Experimental Project:
Common Subsequence Algorithms

Maksim Yegorov

CSCI-665 @ RIT, Winter 2016

Prof. Stanisław Radziszowski

May 10, 2016

Abstract

I implement and compare side by side four algorithms that compute the length of and reconstruct a longest common subsequence (LCS) of two arbitrary strings. The asymptotic performance of the algorithms is compared to the actual execution times.

Table of Contents

..

List of Figures

..

List of Tables

Chapter 1

Introduction

In this report, I implement and investigate the performance of four algorithms that each calculate the length of and reconstruct a longest subsequences common to a pair of input strings. The algorithms are – in the order of increasing sophistication – the naive recursive, top-down memoized recursive, bottom-up dynamic iterative, and Hirschberg's quadratic time linear space recursive algorithms. The implementation of all algorithms except Hirschberg's quadratic-time linear-space algorithm is based on (Cormen & al. 2009). For Hirschberg's Algorithm B and Algorithm C, see (Hirschberg 1975).

This is an empirical investigation of the actual runtime performance. The algorithms where implemented using the `Python` programming language. `Python` is a high-level interpreted language. The reason that I chose `Python` is that it offers a near pseudocode-level clarity of the implementation. The drawback is a comparatively long execution time. For this exercise where we merely compare the algorithms among themselves – without worrying about putting them in production – `Python` proved to be an adequate choice, especially from the standpoint of rapidly coming up with a prototype implementation.

The algorithms were run in two batches remotely on a CS department lab machine (`gorgon.cs.rit.edu`) with the following characteristics:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 42
model name      : Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz
stepping        : 7
microcode       : 0x1b
cpu MHz         : 1600.000
cache size      : 6144 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
   ...

$ cat /proc/meminfo
MemTotal:       16391732 kB
```

```
MemFree:        12982560 kB
Buffers:          485152 kB
Cached:          1695260 kB
SwapCached:            0 kB
   ...
SwapTotal:      15998972 kB
SwapFree:       15998972 kB
   ...
```

Prior to running the experiments, I set artificially high system limits on my stack size so as to prevent the program from failing prematurely in the case of a deep recursion and force any bottleneck into CPU or memory capacity instead:

```
$ ulimit -s 120000 # (kilobytes)
```

and from within the `Python` script:

```
sys.setrecursionlimit(100000)
```

The flowchart in fig. 1.1 shows the overall logic of the driver script (`driver.py`):

Two batches of experiments were run in sequence. The input strings were chosen from two alphabets: a binary alphabet $\{0, 1\}$ and a four-item alphabet representing a quasi DNA strand $\{A, C, G, T\}$. Input string length was varied depending on the algorithm to ensure a reasonable runtime and memory requirements. Strings up to length 20 were used for naive algorithm, up to length 5,000 for the bottom-up dynamic and top-down memoized algorithms, and up to length 40,000 for the Hirschberg algorithm. All algorithms were run on the input strings from the same library randomly assembled for select string lengths using the `generate_string.py` module.

Each algorithm implements an essentially identical interface, so that they can all be run from the driver script with minimum variation. The `tabulate_lcs` function computes the matrix (or vector, as appropriate) of LCS lengths. The `reconstruct_lcs` function reconstructs an LCS.

The performance is measured separately for the tasks of

1) computing the length of an LCS, and
2) for reconstructing an LCS,

except for the *naive* algorithm, where the tasks are coupled.

Profiling the algorithms for time and memory usage is done by wrapping the above two functions in a `Python` decorator – a higher-order function that returns the original function, in addition to logging the time/memory resources. Similarly, to calculate the depth of recursion, I wrap the helper functions that are invoked recursively in a decorator that increments the recursion depth on each invocation. All of the profiling functions are defined in the `profilers.py` module. Here's a typical memory profiler output that my measurements are based on. Here I create a list of characters of length 10^6 with a footprint of approximately `8 MB`:

```
$ python3 profilers.py

Filename: profilers.py

Line #    Mem usage    Increment   Line Contents
================================================
   155     27.0 MiB      0.0 MiB       @time_and_space_profiler()
   156                                 def mem_test():
   157     27.0 MiB      0.0 MiB           a = 'a'
   158     34.7 MiB      7.7 MiB           b = ['a'] * (10**6)
   159     27.1 MiB     -7.6 MiB           del b
   160     27.1 MiB      0.0 MiB           return a
```

For each algorithm, I ran a suite of tests against hand-computed results to ensure the program performs as expected, as in the following assertion statements for the *top-down memoized algorithm*:

```
    print("[%0.7fs] %s(%d) -> %d recursive calls"
           %(elapsed, name, lcs_length, \
              registry['_reconstruct_lcs']))

    # test reconstruction match
    name, elapsed, memlog, lcs_table = \
           tabulate_lcs("","")
    lcs_length = size_lcs(lcs_table)
    waste, waste, memlog, lcs = \
           reconstruct_lcs("", "",
                  lcs_table, lcs_length)
    assert lcs == ""
    name, elapsed, memlog, lcs_table = \
           tabulate_lcs("","123")
    lcs_length = size_lcs(lcs_table)
    waste, waste, memlog, lcs = \
    reconstruct_lcs("", "123",
           lcs_table, lcs_length)
    assert lcs == ""
```

