Implementation and Analysis of Four LCS Algorithms

Jason Smith
RIT Department of Computer Science
jas7553@rit.edu

Daniel Cappuccio
RIT Department of Computer Science
djc5444@rit.edu

ABSTRACT
This paper describes the implementation and measurement of four different algorithms used to find the longest common subsequence between two input strings. The algorithms included are titled as follows: naive recursive, recursive memoization, dynamic programming, and Hirschberg. All algorithms were implemented in Java 7 and all experiments were executed on the same computer through the Eclipse IDE. In a general sense, the algorithms performed as expected with the recursive versions exhibiting an exponential time complexity and the dynamic programming varieties executing in approximately $O(N^2)$.

Categories and Subject Descriptors
F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*LCS*

General Terms
Algorithms, Experimentation, Measurement, Performance

Keywords
LCS, Dynamic Programming, Naive Recursive, Hirschberg

1. INTRODUCTION
The longest common subsequence problem is a classic example of a problem that can be tackled in many different ways. As the name implies, the goal of the algorithm is to discover the longest common subsequence between two strings. Note that this is different from the longest common substring problem in that there can be any number of characters separating two matched characters. Consequently, one can view this as a simple recursive problem whereby one can logically decompose each string and build a solution from the smaller, easier to solve subproblems. While this method makes good sense, the time complexity proves to be exponential. Even with memoization, the algorithm fails to handle strings of any meaningful size.

With a need to handle larger string sizes, different techniques needed to be developed to handle the LCS problem. Because of the optimal substructure nature of the problem, dynamic programming techniques are an ideal approach. Building a matrix that computes the largest possible length of the LCS for any given substrings of the originals (including the full string itself), enables one to solve the problem in $O(N^2)$ time and space. Even though the time complexity is greatly reduced, space soon becomes an issue with strings approaching a size of 20,000.

In 1975, Daniel S. Hirschberg developed an algorithm that used a modification of the the dynamic programming approach for LCS to find merely the length of the LCS in linear space. He then wrapped this modification in an ingenious algorithm that preserves $O(N^2)$ time and space complexity.

The rest of the paper is laid out as follows: Section 1 will go through each algorithm on a technical level and explain some of the implementation decisions. Section 2 will address the methodology for our testing suite and section 3 will compare both of the recursive algorithms with regards to time complexity and number of recursive calls. Section 4 will address the analysis of the dynamic programming and Hirschberg algorithms.

2. ALGORITHM IMPLEMENTATION
Each of the algorithms were implemented in Java 7 and developed in the Eclipse IDE. Furthermore, all of the algorithms are a subclass of the superclass LcsSolver which will be discussed more in section 3.

2.1 Naive Recursive
As figure one shows, the naive recursive function is a very simple recursive function. If either of the inputs is empty, then the LCS must be an empty string. Otherwise, compare the last element of each input. If the character is the same, then recurse on each input minus the last character while adding that character to the end of the LCS. Otherwise, recurse down two different paths: one path where the first input removes its last character and one path where the second input removes its first character. Finally, determine which of the two paths yielded the larger result and return that.

The exponential nature of this algorithm becomes readily apparent when you consider strings of as small a size as five characters. The only case where this algorithm can theo-

```
private String lcs(char[] x, char[] y, int i, int j) {
        performanceMonitor.makeRecursiveCall();

        if (i == 0 || j == 0) {
                return "";
        }

        String lcs;

        char[] xSub = Arrays.copyOf(x, x.length - 1);
        char[] ySub = Arrays.copyOf(y, y.length - 1);

        if (x[i - 1] == y[j - 1]) {
                lcs = lcs(xSub, ySub) + x[i - 1];

        } else {
                String lcsSub1 = lcs(x, ySub);
                String lcsSub2 = lcs(xSub, y);

                lcs = lcsSub1.length() > lcsSub2.length() ? lcsSub1 : lcsSub2;
        }

        return lcs;
}
```

