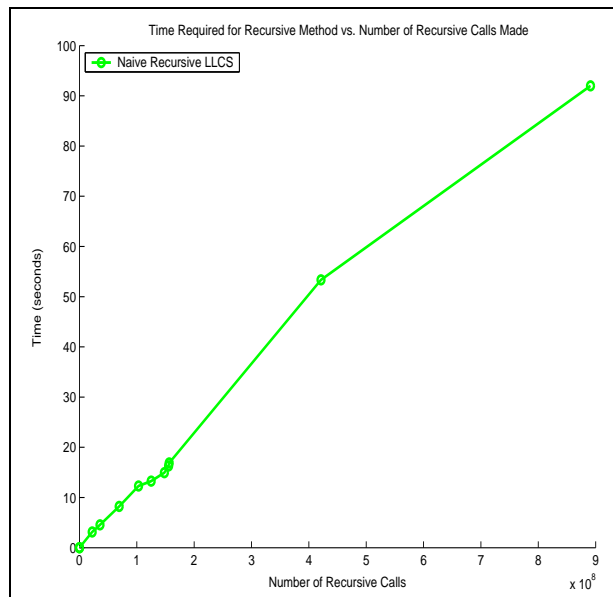


Figure 15: Relationship between the number of recursive calls made and the running time of the naive recursive LLCS algorithm.

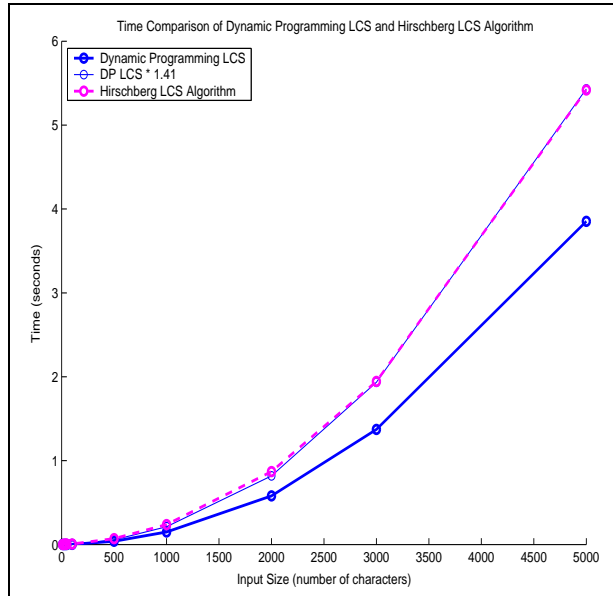


Figure 16: Comparison of observed time complexities of the dynamic programming LCS algorithm and Hirschberg’s linear space LCS algorithm. The increase in time complexity of Hirschberg’s LCS was found to be 1.41 over dynamic programming.

References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.
- [2] David Eppstein. Ics 161: Design and analysis of algorithms, lecture notes for february 29, 1996. <http://www.ics.uci.edu/~eppstein/161/960229.html>.
- [3] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.
- [4] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey, June 1976.