```
    name, elapsed, memlog, lcs_table = \
            tabulate_lcs("123","")
    lcs_length = size_lcs(lcs_table)
    waste, waste, memlog, lcs = \
        reconstruct_lcs("123", "",
                lcs_table, lcs_length)
    assert lcs == ""
    name, elapsed, memlog, lcs_table = \
            tabulate_lcs("123","abc")
    lcs_length = size_lcs(lcs_table)
    waste, waste, memlog, lcs = \
            reconstruct_lcs("123", "abc",
                    lcs_table, lcs_length)
    assert lcs == ""
    name, elapsed, memlog, lcs_table = \
            tabulate_lcs("123","123")
    lcs_length = size_lcs(lcs_table)
    waste, waste, memlog, lcs = \
            reconstruct_lcs("123", "123",
                    lcs_table, lcs_length)
    assert lcs == "123"
    name, elapsed, memlog, lcs_table = \
            tabulate_lcs("bbcaba","cbbbaab")
    lcs_length = size_lcs(lcs_table)
```

/home/max/classes/16_spring/algorithms/project/pylib/memoized.py

Also for verification purposes – for all strings against which the algorithms were tested – I plot the lengths of the reconstructed LCS's in fig. 1.2. This shows, as expected, two LCS matches for each input string length – consistent with two sets of inputs at each input string length (binary and DNA alphabet sets) – except where the two match strings have identical length or are indistinguishable on the plot scale for the shortest of inputs:

In addition to the `Python Standard Library`, I've used the `Python` `matplotlib` module for plotting and `memory_profiler` to track memory consumption. Both packages are under the BSD license.

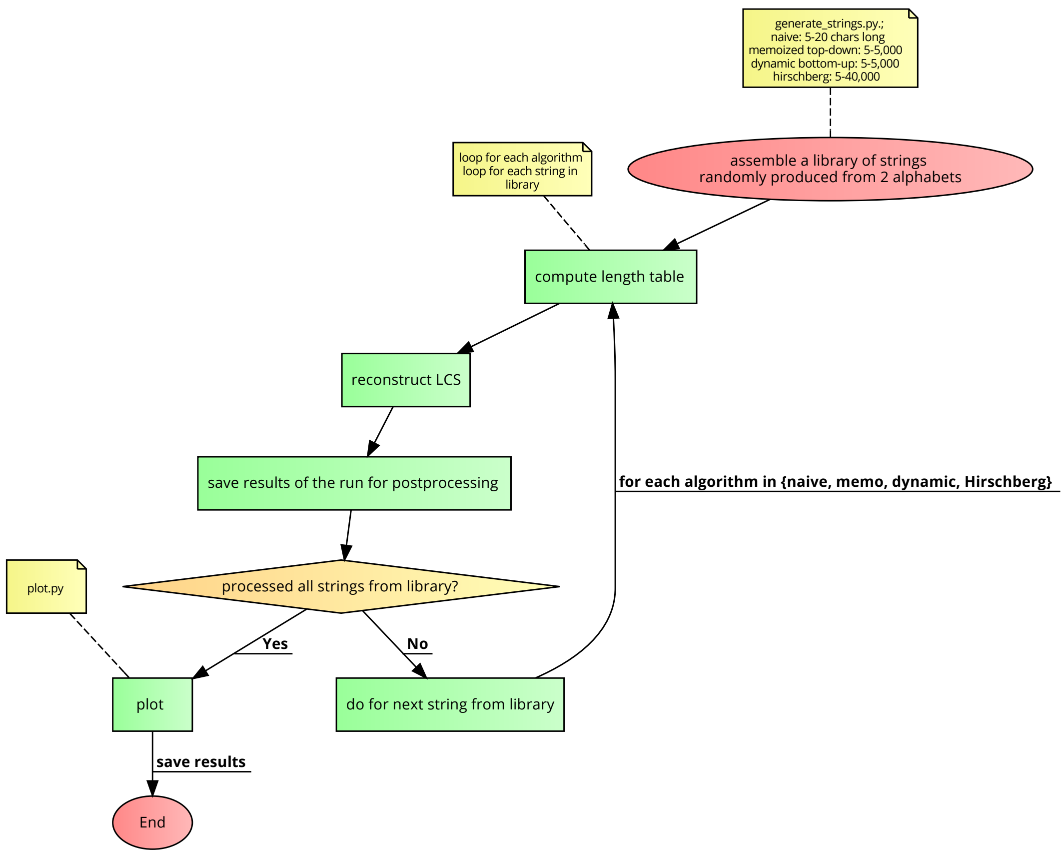

Figure 1.1: Logic of the batch script (`driver.py`)

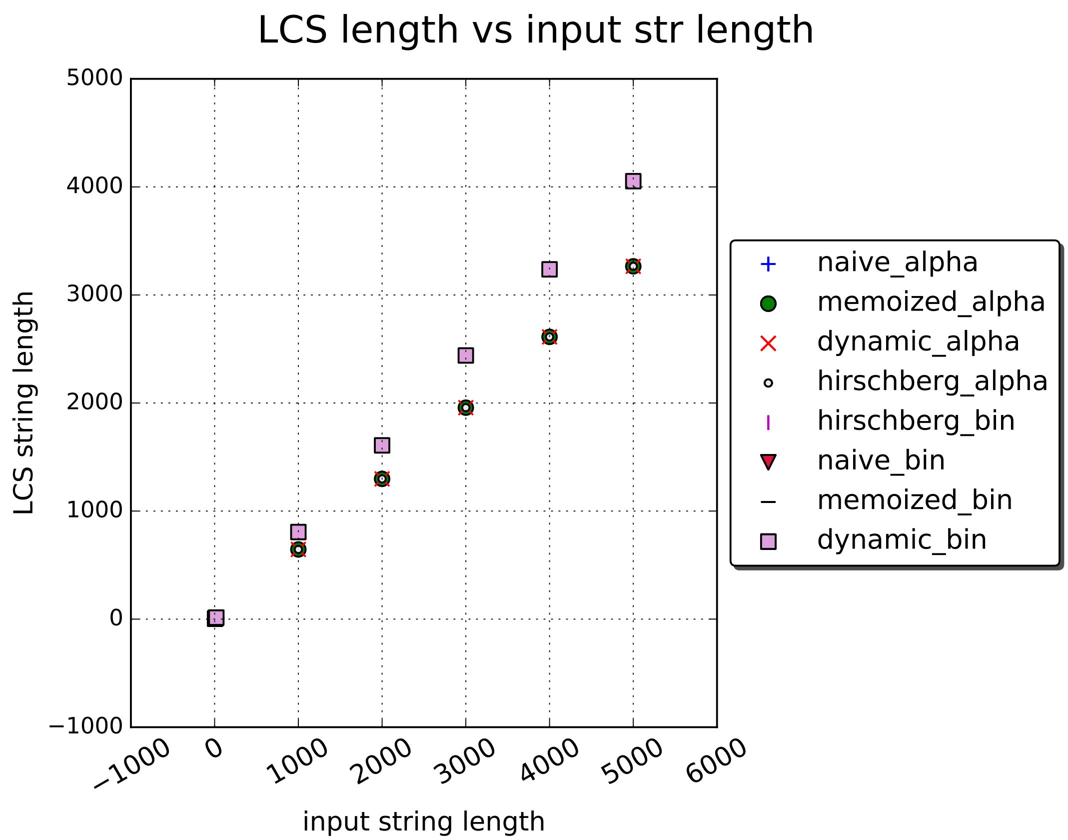

Figure 1.2: Sanity check: Verify all algorithms compute the same LCS length for a given pair of input strings

Chapter 2

Naive Algorithm

The naive recursive solution is based on recursion (15.9) in (Cormen & al. 2009). I repeat the recursion here as it is of fundamental importance for all the algorithms discussed in this report. For the `Python` implementation, see listing in sec. 7.

$$c[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_i, \\ max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases} \tag{2.1}$$

strings of length up to 20 were tested. Asymptotic complexity of the *naive recursive algorithm* is exponential in the length of the input strings. Both the length and the actual LCS match are computed at once. The asymptotic time is confirmed by the experimental results shown in sec. 6, where the performance of the *naive algorithm* is orders of magnitude worse than that of any of the quadratic/linear time algorithms.

The stark difference in performance is most clearly seen in set 2 plots (runtime vs input and recursion depth vs input) in sec. 6 and correlates strongly with the recursion depth, but is further compounded by the repeated re-calculation of the same quantities.

We could fit an exponential curve to the $O(c^n)$ data distribution to approximate the constant c, under the assumption that all lower-order terms are negligible. However, the added precision is not very useful as the numbers will differ, sometimes dramatically, even between different batch runs, to say nothing about different machines. Compare the runtime for strings of size 20 (binary alphabet) for the two sets.

To illustrate, for set 1 we can approximate the distribution with:

1) for alphabetic strings: $CPU\,time = 7.3 \times 10^{-5}\ e^{0.83n}$;
2) for binary strings: $CPU\,time = 2.5 \times 10^{-3}\ e^{0.25n}$;

For set 2:

1) for alphabetic strings: $CPU\,time = 1.8 \times 10^{-4}\ e^{0.80n}$;
2) for binary strings: $CPU\,time = 1.5 \times 10^{-3}\ e^{0.29n}$;

With the above dislaimer about the approximate nature of any prediction (specific to machine and input characteristics), we can estimate that for 10 second execution time, we can process at input strings of at most length 13 if using DNA alphabet.

Chapter 3

Top-Down Memoized Algorithm

The memoized implementation uses top-down recursion essentially identical to the naive approach in sec. 7, except that performed computations are saved in a table to eliminate repeated superfluous calculations. For the `Python` implementation, see listing in sec. 8.

We expect $\Theta(mn)$ running time and memory requirements for the task of sizing an LCS. Also, we expect linear time $\Theta(m+n)$ and quadratic space $\Theta(mn)$ for reconstructing an LCS, given the table computed beforehand.

It will be seen in sec. 6 that the *memoized algorithm* has indeed quadratic execution time and memory performance for sizing an LCS, and linear time for reconstructing an LCS. See, in particular the runtime and memory plots from set 2 in sec. 6.

For set 2 we can approximate the distribution of CPU times for sizing an LCS with:

1) for alphabetic strings: $CPU\,time = 7.28 \times 10^{-5}\ x^2$;
2) for binary strings: $CPU\,time = 1.75 \times 10^{-8}\ x^2$;