Figure 1: Naive Recursive LCS Algorithm

```
private void buildTables() {
        clearTables();

        for (int i = 0; i < m + 1; i++) {
                c[i][0] = 0;
        }

        for (int j = 0; j < n + 1; j++) {
                c[0][j] = 0;
        }

        for (int i = 1; i < m + 1; i++) {
                for (int j = 1; j < n + 1; j++) {
                        performanceMonitor.makeRecursiveCall();
                        if (x[i - 1] == y[j - 1]) {
                                c[i][j] = c[i - 1][j - 1] + 1;
                                b[i][j] = UP_LEFT;
                        } else if (c[i - 1][j] >= c[i][j - 1]) {
                                c[i][j] = c[i - 1][j];
                                b[i][j] = UP;
                        } else {
                                c[i][j] = c[i][j - 1];
                                b[i][j] = LEFT;
                        }
                }
        }
}
```

Figure 2: Dynamic Programming:Build Tables

retically perform better is when the two input strings are identical. In that case, you would only observe one recursive call at each level of the algorithm which would also coincide with the length of the initial inputs. However, this is clearly a trivial situation and there would be no need for computation in that case.

2.2 Recursive Memoization

The algorithm can be modestly improved by the addition of memoization. This technique stores the results of each lcs call so that it can be accessed at a later time thus preventing duplicating computations. The process is simple enough: create a dictionary to store a mapping between the input strings and the resulting lcs. After the recursive base case but before computing the lcs, check if the current problem as already been solved by accessing the dictionary and return the result. Otherwise, compute the lcs as normal and then store the result in the dictionary.

Despite being a neat trick, the number of recursive calls still explodes exponentially, thus rendering this algorithm next to useless.

2.3 Dynamic Programming

The dynamic programming solver takes a different approach to the problem. Instead of breaking the strings apart, this algorithm builds a matrix that merely tracks the size of the lcs as it goes. It initializes a matrix, c, with the first row and column set to 0. Then, it moves through the matrix comparing the character at index i of the first input with the character at position j of the second input. If the two characters match, then the value stored in that index is equal to one plus the value stored in the index to the upper left (c[i-1][j-1]). Otherwise, it takes the greater of the values stored in the index to the left or above. Simultaneously with the creation of this matrix, a second matrix is made which tracks the cell from which the value in c was derived. This matrix, b, is used in the reconstruction of the actual lcs by following the path indicated and adding a character to the end of the lcs if the value of the b matrix is up_left.

A point of note with this algorithm is that the reconstruction of the lcs from the b table can be done both iteratively and recursively. Our initial implementation used a recursive approach which we believed was causing the algorithm to fail[1] for input strings of size 15,000 or greater. Because of this, we rewrote the function to proceed in an iterative manner.

However, despite a modest increase in performance, the algorithm was still severely under-performing. We speculated that the intensive use of memory was perhaps the root of the problem and adjusted a setting in Eclipse to allow the program to use all available memory on the system if needed. Again, there was a small increase in performance but the numbers were still way out of line with expected $O(N^2)$ behavior. By timing simply the creation of the lcs length and noticing a drastic improvement in speed, we turned our attention to the data type of the values being stored in our b table: enums. Apparently, Java enums are implemented as a class instead of a primitive data type. Consequently, we were creating over 100,000,000 instances of a class to fill this table. No wonder memory was an issue. By replaces the enums with private static final int variables, the execution time finally came into line with our expectations.

2.4 Hirschberg

Despite both dynamic programing and Hirschberg running in $O(N^2)$ time, dynamic programming's performance starts to degrade because of memory issues with larger string sizes. In fact, Hirschberg is able to do string sizes considerably larger than basic dynamic programming because of the fact that it operates using only O(N) space.

Hirschberg achieves O(N) space using a modified version of the dynamic programming approach. In what his paper and our implementation calls "algorithm B", Hirschberg creates a windowed portion of the c table described above. At any given point, there are only two rows of that stable stored

[1]Fail is used here in the sense of the algorithm not performing up to expectations. While the algorithm was running for those inputs, it was taking a significantly longer time than expected and we believed the recursion in this step of the process was to blame.

in memory. The only real difference between this and the dynamic programming algorithm is the size of K (a 2 by n+1 matrix) and the first inner for loop that reassigns the first row of K before proceeding with the algorithm. Thus, at the end of algorithm b, one can find the lcs of the input by looking at K[1][j].

```
private int[] algB(int m, int n, char[] x, char[] y) {
        int[][] K = new int[2][n + 1];

        for (int j = 0; j < n + 1; j++) {
                K[1][j] = 0;
        }

        for (int i = 0; i < m; i++) {
                for (int j = 0; j < n + 1; j++) {
                        K[0][j] = K[1][j];
                }
                for (int j = 1; j < n + 1; j++) {
                        if (x[i] == y[j - 1]) {
                                K[1][j] = K[0][j - 1] + 1;
                        } else {
                                K[1][j] = Math.max(K[1][j - 1], K[0][j]);
                        }
                }
        }

        return K[1];
}
```

Figure 3: Hirschberg: Algorithm B

The array returned by algorithm be is used to determine the best location to split the string for subsequent recursions. The truly ingenious party of Hirschberg is the way that he splits and reverses the input strings for the calls to alg b. By finding the optimal position in respect to both of the returned arrays, one is able to optimally split the problem for future recursive calls which go about creating the first and second half of the lcs respectively. Please refer to the attached code to see the full implementation.

3. UTILITY CLASSES AND METHODOLOGY

3.1 Utility Classes

Each of the algorithms mentioned above is a subclass of LcsSolver. This helps to facilitate the creation of a generic solver that can be used in the higher level scripts. The LcsSolver stores the processed input strings[2], size of the input strings, and a performance monitor. The lcs and lcsLength methods of this class are invoked by the user, process the input, then delegate the specific functionality of finding the lcs to the overridden method calls of the subclass. A timer is started immediately before this delegated call and stopped immediately after it returns. This precludes any activities not directly related to solving the lcs problem from our total time.

The performance monitor class provides a suite of functions designed to measure the execution time of the various algorithms. This class makes it easy to start and stop system timers as well as obtain total execution time and recursive call counts. Throughout the various algorithms, this performance monitor increments the recursive call count at the beginning of each lcs or lcsLength function call.

[2]The class accepts strings as input but converts them to character arrays in all cases.

The other utility class, RandomStringGenerator, does exactly what its name implies. It takes as arguments to its constructor the character alphebet to use, the size of strings to generate, and optionally, a random seed. The seed is used to test identical strings across various algorithms. The next() function of this class will always return a new randomly created string of the specified size from the given alphabet.

Finally, SanityTest is a class which tests whether the lcs or lcs length generated by the algorithms is, in fact, a subsequence of the input strings. This tool was extremely useful in the development of the algorithms.

3.2 Methodology

Using the algorithms and utilities discussed above, two main programs were developed to thoroughly test the performance capabilities of the algorithms.

The LCS class takes the name of the algorithm you wish to run, the number of iterations of that algorithm to average, the minimum length and maximum length of strings to test, and the delta to increment string sizes. Although the program does not take an alphabet in the arguments, the alphabet is set in the first few lines of code. The random string generator is constructed using the alphabet specified in the code. Then, for the starting string size to the ending string size, do the following:

1. Set the string generator's size to the current size.
2. Run the solver on randomly generated strings of the given length a number of times equal to the given number of iterations.
3. Average the total time and recursive calls.
4. Print out the information to the screen and to a file.

Because of the structure of the solver classes and the utilities, this relatively simple class can do a lot of heavy lifting.

The other test program operates by stopping the execution of the program when the average completion time of a given algorithm exceeds the specified time in milliseconds. Both programs export their data directly to a .csv file for easy compilation and analysis.

The value for interations was set to 15 for every expiriment done in this report. We were getting fairly stable numbers at this mark and it kept test run times down within reason.

4. ALGORITHMIC COMPARISONS OF RECURSIVE FUNCTIONS

First, let's examine the output of each algorithm individually. With the naive recursive algorithm, the exponential nature of the algorithm is extremely apparent. By the time you get to a string size of 20, it is taking approximately 9 seconds of execution time with almost 3×10^8 recursive calls. However, the assumption that the algorithm is exponential is verified in figure 5.

By comparison, the recursive algorithm with memoization appears to do a lot better (figure 6). However, it fails to surpass exponential speed (figure 7). Notice that the algorithm fails to handle a string size greater than 500 in less than 10 seconds. Because of the nature of the algorithm, even attempting string sizes from the next two algorithms would be folly.