Correspondingly, for a 10 second runtime, we could process inputs up to about size 460 characters.

Chapter 4

Bottom-Up Dynamic Programming Algorithm

The DP implementation uses bottom-up iterative approach in `Fig. 15.8` in (Cormen & al. 2009). For the `Python` implementation, see listing in sec. 9.

As with the *memoized algorithm*, we expect $\Theta(mn)$ running time and memory requirements for the task of sizing an LCS. Also, we expect linear time $\Theta(m+n)$ and quadratic space $\Theta(mn)$ for reconstructing an LCS, given the table computed beforehand.

As the plots in sec. 6 demonstrate, the *dynamic algorithm* has indeed quadratic execution time and memory performance for sizing an LCS, and linear time for reconstructing an LCS.

It will be seen from the plots in sec. 6 that the *dynamic algorithm* implementation is more efficient than *memoized algorithm* because of the recursive overhead of the latter. However, my table storage implementation for the two algorithms is differenty (by accident). The table for the *dynamic algorithm* just happens to be less efficiently implemented. This results in the *dynamic algorithm* requiring significantly more memory for the same input length, compared to my implementation of the *memoized algorithm*. Again, this is a mere fluke of implementation and not in any way intrinsic in the algorithms themselves. I will comment on the particular plots that illustrate this fluke further in sec. 6.

For set 2 we can approximate the distribution of CPU times for sizing an LCS for both alphabetic and binary strings: $CPU\,time = 5.79 \times 10^{-5}\ x^2$;

Correspondingly, for a 10 second runtime, we could process inputs up to about size 420 characters, which approximately matches the performance of the *memoized* algorithm.

Chapter 5

Hirschberg Linear Space Dynamic Programming Algorithm

The *Hirschberg algorithm* implementation follows the pseudo-code in (Hirschberg 1975). For the `Python` implementation, see listing in sec. 10.

Theoretically, we expect $\Theta(mn)$ time complexity and $\Theta(m+n)$ space. By distinction from the *memoized* and *dynamic* algorithms that require quadratic ($\Theta(mn)$ space for recovery, not just sizing an LCS), *Hirschberg* algorithm allows one also to recover an LCS in $\Theta(m+n)$ space. However, also by contrast to the *memoized* and *dynamic* algorithms, *Hirschberg* requires a $\Theta(mn)$ time to recover an LCS, where the former two algorithms are linear $\Theta(m+n)$. I.e. in the tradeoff between time and memory consumption – the former two algorithms excel in the time requirements (for recovering an LCS), while *Hirschberg* excels in the space requirements (similarly for recovering an LCS).

The linear space requirements and polynomial time requirements will indeed be evident in the plots in sec. 6.

For set 2 we can approximate the distribution of CPU times for sizing an LCS with:

1) for alphabetic strings: $CPU\,time = 5.09 \times 10^{-6}\ x^2$;
2) for binary strings: $CPU\,time = 5.15 \times 10^{-6}\ x^2$;

Correspondingly, for a 10 second runtime, we could process inputs up to about size 1350 characters, almost three times the performance of the *memoized* or *dynamic* algorithms.

Chapter 6

Summary of results

Two sets of experiments have been performed. They show the same tendencies, but the actual execution time and memory usage occasionally differs, which demonstrates the vagaries of attaching too much precision beyond the approximate asymptotic estimates. In this section, I compare experimental runs side by side.

6.1 Set 1

Note that I distinguish the tasks of sizing and reconstructing the LCS for all algorithms except the *naive algorithm*. From fig. 6.1 it can be seen that the execution time of the three algorithms for sizing LCS (excluding *naive*) is quadratic in the length of input string. What is truly remarkable is how much more efficient Hirschberg's *Algorithm B* is compared even to its very close cousin *dynamic* *bottom-up algorithm*. Essentially, the only difference between the algorithms is that the *dynamic bottom-up algorithm* keeps an in-memory matrix of lengths that is the size of Hirschberg's vector in-memory storage squared.

fig. 6.2 demonstrates vividly the inefficiency of the *naive algorithm* that takes longer than a Hirschberg's algorithm on an input that is three orders of magnitude naive's. One can also clearly see the quadratic nature of Hirschberg's reconstruction scheme (for the CPU time, as opposed to memory usage). Compare this to linear time reconstruction algorithms (*dynamic* and *memoized*).

fig. 6.3 illustrates the difference between recursive and iterative algorithms. For the recursive *memoized algorithm* (*naive* not shown, as it performs reconstruction coupled with sizing the LCS), one can see the quadratic nature of recursion depth vs. input string length. This will become even clearer on set 2 plots below. By distinction, *dynamic* and *Hirschberg* algorithms are iterative.

Finally, fig. 6.4 shows the quadratic relationship between memory usage and input length for *dynamic* and *memoized* algorithms, as opposed to linear relationship for *Hirschberg*, which barely grows for its very low footprint.

6.2 Set 2

The second run has broadly comparable results. Remarkably, there are sometimes dramatic differences, which demonstrates the risk of estimating the runtime or memory consumption with more precision than can be justified. Compare for example the tables in sec. 15 for the *dynamic* runs 1 and 2 for input of size 5000.

For better resolution, the plots for set 2 exclude the runs for inputs of size above 5,000 (see instead set 1 plots for *Hirschberg algorithm* inputs for sizes $> 5,000$).

We observe from the plots that alphabetic input matching appears to be less efficient than binary. This is probably due to the fact that the longer length of matched strings (for the quasi-random algorithm I used in generating input strings) results in faster "convergence" for binary strings compared to DNA strings. Refer to fig. 6.9 and to fig. 1.2. It would be interesting to compare the efficiency if the length of match were controlled for.

Memoized scheme is less efficient than *dynamic*, which is probably due to the overhead from recursion (vs. iterative implementation of the *dynamic* algorithm). *Hirschberg's* implementation (also iterative), trumps *dynamic* by far in virtue of its lean operations on vector storage of the LCS lengths (vs. 2D matrix in case of the *dynamic* algorithm). It should be mentioned that I used the rather inefficient storage scheme using $m \times n$ sized lists from Python's Standard Library instead of using arrays from the outside `numpy` library that are much more compact and efficient.

With reference to fig. 6.6: Reconstructing an LCS match using the naive algorithm is tremendously inefficient. The distinction between exponential and polynomial algorithm is evident in this plot, where maximum-length *naive* input is 20, evidently due to its wastefull recursive calls. Note the depth of recursion in fig. 6.7 even for such a small input size.

With reference to fig. 6.9: For recursive algorithms, the quadratic relationship between recursion vs input length mirrors that between CPU time vs input length. There's a linear relationship between recursion depth and CPU time for the recursive *memoized algorithm*.

It is interesting to note that it takes about twice as many recursive calls for an alphabetic string compared to binary string – for the same algorithm and string length input! Note that the DNA alphabet is also twice the size of the binary alphabet. Again, I suspect this is due to the longer match and correspondingly faster convergence, which is accidental, in the sense that it is not intrinsic to the alphabet representation in my case but is just a fluke of string generation.

With reference to fig. 6.10, the memory usage is also quadratic in the length of input for all algorithms, except *Hirschberg's*, which is linear as expected (barely noticeable footprint). This is expected for 2D tables. Also, one notes the difference between the *dynamic* and *memoized* memory usage for the **same** input strings! This is not due to anything intrinsic in the algorithms. One would expect that the two algorithms would have identical memory usage. The difference is explained by my implementation: I just happened to use very sparsely populated arrays (mostly filled by `None` pointers) for the *memoized* implementation. Whereas, all entries in the *dynamic* arrays are initialized to `0`. I didn't put much thought into the difference of implementation, but it obviously led to some dramatic difference in memory usage.

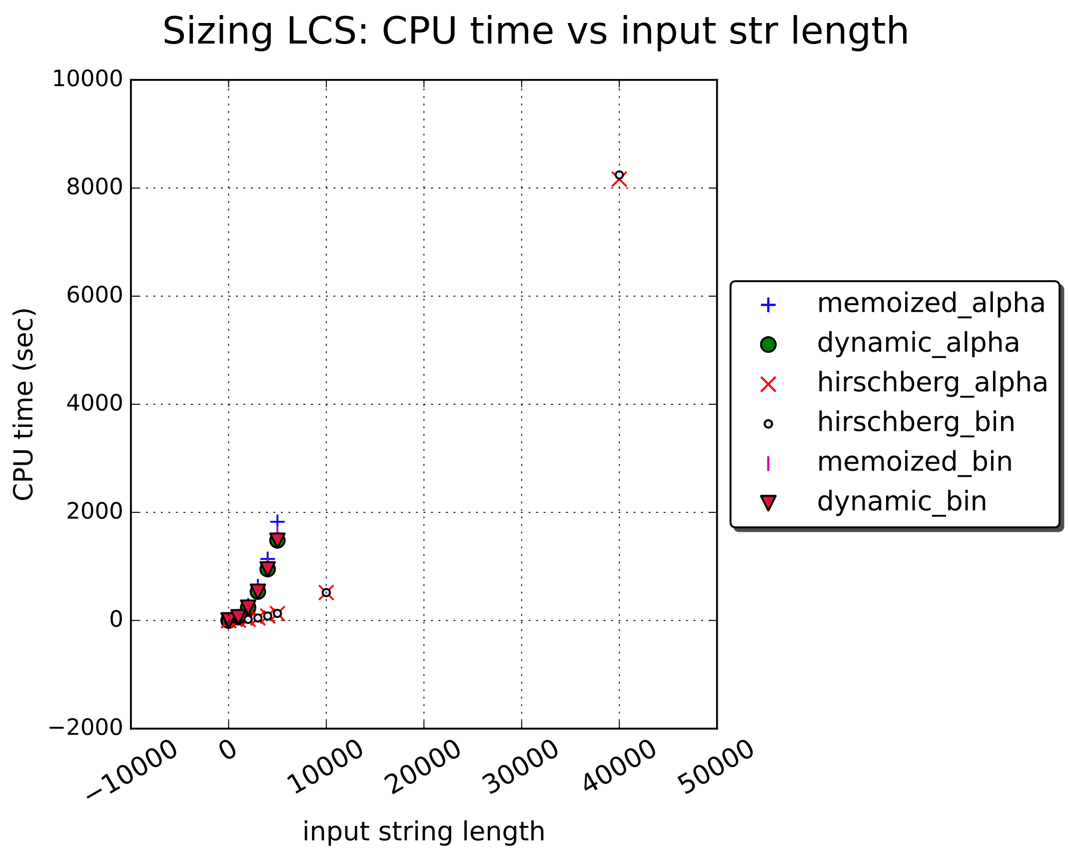

Figure 6.1: Set 1: Runtime vs input length – sizing LCS

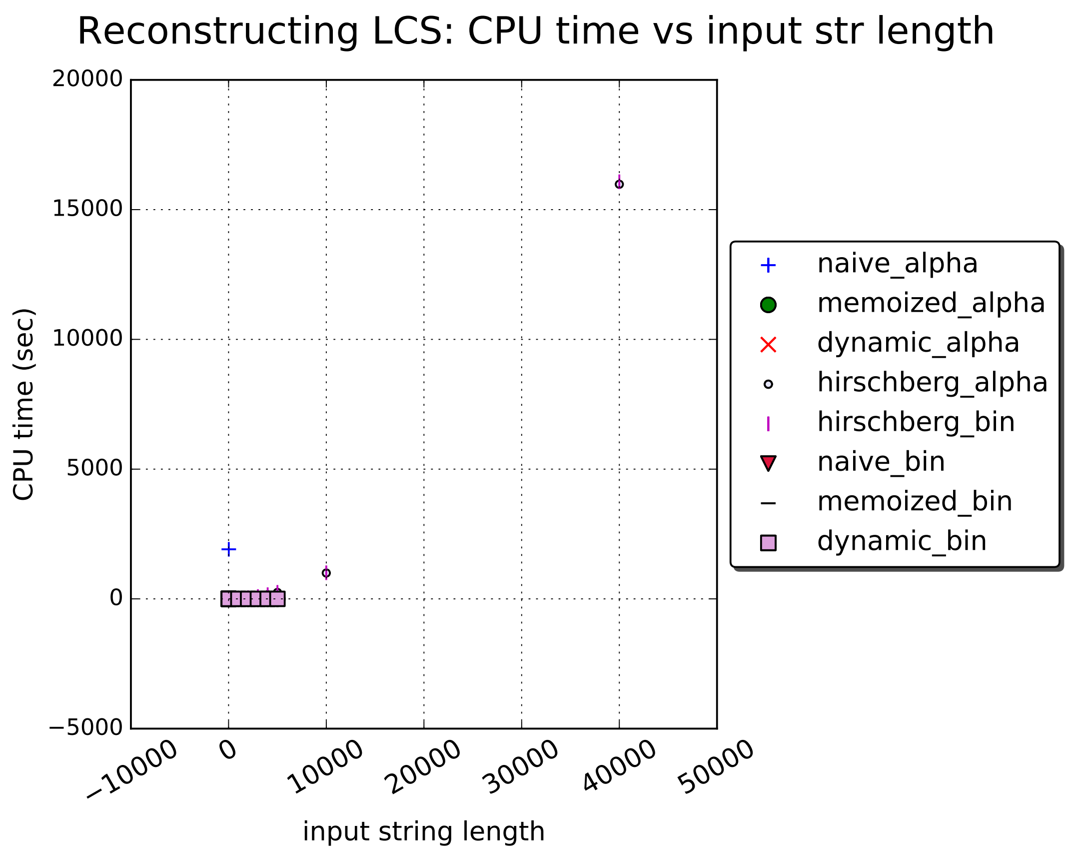

Figure 6.2: Set 1: Runtime vs input length – reconstructing LCS

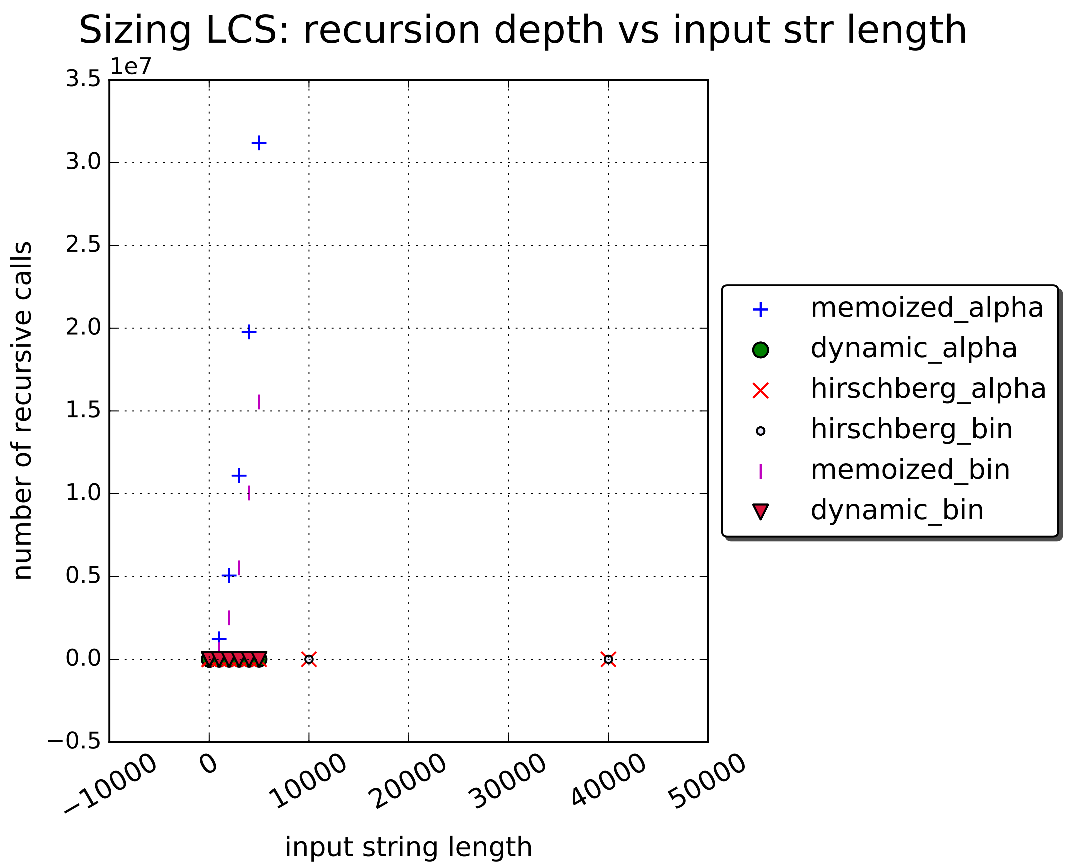

Figure 6.3: Set 1: Recursion depth vs input length – sizing LCS

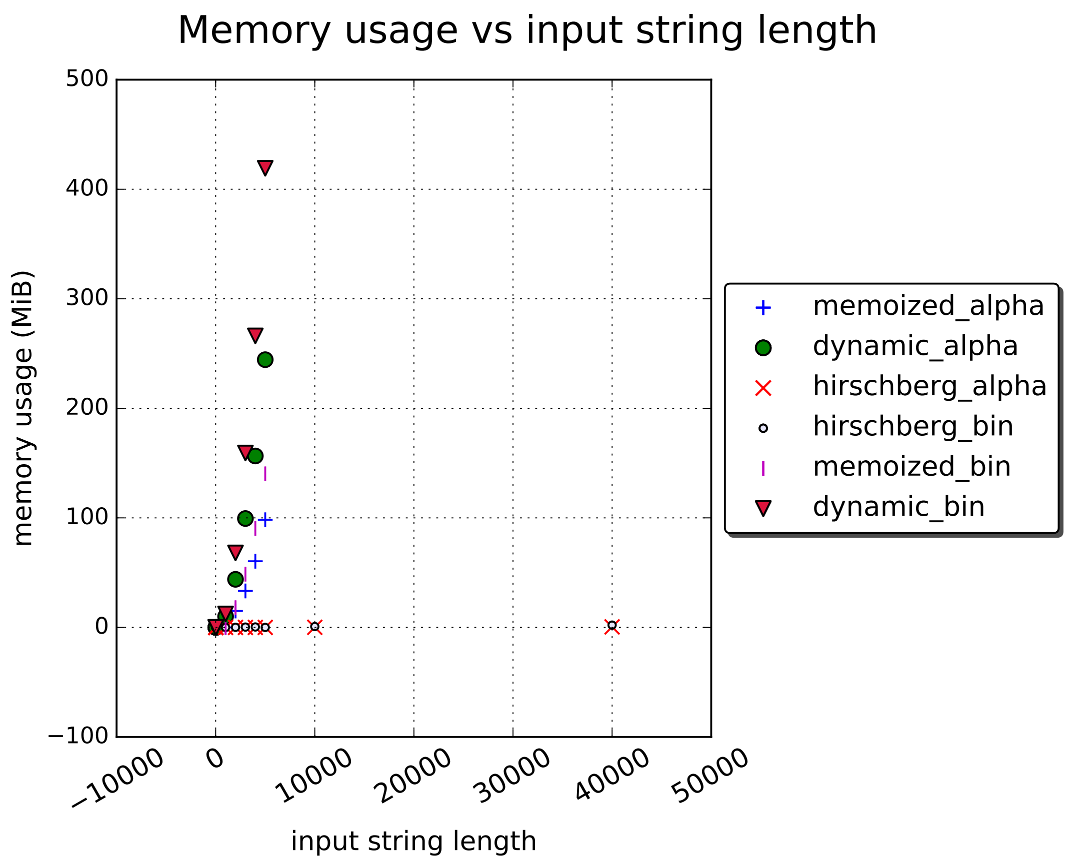

Figure 6.4: Set 1: Memory usage – sizing LCS

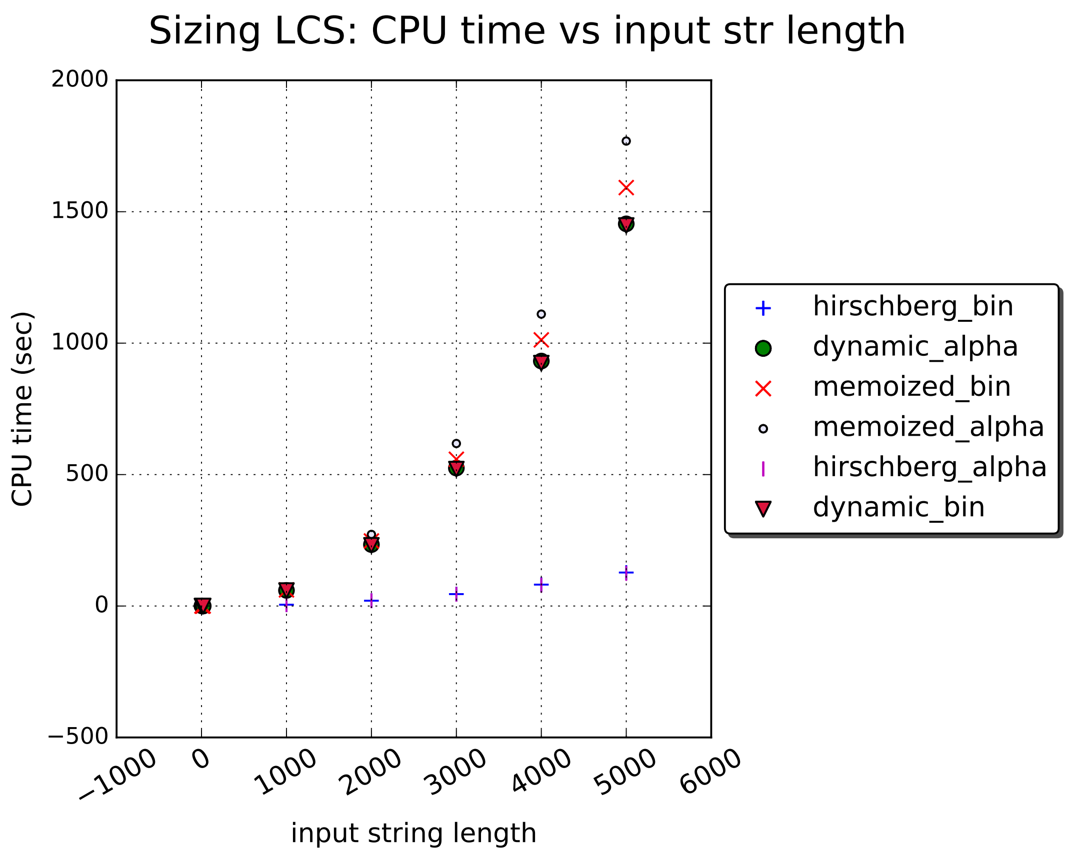

Figure 6.5: Set 2: Runtime vs input length – sizing LCS

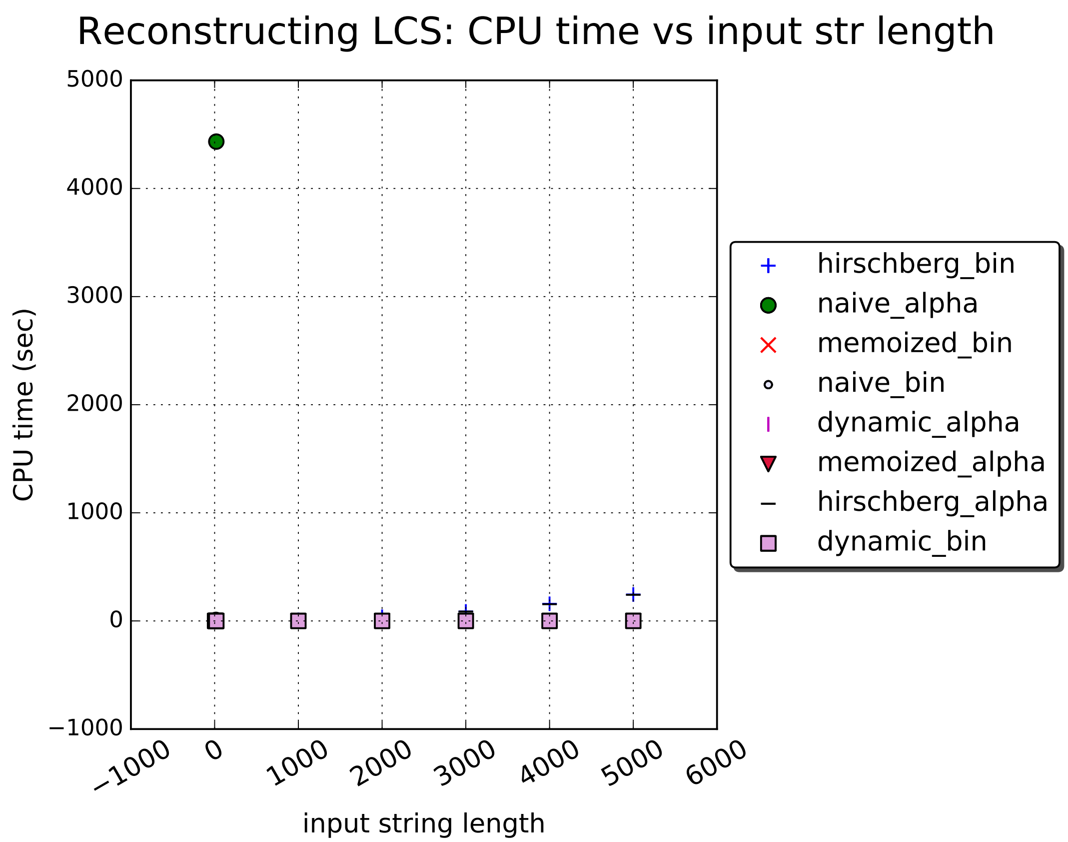

Figure 6.6: Set 2: Runtime vs input length – reconstructing LCS

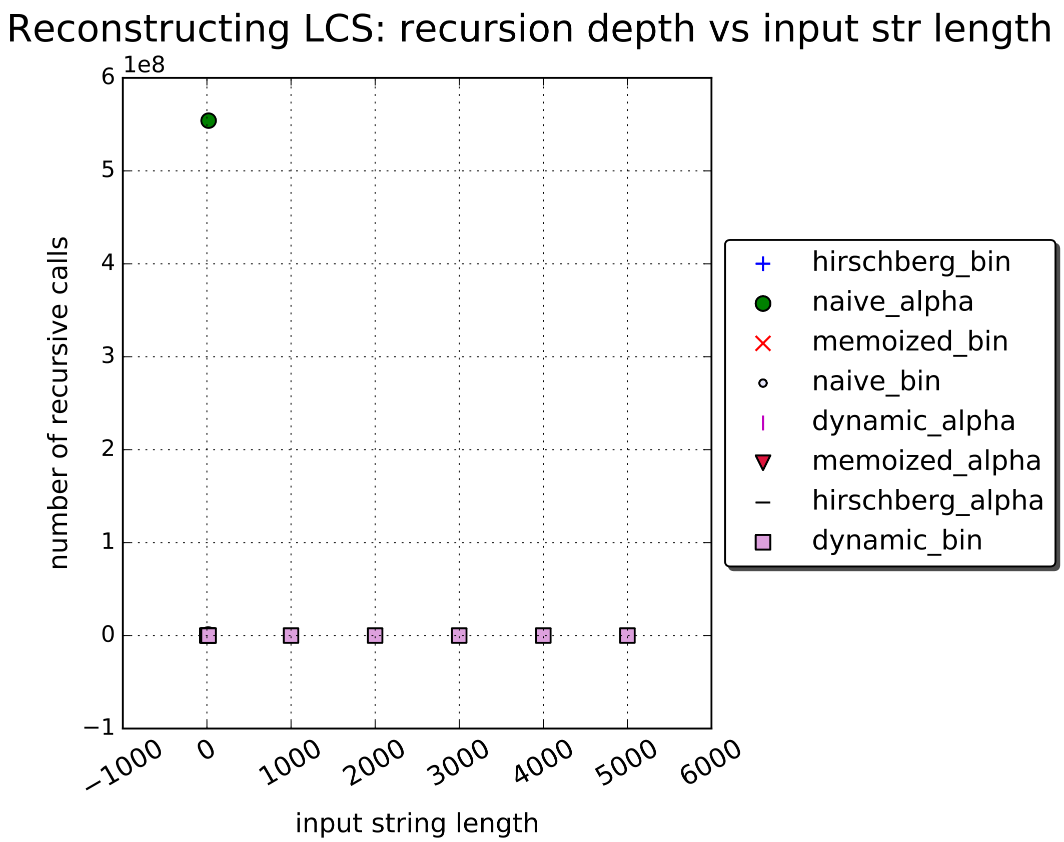

Figure 6.7: Set 2: Recursion depth vs input length – reconstructing LCS

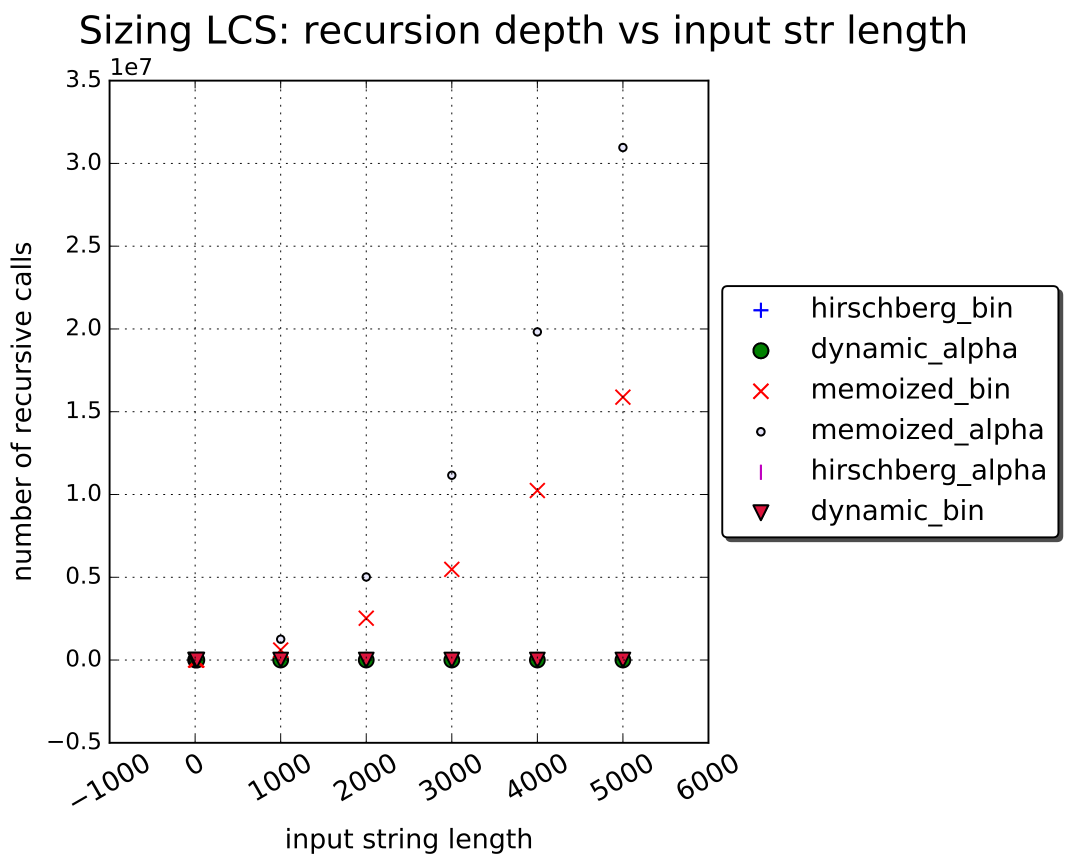

Figure 6.8: Set 2: Recursion depth vs input length – sizing LCS

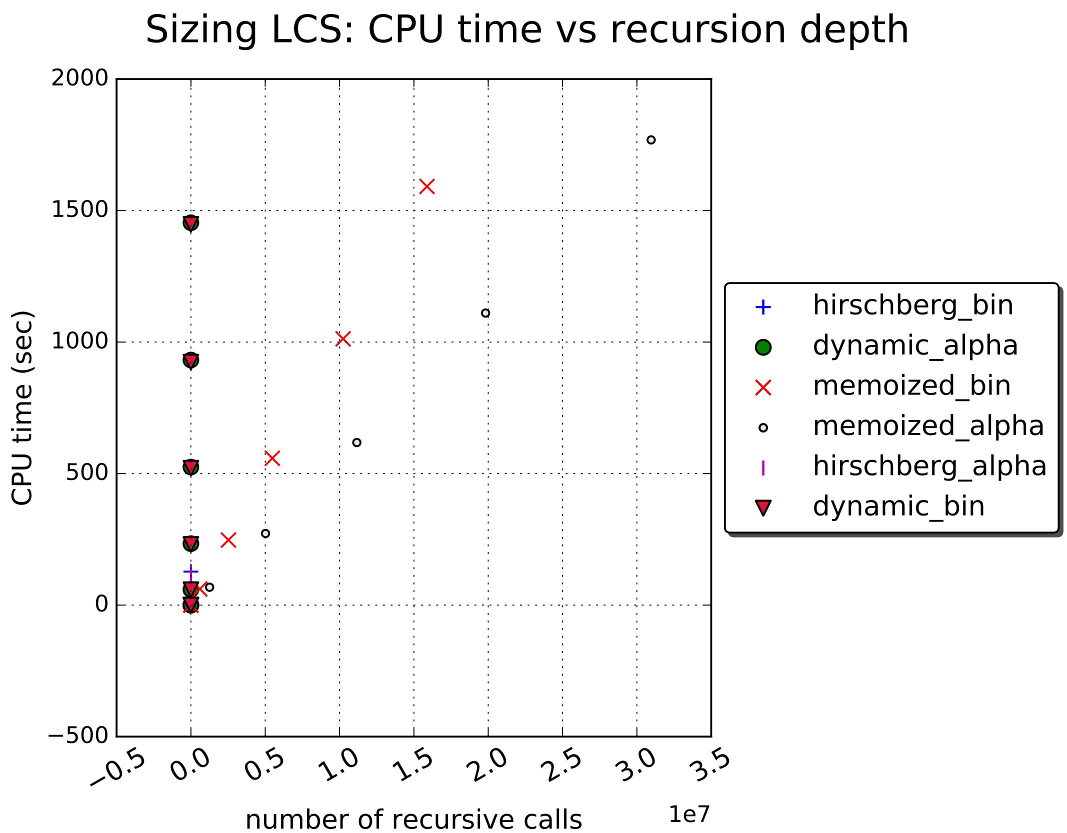

Figure 6.9: Set 2: Runtime vs recursion depth – sizing LCS

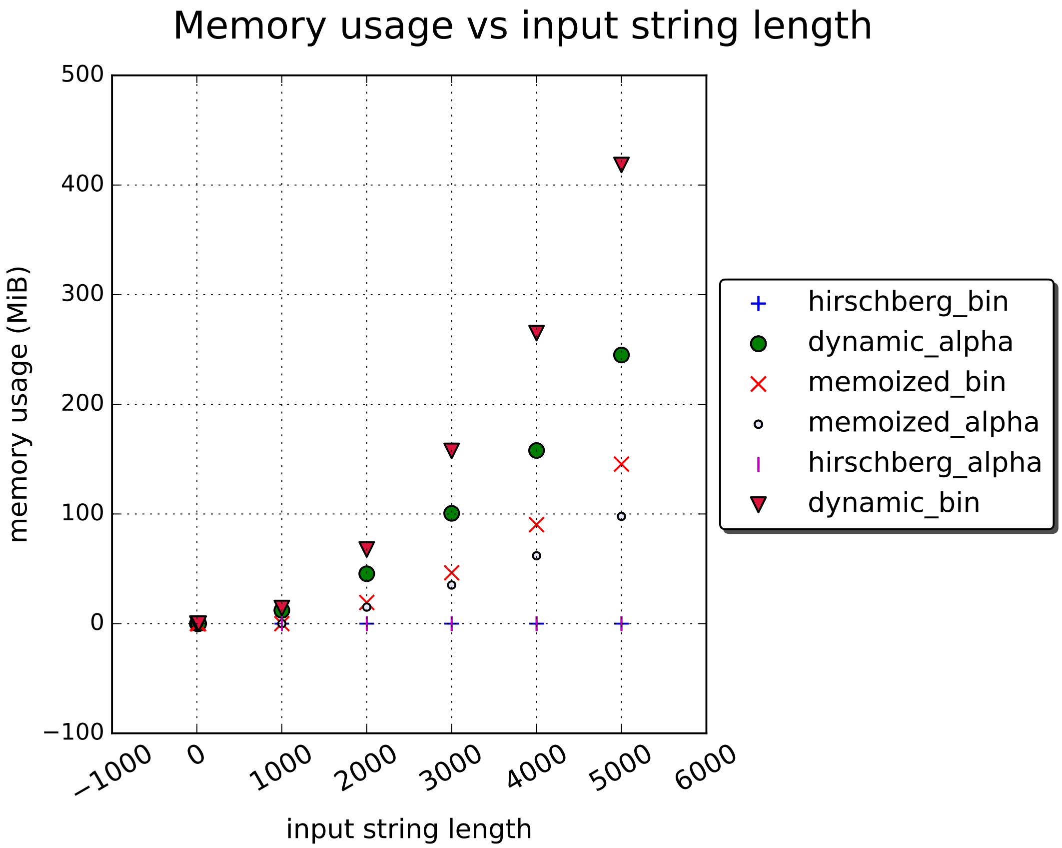

Figure 6.10: Set 2: Memory usage – sizing/reconstructing LCS

Chapter 7

Appendix 1: Naive Algorithm Implementation

Following is the implementation of the naive algorithm in sec. 2:

```
from profilers import log_recursion
from profilers import time_and_space_profiler
from profilers import registry
from generate_string import strgen
import sys

sys.setrecursionlimit(100000)


@time_and_space_profiler(repeat = 1)
def reconstruct_lcs(seq1, seq2, *args):
    """Calls helper function to calculate an LCS.

    Args:
        *args: extra arguments that some algorithms
               require
        """
    # reset registry
    registry['_reconstruct_lcs'] = 0

    return _reconstruct_lcs(seq1, seq2, len(seq1)-1, \
               len(seq2)-1, "")

```

```python
@log_recursion
def _reconstruct_lcs(seq1, seq2, i, j, lcs):
    """Naive recursive solution to LCS problem.
    See CLRS pp.392-393 for the recursive formula.

    Args:
        seq1 (string): a string sequence generated
                       by generate_string.strgen()
        seq2 (string): another random string
                             sequence like seq1
        i (int): index into seq1
        j (int): index into seq2
        lcs (string): an LCS string being built-up
    Returns:
        lcs: longest common subsequence (can be
                   empty string)
    """

    if i < 0 or j < 0:
        return lcs
    else:
        if seq1[i] == seq2[j]:
            return _reconstruct_lcs(seq1, seq2, \
                    i-1, j-1, seq1[i] + lcs)
        else:
            return max(_reconstruct_lcs(seq1, \
                            seq2, i-1, j, lcs),
                    _reconstruct_lcs(seq1, \
                            seq2, i, j-1, lcs), \
                            key=len)
```

/home/max/classes/16_spring/algorithms/project/pylib/naive.py

Chapter 8

Appendix 2: Memoized Algorithm Implementation

Following is the implementation of the memoized dynamic programming algorithm in sec. 3:

```
from profilers import log_recursion
from profilers import time_and_space_profiler
from profilers import registry
from generate_string import strgen
import sys

# set system recursion limit
sys.setrecursionlimit(100000)


@time_and_space_profiler(repeat = 1)
def tabulate_lcs(seq1, seq2, *args):
    """Calls helper function to calculate an LCS.

    Args:
        seq1 (string): a random string sequence
                generated by generate_string.strgen()
        seq2 (string): another random string
                sequence like seq1
    Returns:
```

```
        table of LCS lengths (int): so-called table
                c in Figure 15.8 in CLRS

    """
    # reset registry
    registry['_tabulate_lcs'] = 0

    len1 = len(seq1)
    len2 = len(seq2)

    # store length of LCS[i,j] in lcs_table
    lcs_table = [[None for j in range(len2)] \
                for i in range(len1)]
    _tabulate_lcs(seq1, seq2, len1-1, len2-1, \
                lcs_table)
    #return lcs_table[len1-1][len2-1]
    return lcs_table

@log_recursion
def _tabulate_lcs(seq1, seq2, i, j, lcs_table):
    """Recursive solution with memoization to LCS
    problem. See CLRS ex. 15.4-3.