It's important to note that not a lot of time was spent dissecting the two recursive algorithms. They are, to say the least, inefficient and simply fail when the string size gets too large. For all moderately large computations, these algorithms do not solve the lcs problem in the lifetime of a human being. Consequently, deeper analysis is shown with the two other algorithms.

5. DYNAMIC PROGRAMMING AND HIRSCHBERG'S ALGORITHM

As stated above, these two algorithms handle the lcs problem in a much better way. Figure 8 shows the execution times of both dynamic programming and Hirschberg up to a string size of 10,000. The fact that these algorithms can compute the lcs of two strings 10,000 characters long in under four seconds for dynamic programming and under 1.5 seconds (!) for Hirschberg is an astonishing improvement over the recursive implementations.

However, in both figure 8 and figure 9, dynamic programming fails to outperform Hirschberg. This is in stark contrast to our expectations going into the experiments. In fact, it was this discrepancy that caused us to look into the flaws with enums mentioned earlier. We also explored the possibility that Java's garbage collection was interfering with the timing of the algorithm and/or memory usage was slowing down the machine. In order to address this, we tried suggesting that Java run its garbage collector after each solution for a given string size is returned. Figure 10 shows our different attempts at this process. The red line is our dynamic programming algorithm with no modifications to the garbage collector cleanup process. The green line represents the algorithm with a suggestion to garbage collect. The purple line is the same as before except with the main thread being forced to sleep for 1 millisecond. This effectively forces the garbage collector to run and frees up memory on a more steady basis. As you can see, the purple line has no unexpected peaks throughout its entire execution. Still, even with our modifications to the dynamic programming algorithm, we were unable to beat Hirschberg.

In terms of time complexity, the trendline in figure 10 clearly puts both dynamic programming and Hirshberg into $O(N^2)$ with Hirshberg apparently having a lower big-o coefficient.

5.1 Other Tests

Because Hirshberg is the most efficient and most stable algorithm at our disposal, we tested other elements of the algorithm using Hirshberg.

A test with fairly straight forward results measured the execution time using a string that was twice the size of the other. As figure 11 shows, the algorithm plots the normal execution (blue line) against the execution when one string is larger than the other(red line). This reduction in time is due to the smaller K matrix dimension or a reduced number of times the matrix needs to be updated. Furthermore, because the algorithm will hit a recursive base case sooner with one smaller string, the number of recursive calls is also reduced.

The other expiriment that had a slightly more interesting result was the alphabet comparison test. Our hypothesis was that alphabet size wouldn't matter in the slightest. After all, it is no more difficult (seemingly) for a computer to check of 'T' == 'T' than it is to check if 'A' == 'A' or 'L' == 'L'. If anything, we would have thought that the alphabet of 0,1 would have produced a faster run. Alas, our intuition seems to have betrayed us as the exact opposite appears to be true. The standard alphabet clearly outperforms the A,C,G,T alphabet which in turn outperforms 0,1. I honestly have no sound explanation for this result and feel as though this particular topic could be appropriate for an entirely different research paper. Figure 12 shows the comparison between the standard alphabet (blue line) and A,C,G,T (red line).

6. APPENDIX

Several other datasets were created that played around with the various algorithms. They were not included in this report because they did not add any new information. However, some information that may have been glossed over or downplayed in significance is included in the subsequent graphs and charts. The figures associated with the appendix will be captioned with an (A) preceeding the title.

Please take particular note of the Hirshberg Stress Test (Figure 17) as this shows what the Hirshberg algorithm is capable of doing with extremely large string sizes. In fact, if you extrapolate the regression equation out, it determines that it can do a string size of 1,000,000 in approximately 90 minutes.