    Args:
        seq1 (string): a string sequence generated by
                    generate_string.strgen()
        seq2 (string): another random string sequence
                like seq1
        i (int): index into seq1
        j (int): index into seq2
        lcs_table (2D list): a matrix of LCS length
                    for [i, j] prefix
    Returns:
        None: modifies in place LCS length table
    """

    if i < 0 or j < 0:
        return 0
    else:
        if lcs_table[i][j] is not None:
            return lcs_table[i][j]
        else:
            if seq1[i] == seq2[j]:
                val = 1 + \
                    _tabulate_lcs(seq1, seq2, i-1, \
                                j-1, lcs_table)
            else:
                val = max(_tabulate_lcs(seq1, seq2, \
                                i-1, j, lcs_table),
                        _tabulate_lcs(seq1, seq2, i, \
                                j-1, lcs_table))

            lcs_table[i][j] = val
```

```
            return val

def size_lcs(lcs_table):
    """Returns length of maximum common subsequence.

    Args:
        lcs_table (2D list): a matrix of LCS length for
                            [i, j] prefix
    Returns:
        length (int): LCS length
    """
    if len(lcs_table) > 0 and len(lcs_table[0]) > 0:
        return lcs_table[-1][-1]
    else:
        return 0

@time_and_space_profiler(repeat = 1)#, stream = MEMLOG)
def reconstruct_lcs(seq1, seq2, lcs_table, lcs_length):
    """Calls helper function to reconstruct
    one possible LCS based on saved LCS lengths table.

    Args:
        seq1 (string): a string sequence generated by
                        generate_string.strgen()
        seq2 (string): another random string sequence
                        like seq1
        lcs_length (int): length of LCS
        lcs_table (2D list): a matrix of LCS length for
                            [i, j] prefix
    Returns:
        lcs (string): an LCS
    """

    # reset registry
    registry['_reconstruct_lcs'] = 0

    i = len(lcs_table) - 1
    if i < 0:
        return ""
    else:
        j = len(lcs_table[0]) - 1
        lcs_arr = _reconstruct_lcs(seq1, seq2, lcs_table,
                lcs_length-1, i, j, [None] * lcs_length)
        lcs = "".join(lcs_arr)
        return lcs

@log_recursion
def _reconstruct_lcs(seq1, seq2, lcs_table, char, i, j,\
        lcs_arr):

    # if already constructed LCS, return
    if (char < 0 or i < 0 or j < 0):
        return lcs_arr
```

```
    # else if looking for first character of LCS...
    elif (i == 0):
        if (lcs_table[i][j] == 1):
            if (seq1[i] == seq2[j]):
                lcs_arr[char] = seq1[i]
                return lcs_arr
            else:
                return _reconstruct_lcs(seq1, seq2, \
                    lcs_table, char, i, j-1, lcs_arr)
        else:
            return lcs_arr
    elif (j == 0):
        if (lcs_table[i][j] == 1):
            if (seq1[i] == seq2[j]):
                lcs_arr[char] = seq1[i]
                return lcs_arr
            else:
                return _reconstruct_lcs(seq1, seq2, \
                    lcs_table, char, i-1, j, lcs_arr)
        else:
            return lcs_arr
    # else consider general case
    else:
        prev, up, left = (lcs_table[i-1][j-1],
                          lcs_table[i-1][j],
                          lcs_table[i][j-1])

        if (seq1[i] == seq2[j]):
            lcs_arr[char] = seq1[i]
            return _reconstruct_lcs(seq1, seq2, \
                    lcs_table, char-1, i-1, j-1, lcs_arr)

        elif (left is not None and up is not None):
            if lcs_table[i-1][j] > lcs_table[i][j-1]:
                return _reconstruct_lcs(seq1, seq2, \
                    lcs_table, char, i-1, j, lcs_arr)
            else:
                return _reconstruct_lcs(seq1, seq2, \
                        lcs_table, char, i, j-1, lcs_arr)
        elif (left is not None):
            return _reconstruct_lcs(seq1, seq2, lcs_table,
                    char, i, j-1, lcs_arr)
        else:
            return _reconstruct_lcs(seq1, seq2, lcs_table,
                    char, i-1, j, lcs_arr)
```

/home/max/classes/16_spring/algorithms/project/pylib/memoized.py

Chapter 9

Appendix 3: Bottom-Up DP Algorithm Implementation

Following is the implementation of the bottom-up dynamic programming algorithm in sec. 4:

```
from profilers import log_recursion
from profilers import time_and_space_profiler
from profilers import registry
from generate_string import strgen
import sys

sys.setrecursionlimit(100000)

@time_and_space_profiler(repeat = 1)
def tabulate_lcs(seq1, seq2, *args):
    """Calls helper function to calculate an LCS.

    Args:
        seq1 (string): a random string sequence
                       generated by
                       generate_string.strgen()
        seq2 (string): another random string
                       sequence like seq1
    Returns:
        LCS table
```

```

    """
    # reset registry
    registry['_tabulate_lcs'] = 0

    len1 = len(seq1)
    len2 = len(seq2)

    # store length of LCS[i,j] in lcs_table
    lcs_table = [[0 for j in range(len2+1)] \
                    for i in range(len1+1)]
    _tabulate_lcs(seq1, seq2, len1+1, len2+1,
                    lcs_table)
    return lcs_table

@log_recursion
def _tabulate_lcs(seq1, seq2, i, j, lcs_table):
    """Iterative bottom-up dynamic programming
    solution to LCS problem. See CLRS p.394.

    Args:
        seq1 (string): a string sequence generated by
                            generate_string.strgen()
        seq2 (string): another random string sequence
                            like seq1
        i (int): number of rows in LCS table
                            (=len(seq1) + 1)
        j (int): number of columns in LCS table
                            (=len(seq2) + 1)
        lcs_table (2D list): a matrix of LCS length
                            for [i-1, j-1] prefix
    Returns:
        None: modifies in place LCS length table
    """

    for row in range(1, i):
        for col in range(1, j):
            if seq1[row-1] == seq2[col-1]:
                lcs_table[row][col] = \
                    lcs_table[row-1][col-1] + 1
            elif lcs_table[row-1][col] \
                    >= lcs_table[row][col-1]:
                lcs_table[row][col] = \
                        lcs_table[row-1][col]
            else:
                lcs_table[row][col] = \
                        lcs_table[row][col-1]
def size_lcs(lcs_table):
    """Returns length of maximum common subsequence.

    Args:
        lcs_table (2D list): a matrix of LCS length
                            for [i, j] prefix
```

```
    Returns:
        length (int): LCS length
    """
    return lcs_table[-1][-1]

@time_and_space_profiler(repeat = 1) #, stream = MEMLOG)
def reconstruct_lcs(seq1, seq2, lcs_table, lcs_length):
    """Calls helper function to reconstruct
    one possible LCS based on saved LCS lengths table.

    Args:
        seq1 (string): a string sequence generated by
                       generate_string.strgen()
        seq2 (string): another random string sequence
                    like seq1
        lcs_length (int): length of LCS
        lcs_table (2D list): a matrix of LCS length
                        for [i, j] prefix
    Returns:
        lcs (string): an LCS
    """
    # reset registry
    registry['_reconstruct_lcs'] = 0

    i = len(seq1)
    if i < 1:
       return ""
    else:
       j = len(seq2)
       lcs_arr = _reconstruct_lcs(seq1, seq2, lcs_table,
              lcs_length-1, i, j, [None] * lcs_length)
       lcs = "".join(lcs_arr)
       return lcs

@log_recursion
def _reconstruct_lcs(seq1, seq2, lcs_table, char, i, j,\
       lcs_arr):

    # if already done with LCS, return
    if (char < 0 or i < 1 or j < 1):
       return lcs_arr
    # else consider general case
    else:
       prev, up, left = (lcs_table[i-1][j-1],
                        lcs_table[i-1][j],
                        lcs_table[i][j-1])

       if (seq1[i-1] == seq2[j-1]):
          lcs_arr[char] = seq1[i-1]
          return _reconstruct_lcs(seq1, seq2, lcs_table,
                 char-1, i-1, j-1, lcs_arr)
       elif (up >= left):
          return _reconstruct_lcs(seq1, seq2, lcs_table,
```

```
                    char, i-1, j, lcs_arr)
        else:
            return _reconstruct_lcs(seq1, seq2, lcs_table,
                    char, i, j-1, lcs_arr)
```

/home/max/classes/16_spring/algorithms/project/pylib/dynamic.py

Chapter 10

Appendix 4: Hirschberg DP Algorithm Implementation

Following is the implementation of the Hirschberg programming algorithm in sec. 5:

```
from profilers import log_recursion
from profilers import time_and_space_profiler
from profilers import registry
from generate_string import strgen
import sys

sys.setrecursionlimit(100000)

@time_and_space_profiler(repeat = 1)
def tabulate_lcs(seq1, seq2, *args):
    """Calls helper function to calculate an LCS.
    ALG B in Hirschberg.

    Args:
        seq1 (string): a random string sequence
            generated by generate_string.strgen()
        seq2 (string): another random string
            sequence like seq1
    Returns:
        LCS table
```

```
    """
    # reset registry
    registry['_tabulate_lcs'] = 0

    len1 = len(seq1)
    len2 = len(seq2)

    # for efficiency (see ALG B)
    # select min(|seq1|, |seq2|) for vector
    # storage
    if len1 < len2:
        seq1, seq2 = seq2, seq1
        len1, len2 = len2, len1

    # if only the length of the LCS is required,
    # the matrix can be reduced to a min(m,n)+1
    # vector as the dynamic programming approach
    # only needs the
    # current and previous columns of the matrix.
    lcs_vector = [0 for j in range(len2+1)]
    _tabulate_lcs(seq1, seq2, len1+1, len2+1,
                  lcs_vector)
    return lcs_vector

@log_recursion
def _tabulate_lcs(seq1, seq2, i, j, lcs_vector):
    """Iterative Hirschberg dynamic programming
    solution to LCS problem. See Hirschberg's ALG B.

    Args:
        seq1 (string): a string sequence generated by
                       generate_string.strgen()
        seq2 (string): another random string sequence
                       like seq1
        i (int): number of rows in LCS table
                       (=len(seq1) + 1)
        j (int): number of columns in LCS table
                       (=len(seq2) + 1)
        lcs_vector (1D list): a vector of LCS length
                    for [i-1, j-1] prefix, as in ALG B
    Returns:
        None: modifies in place LCS length table
    """

    for char in range(1, i):
        prev = 0
        for col in range(1, j):
            if seq1[char-1] == seq2[col-1]:
                tmp = prev
                prev = lcs_vector[col-1] + 1
                lcs_vector[col-1] = tmp
            elif lcs_vector[col] >= prev:
```

```
                lcs_vector[col-1] = prev
                prev = lcs_vector[col]
            else:
                lcs_vector[col-1] = prev
                # prev = prev
            if col == j - 1:
                lcs_vector[col] = prev

@time_and_space_profiler(repeat = 1)
def reconstruct_lcs(seq1, seq2):
    """Calls helper function to construct LCS.
    ALG C in Hirschberg."""
    # reset registry
    registry['_reconstruct_lcs'] = 0
    registry['_tabulate_lcs'] = 0

    m = len(seq1)
    n = len(seq2)

    if (m == 0 or n == 0):
        return ""

    # for efficiency (see ALG B)
    # select min(|seq1|, |seq2|) for vector storage
    if m < n:
        seq1, seq2 = seq2, seq1
        m, n = n, m

    lcs_arr = _reconstruct_lcs(seq1, seq2, m, n)

    lcs = "".join(lcs_arr)
    return lcs

@log_recursion
def _reconstruct_lcs(seq1, seq2, m, n):
    """Implements Algorithm C by Hirschberg.