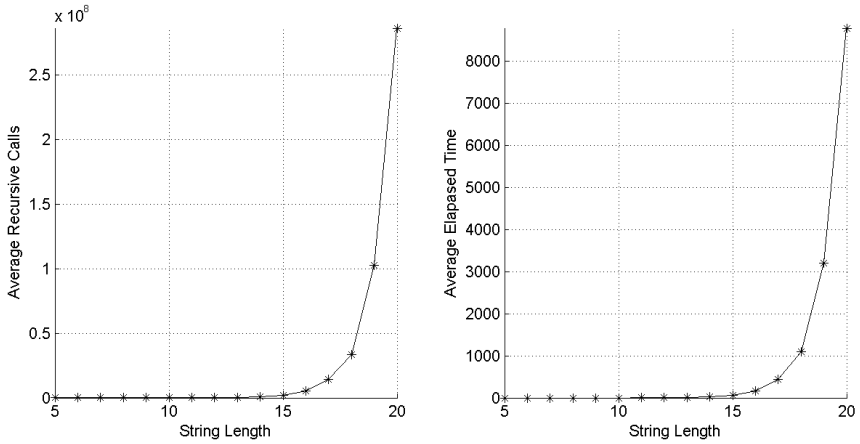

Figure 4: Naive Recursive:Avg. Recursive Calls and Avg. Time

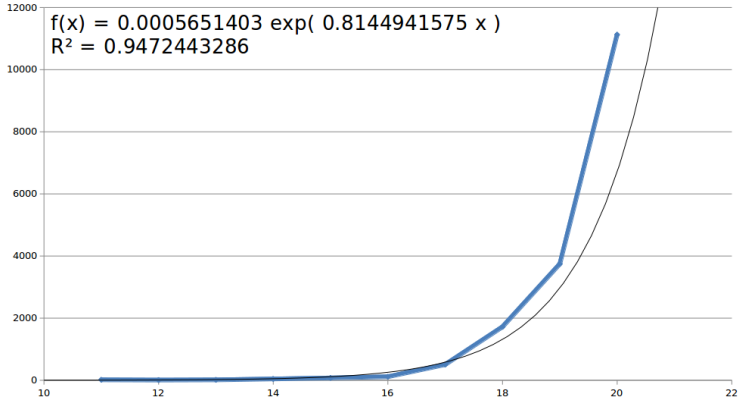

Figure 5: Regression of Naive Recursive

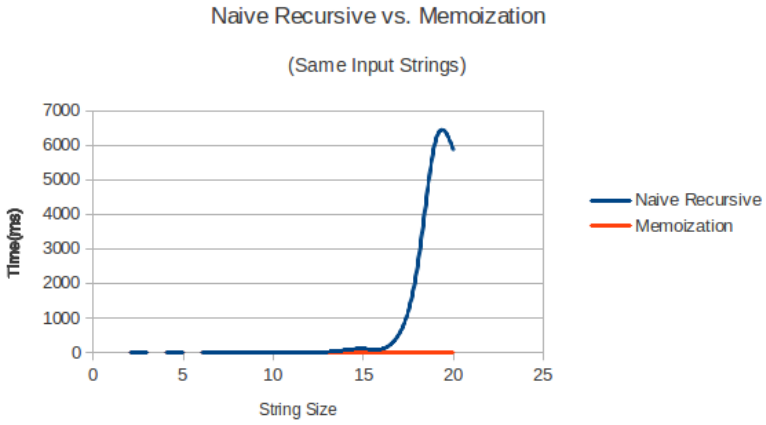

Figure 6: Comparison of Recursive Algorithms

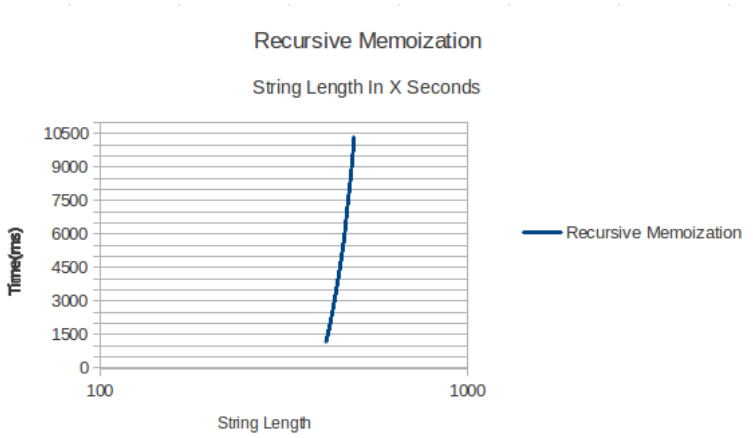