    Args:
        seq1 (str): sequence 1
        seq2 (str): sequence 2
        m (int): length of seq1
        n (int): length of seq2
    """
    if n == 0:
        return []
    elif m == 1:
        if seq1[0] in seq2:
            return [seq1[0]]
        else:
            return []
    else:
        mid = m // 2

```

```
        lcs_vector_1 = [0 for j in range(n+1)]
        lcs_vector_2 = [0 for j in range(n+1)]

        _tabulate_lcs(seq1[:mid], seq2, \
                mid+1, n+1, lcs_vector_1)
        _tabulate_lcs(seq1[:mid-1:-1], seq2[::-1],
                m-mid+1, n+1, lcs_vector_2)

        sums = [lcs_vector_1[i] + lcs_vector_2[n-i] \
                for i in range(len(lcs_vector_1))]

        k = sums.index(max(sums))

        C1 = _reconstruct_lcs(seq1[:mid], seq2[:k], \
                mid, min(k, n))
        C2 = _reconstruct_lcs(seq1[mid:], seq2[k:], \
                m-mid, n-min(k, n))
        C1.extend(C2)
        return C1

def size_lcs(lcs_vector):
    """Returns length of maximum common subsequence.

    Args:
        lcs_vector (1D list): a vector of LCS
            length for [i, j] prefix
    Returns:
        length (int): LCS length
    """
    return lcs_vector[-1]
```

/home/max/classes/16_spring/algorithms/project/pylib/hirschberg.py

Chapter 11

Appendix 5: Driver Program

Listing for the overall driver program:

```
import os, sys
from datetime import datetime
import importlib
from subprocess import call
from plot import plot_scatter
import csv
from generate_string import strgen

import naive
import memoized
import dynamic
import hirschberg


# increase recursion limit
sys.setrecursionlimit(100000)

# set up directory refs
CURDIR = os.path.abspath(os.path.curdir)
FIGDIR = os.path.join(os.path.dirname(CURDIR),\
           'docs/source/figures')
RESULTS = os.path.join(FIGDIR, 'results.csv')

# alphabets
ALPHAS = {'bin': ['0', '1'],
         'alpha': ['A','C','G','T']}

```

```
# lengths of strings to consider
LENGTHS = {'naive': [5, 10, 15, 20],
           'memoized': [5, 10, 15, 20, 1000, 2000, \
                   3000, 4000, 5000],
           'dynamic': [5, 10, 15, 20, 1000, 2000, \
                   3000, 4000, 5000],
           'hirschberg': [5, 10, 15, 20, 1000, 2000, \
                   3000, 4000, 5000, 10000,\
                   40000]
            }


# key to memory log: line numbers to parse
LOG_LINES = {'memoized': {'size':['43', '51']},
            'dynamic': {'size':['42', '49']},
            'hirschberg': {'lcs':['108', '122']}
            }

MODULES = {
        'naive': naive,
        'memoized': memoized,
        'dynamic': dynamic,
        'hirschberg': hirschberg}

def parse_log(memlog, algorithm, target):
    start = LOG_LINES[algorithm][target][0]
    end = LOG_LINES[algorithm][target][1]
    missing_start = True
    missing_end = True

    for line in memlog.split('\n'):
        toks = line.split()
        if len(toks) > 1 and toks[0] == start and \
            missing_start:
            missing_start = False
            start_val = float(toks[1])
        elif len(toks) > 1 and toks[0] == end and \
                missing_end:
            missing_end = False
            end_val = float(toks[1])

    if (missing_start or missing_end):
        print('tried parsing mem log for: ' + algorithm)
        sys.exit('failed to parse memory log')
    else:
        return (end_val - start_val)

def echo(memo):
    """Prints time stamped debugging message to std out.

    Args:
        memo (str): a message to be printed to screen
    """
```

```
    print("[%s] %s" \
            %(datetime.now().strftime("%m/%d/%y %H:%M:%S"),\
                memo))

def run_experiments():

    # create a library of strings for each alphabet
    # on which algos will be tested:
    # dict(1='z', 3='yzx',...)
    echo("Compiling a library of test strings...")
    test_lengths = \
        set([l for key in LENGTHS.keys() \
                for l in LENGTHS[key]])
    strings_alpha = \
            {l:[strgen(alphabet=ALPHAS['alpha'], size=l),\
                        strgen(alphabet=ALPHAS['alpha'], \
                        size=l)] for l in test_lengths}
    strings_bin = {l:[strgen(alphabet=ALPHAS['bin'], \
                size=l), \
                strgen(alphabet=ALPHAS['bin'], size=l)] \
                for l in test_lengths}

    # list of experimental results (list of dicts)
    experiments = []

    # run tests for each algo for either alphabet
    echo("About to run each algorithm in turn "
            "on each test string...")
    for algorithm in LENGTHS.keys():
        module = MODULES[algorithm]
        echo("Running algorithm module " + \
                module.__name__)

        for str_len in LENGTHS[algorithm]:
            echo("\__ for input string length " + \
                    str(str_len))

            for alphabet in ALPHAS.keys():
                echo(" \__ for alphabet " + \
                        alphabet)

                if alphabet == 'bin':
                    strings = strings_bin
                else:
                    strings = strings_alpha

                # build up a table of LCS lengths
                echo(" |--> calculating LCS length")
                sys.stdout.flush()
                if algorithm != 'naive':
                    algo_size, time_size, memlog_size, \
                        lcs_table = \
                            module.tabulate_lcs(\
```

```
                            strings[str_len][0],
                            strings[str_len][1])
                match = module.size_lcs(lcs_table)
                recursion_depth_size = \
                    module.registry['_tabulate_lcs']
                if algorithm != 'hirschberg':
                    space_size = parse_log(memlog_size,
                                            algorithm,
                                            'size')
                else: # algorithm == 'hirschberg'
                    # negligible for vector
                    space_size = None
            else: # algorithm == 'naive'
                time_size = None
                space_size = None
                recursion_depth_size = None

            # reconstruct actual LCS
            echo(" |--> reconstructing an LCS")
            sys.stdout.flush()
            if algorithm in ('naive', 'hirschberg'):
                algo_lcs, time_lcs, memlog_lcs, \
                        lcs = \
                        module.reconstruct_lcs(\
                            strings[str_len][0],
                            strings[str_len][1])

            else:
                algo_lcs, time_lcs, \
                        memlog_lcs, lcs = \
                            module.reconstruct_lcs(\
                                strings[str_len][0],
                                strings[str_len][1],
                                lcs_table,
                                match)

            recursion_depth_lcs = \
                module.registry['_reconstruct_lcs']

            if algorithm == 'naive':
                # not tracking for short strings
                space_lcs = None
                match = len(lcs)
            elif algorithm in ('memoized', 'dynamic'):
                space_lcs = None
            else:
                space_lcs = parse_log(memlog_lcs,
                                        algorithm,
                                        'lcs')

            echo(" |--> saving results of the run")
            sys.stdout.flush()
            experiments.append({ \
```

```
                    'algo':algorithm,
                    'alphabet':alphabet,
                    'time_sizing':time_size,
                    'time_reconstruct':time_lcs,
                    'space_sizing':space_size,
                    'space_reconstruct':space_lcs,
                    'input_size':str_len,
                    'match_size':match,
                    'recursion_sizing':\
                           recursion_depth_size,
                    'recursion_reconstruct': \
                           recursion_depth_lcs})

    return experiments

def plot_sanity_check(experiments, \
        fname = "sanity_check.ps"):
    """Plot input string length vs LCS length to verify
    all algorithms agree on length of LCS for each input.

    Args:
        experiments (list of dicts): experiment data
        fname (str): filename for plot
    """

    # compile a dict of dicts organized by algo + alphabet
    # {'label': {'x':[...], 'y':[...]},...}
    data = {}
    input_lens = [5, 10, 15, 20, 1000, 2000, \
           3000, 4000, 5000]
    for experiment in experiments:
        label = experiment['algo'] + "_" + \
               experiment['alphabet']
        if experiment['input_size'] in input_lens:
            if label in data.keys():
                data[label]['x'].append(\
                       experiment['input_size'])
                data[label]['y'].append(\
                       experiment['match_size'])
            else:
                data[label] = {}
                data[label]['x'] = \
                       [experiment['input_size']]
                data[label]['y'] = \
                       [experiment['match_size']]
    plot_scatter(data, title = \
           "LCS length vs input str length",
               xlabel = "input string length",
               ylabel = "LCS string length",
               fname = fname)

def plot_all(experiments, attrx, attry,\
        xlabel, ylabel, title, fname):
```

```
    """Plot attrx vs attry for each algorithm
    and alphabet.

    Args:
        experiments (list of dicts): experiment data
        attrx (str): type (size LCS or reconstruct LCS)
        attry (str): type (size LCS or reconstruct LCS)
        title (str): plot title
        fname (str): file name for plot
    """

    # compile a dict of dicts organized by algo + alphabet
    # {'label': {'x':[...], 'y':[...]},...}
    data = {}
    for experiment in experiments:
        label = experiment['algo'] + "_" + \
                experiment['alphabet']
        if experiment[attry]: # if we kept track
            if label in data.keys():
                data[label]['x'].append(experiment[attrx])
                data[label]['y'].append(experiment[attry])
            else:
                data[label] = {}
                data[label]['x'] = [experiment[attrx]]
                data[label]['y'] = [experiment[attry]]
    plot_scatter(data, title = title,
                xlabel = xlabel,
                ylabel = ylabel,
                fname = fname)

def plot_memory_vs_input(experiments, attrx, attry,\
        xlabel, ylabel, title, fname='mem_usage.ps'):
    """Plot memory usage for the most memory expensive
    operation (sizing LCS or reconstructing LCS) vs
    input string length.