Figure 7: Recursive Memoization in Exponential Time

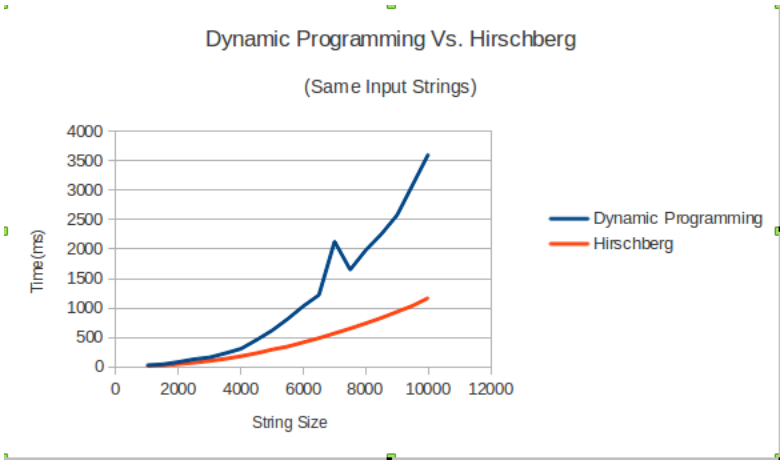

Figure 8: Dynamic Programming vs. Hirschberg

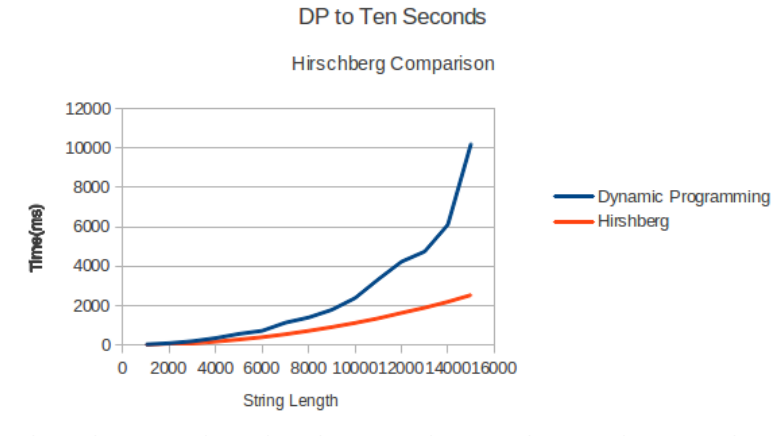

Figure 9: Dynamic Programming to 10 Seconds

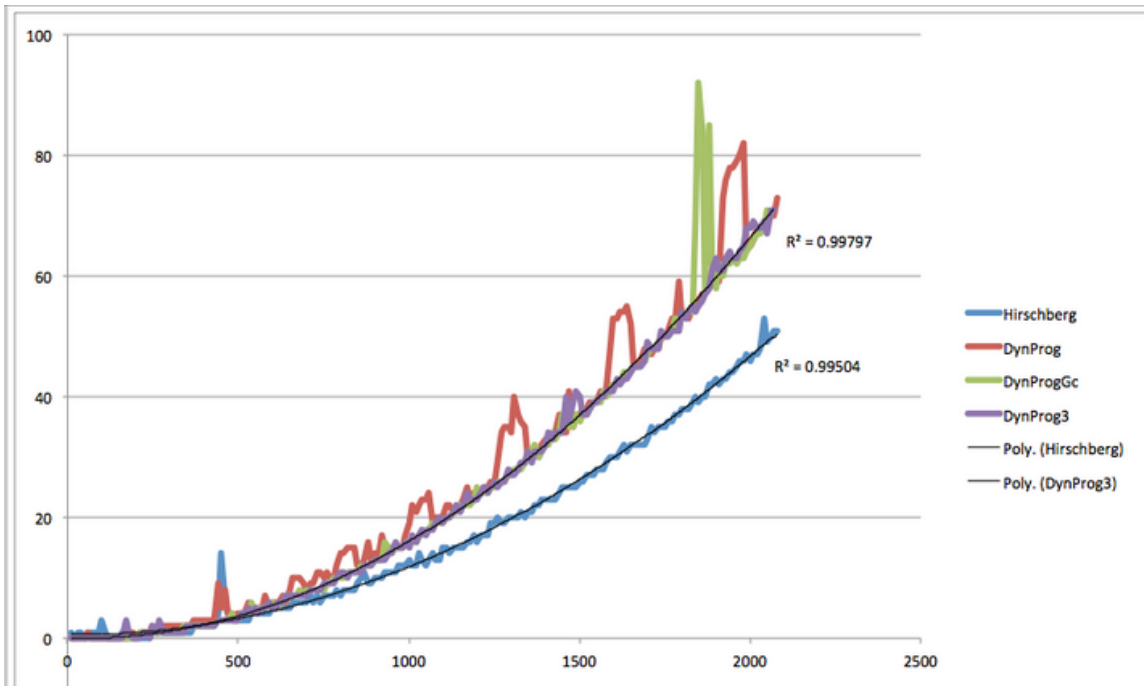

Figure 10: Dynamic Programming Garbage Collection

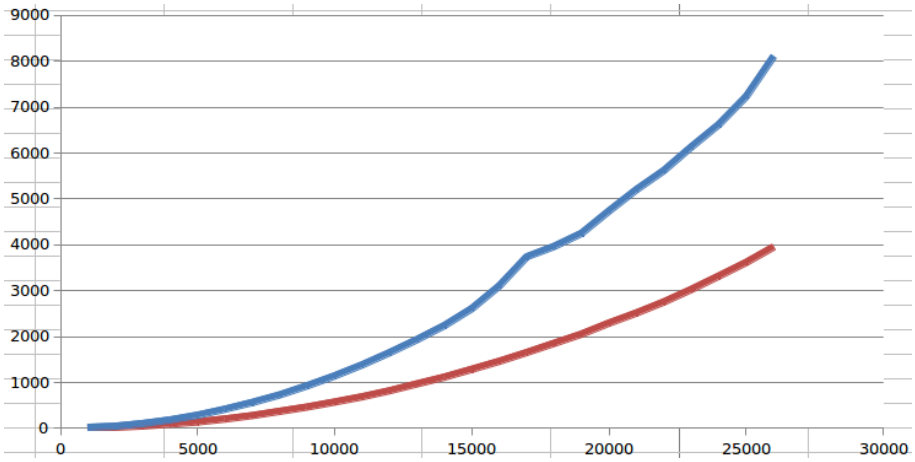

Figure 11: Different Size Strings

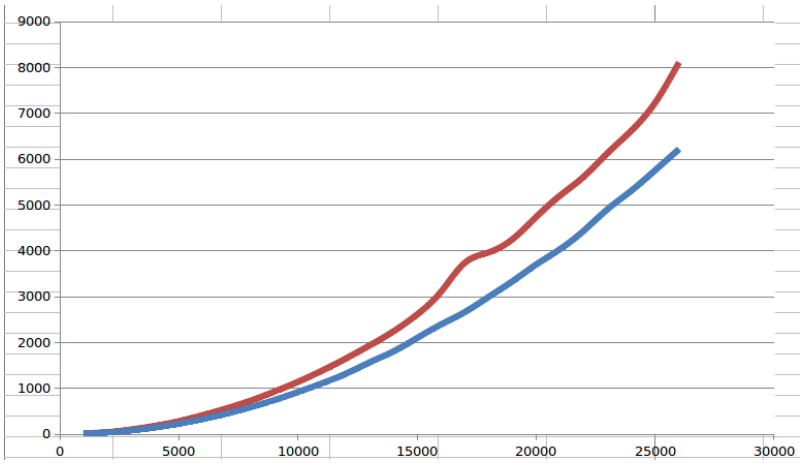

Figure 12: Different Size Strings

String Lengths Handled in Under 10 Seconds

Naive Recursive	Recursive Memoization	Dynamic Programming	Hirschberg
19	490	15000	30000

Figure 13: (A)Max String Sizes Doable in Under 10 Seconds

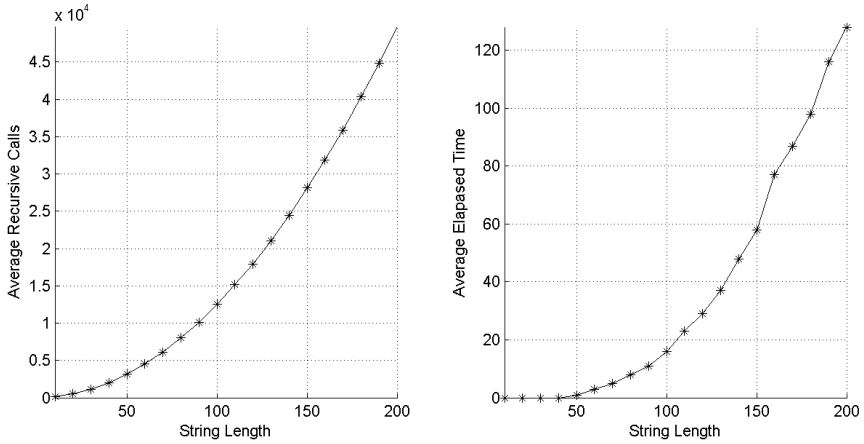

Figure 14: (A)Recursive Memoization Recursive calls vs. Time

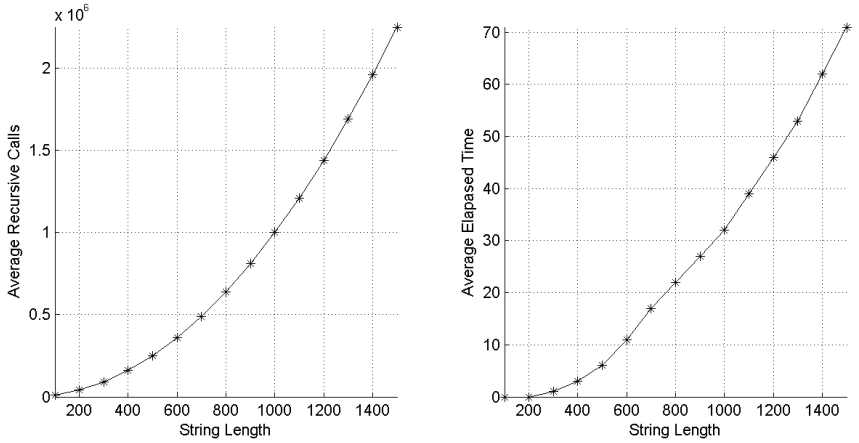

Figure 15: (A)Dynamic Programming Recursive calls vs. Time

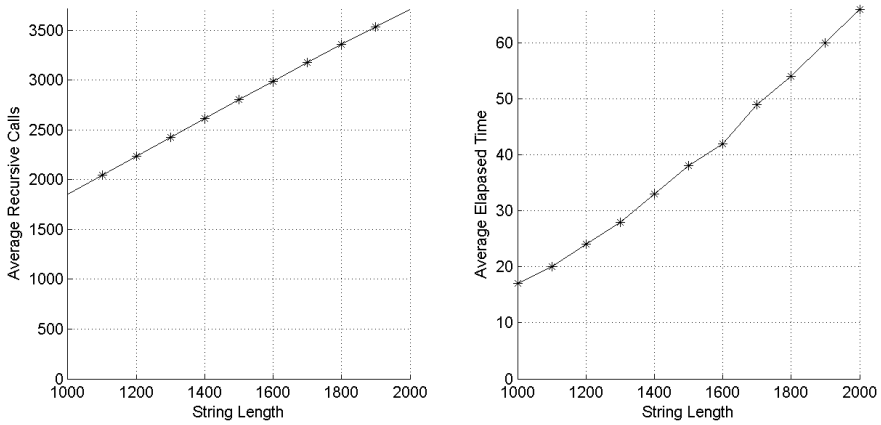

Figure 16: (A)Hirschberg Recursive calls vs. Time

40000	10353
45000	11829
50000	14571
55000	17885
60000	21595
65000	24831
70000	29068
75000	32918

Figure 17: (A)Hirshberg Stress Test (Only Finding Length)