    Args:
        experiments (list of dicts): experiment data
        attrx (str): type (size LCS or reconstruct LCS)
        attry (list of str): type (size LCS or reconstruct
                             LCS)
        title (str): plot title
        fname (str): file name for plot
    """
    data = {}
    for experiment in experiments:
        label = experiment['algo'] + "_" + \
                experiment['alphabet']
        # if we kept track
        if experiment[attry[0]] is not None:
            attr = attry[0]
        elif experiment[attry[1]] is not None:
            attr = attry[1]
```

```
        else:
            attr = None

        if attr is not None:
            if label in data.keys():
                data[label]['x'].append(experiment[attrx])
                data[label]['y'].append(experiment[attr])
            else:
                data[label] = {}
                data[label]['x'] = [experiment[attrx]]
                data[label]['y'] = [experiment[attr]]
    plot_scatter(data, title = title,
                xlabel = xlabel,
                ylabel = ylabel,
                fname = fname)

if __name__ == "__main__":

    experiments = run_experiments()

    # write data to file
    header = experiments[0].keys()
    with open(RESULTS, 'w') as csvfile:
        dict_writer = csv.DictWriter(csvfile, \
                fieldnames=header)
        dict_writer.writeheader()
        dict_writer.writerows(experiments)

    # plot data
    echo("Done with algorithm runs. About to plot data...")

    ## (1) as a sanity check, plot LCS length vs.
    ## input str length
    ## for all test strings of length 5 <= len <= 20
    plot_sanity_check(experiments)

    ## (2) plot input string length vs. CPU time for
    ## each algorithm
    plot_all(experiments, attrx = 'input_size',
            attry = 'time_sizing',
            xlabel = "input string length",
            ylabel = "CPU time (sec)",
            title = \
                "Sizing LCS: CPU time vs input str length",
            fname = 'cpu_input_sizing.ps')
    plot_all(experiments, attrx = 'input_size',
            attry = 'time_reconstruct',
            xlabel = "input string length",
            ylabel = "CPU time (sec)",
            title = \
                "Reconstructing LCS: CPU time vs input str length",
            fname = 'cpu_input_reconstruct.ps')

```

```
    ## (3) plot recursion depth vs. CPU time for each algorithm
    plot_all(experiments, attrx = 'recursion_sizing',
           attry = 'time_sizing',
           xlabel = "number of recursive calls",
           ylabel = "CPU time (sec)",
           title = "Sizing LCS: CPU time vs recursion depth",
           fname = 'cpu_recursion_sizing.ps')
    plot_all(experiments, attrx = 'recursion_reconstruct',
           attry = 'time_reconstruct',
           xlabel = "number of recursive calls",
           ylabel = "CPU time (sec)",
           title = \
              "Reconstructing LCS: CPU time vs recursion depth",
           fname = 'cpu_recursion_reconstruct.ps')

    ## (4) plot recursion depth vs input string length for each algo
    plot_all(experiments, attrx = 'input_size',
           attry = 'recursion_sizing',
           xlabel = "input string length",
           ylabel = "number of recursive calls",
           title = \
              "Sizing LCS: recursion depth vs input str length",
           fname = 'recursion_input_sizing.ps')
    plot_all(experiments, attrx = 'input_size',
        attry = 'recursion_reconstruct',
        xlabel = "input string length",
        ylabel = "number of recursive calls",
        title = \
           "Reconstructing LCS: recursion depth vs input str length",
        fname = 'recursion_input_reconstruct.ps')

    ## (5) plot input string lnegth vs memory usage for each algo
    plot_memory_vs_input(experiments, attrx = 'input_size',
           attry = ['space_sizing', 'space_reconstruct'],
           xlabel = "input string length",
           ylabel = "memory usage (MiB)",
           title = "Memory usage vs input string length")

    echo("All done. Exiting...")
```

/home/max/classes/16_spring/algorithms/project/pylib/driver.py

Chapter 12

Appendix 6: Plotter Program

Listing for the plotting routine:

```
import matplotlib.pyplot as plt
import os.path, itertools

CURDIR = os.path.abspath(os.path.curdir)
DOCDIR = os.path.join(os.path.dirname(CURDIR), \
          'docs/source/figures')

def plot_scatter(data, title, xlabel, ylabel, fname):
    """Save 2D scatter plots of data.

    Args:
        data (dict of dicts): dict of x and y series;
                    data['algo_label'] =
                    {'x':[list of x-coords],
                    'y':[list of y-coords]}
        title (string): plot title
        xlabel, ylabel (string): axes' labels
        fname (string): file name to save plot to
    """

    fig = plt.figure()
    axes = plt.gca()

    ax = plt.subplot(111)
    box = ax.get_position()
    ax.set_position([box.x0, box.y0, \
          box.width * 0.7, box.height])
```

```
    fig.suptitle(title, fontsize=20)
    plt.xlabel(xlabel, fontsize=14)
    plt.ylabel(ylabel, fontsize=14)
    labels = ax.get_xticklabels()
    plt.setp(labels, rotation=30, fontsize = 14)

    colors = itertools.cycle(['b', 'g', 'r', \
           'lavender', 'm', 'crimson', 'k', 'plum'])
    markers = itertools.cycle(['+', 'o', 'x', '.', \
           '|', 'v', '_', 's', '*'])

    for algo in data.keys():
        color = next(colors)
        marker = next(markers)

        plt.scatter(data[algo]['x'], data[algo]['y'], \
               s=60, c=color, marker=marker, label=algo)
    plt.grid()
    plt.legend(loc='center left', \
           bbox_to_anchor=(1, 0.5),
                ncol=1, fancybox=True, shadow=True,
                scatterpoints = 1)
    fig.savefig(os.path.join(DOCDIR,fname), \
           bbox_inches = 'tight')
    #plt.show()
```

/home/max/classes/16_spring/algorithms/project/pylib/plot.py

Chapter 13

Appendix 7: String Generator Program

Listing for the string generator routine:

```
from random import choice

def strgen(alphabet=['0', '1'], size=40000):
    """Generates string of characters from
    alphabet of given length."""
    astring = ""
    for i in range(size):
        astring += choice(alphabet)
    return astring

if __name__ == "__main__":
    # functionality test
    for i in range(5):
        some_string = \
                strgen(['A','C','G','T'], 3)
        print("Generated: %s of length %d" \
                %(some_string, len(some_string)))
```

/home/max/classes/16_spring/algorithms/project/pylib/generate_string.py

Chapter 14

Appendix 8: Performance Profiler Program

Listing for runtime, recursion depth and memory profilers:

```
import time, sys
from memory_profiler import LineProfiler, show_results
from collections import defaultdict
import os.path
import io

# keep track of recursive function calls
registry = defaultdict(int)

# keep track of memory usage
CURDIR = os.path.abspath(os.path.curdir)

def log_recursion(func):
    """Decorator that counts the number of function
    invocations.

    Args:
        func: function to be decorated
    Returns:
        decorated func
    Caveats:
        does not account for repeated runs!
    """
```

```python
    # count number of invocations
    def inner(*args, **kwargs):
        """Increments invocations and returns the
        callable unchanged."""

        registry[func.__name__] += 1
        return func(*args, **kwargs)
    return inner


def time_and_space_profiler(repeat = 1):
    """Decorator factory that times the function
    invocation. A function is timed over 'repeat' times
    and then runtime is averaged.

    Args:
        repeat (int): number of repeat runs to average
                             runtime over.
    Returns:
        decorated func (in particular, rutime
            averaged over number of repeat runs)
    """
    def decorate(func):
        """Decorator.

        Args:
            func: function to be decorated
        """
        def inner(*args, **kwargs):
            """Sets timer and returns the elapsed time
            and result of original function.

            Returns:
                func.__name__, elapsed_time,
                    original_return_value (tuple)
            """
            outstream = io.StringIO()
            mem_profiler = LineProfiler()
            start = time.perf_counter()
            for i in range(repeat):
                return_val = \
                    mem_profiler(func)(*args, **kwargs)
            finish = time.perf_counter()
            # log memory usage
            show_results(mem_profiler, \
                    stream=outstream, precision=1)
            # return amortized average cost per run
            elapsed = (finish - start) / repeat
            memlog = outstream.getvalue()
            outstream.close()

            return (func.__name__, elapsed, \
                    memlog, return_val)
```

```python
        return inner
    return decorate
```

/home/max/classes/16_spring/algorithms/project/pylib/profilers.py

Chapter 15

Appendix 9: Tabulated Results

15.1 Legend

1. Algorithms: h = *Hirschberg* | d = *dynamic* | m = *memoized* | n = *naive*
2. alpha(bet): a = *alphabetic (DNA)* | b = *binary*
3. metric: rcnstr = *reconstruction of LCS* | size = *sizing LCS*
4. other: recur = *depth of recursion* | len = *length of string* | space = *memory usage (MiB)*

15.2 Test Set 1

Table 15.1: Test set 1 result summary

algo	input	alpha	match	recur	recur	time	time	space	space
	(len)		(len)	(rcnstr)	(size)	(size)	(rcnstr)	(rcnstr)	(size)
h	5	b	3	9	1	0.4424	0.0145	0	
h	5	a	4	9	1	0.0161	0.016	0	
h	10	b	9	19	1	0.0169	0.0177	0	
h	10	a	7	19	1	0.0162	0.019	0	
h	15	b	11	25	1	0.0196	0.0203	0	
h	15	a	8	27	1	0.0191	0.0221	0	
h	20	b	14	37	1	0.0212	0.0257	0	
h	20	a	13	37	1	0.0209	0.0256	0	
h	1000	b	806	1955	1	5.187	10.071	0	
h	1000	a	645	1865	1	5.1413	9.9765	0	
h	2000	b	1608	3853	1	20.7777	40.0546	0.1	
h	2000	a	1298	3727	1	20.384	40.1049	0	
h	3000	b	2440	5855	1	46.2451	90.5748	0.3	
h	3000	a	1957	5611	1	46.2217	89.3372	0	
h	4000	b	3237	7775	1	82.8076	162.1242	0.4	
h	4000	a	2612	7425	1	82.8802	160.251	0	
h	5000	b	4053	9709	1	131.3543	251.3526	0	
h	5000	a	3265	9239	1	127.6273	248.2184	0	
h	10000	b	8108	19455	1	516.3635	1006.5678	0.8	
h	10000	a	6509	18729	1	516.9911	997.1549	0.1	

algo	input	alpha	match	recur	recur	time	time	space	space
h	40000	b	32447	77833	1	8242.7208	16074.518	2.1	
h	40000	a	26180	74769	1	8164.5099	15977.7473	0.5	
d	5	b	3	8	1	0.0253	0.0147		0
d	5	a	4	6	1	0.0212	0.0144		0
d	10	b	9	11	1	0.0316	0.0144		0
d	10	a	7	13	1	0.0316	0.0137		0
d	15	b	11	19	1	0.0402	0.0142		0
d	15	a	8	19	1	0.0378	0.0147		0
d	20	b	14	26	1	0.0521	0.0148		0
d	20	a	13	27	1	0.0492	0.0155		0
d	1000	b	806	1193	1	59.2117	0.0462		12.3
d	1000	a	645	1356	1	59.824	0.0495		10
d	2000	b	1608	2393	1	236.2218	0.0835		68
d	2000	a	1298	2703	1	238.2887	0.1224		43.9
d	3000	b	2440	3561	1	531.2589	0.1262		159.2
d	3000	a	1957	4042	1	533.7933	0.1331		99.4
d	4000	b	3237	4759	1	945.8537	0.1338		266.1
d	4000	a	2612	5388	1	949.9255	0.1969		156.5
d	5000	b	4053	5935	1	1475.4208	0.2326		419.1
d	5000	a	3265	6727	1	1482.2772	0.3782		244.4
n	5	b	3	58			0.018		
n	5	a	4	39			0.0139		

algo	input	alpha	match	recur	recur	time	time	space	space
n	10	b	9	48			0.0139		
n	10	a	7	4320			0.0524		
n	15	b	11	4707			0.0584		
n	15	a	8	3164454			26.2239		
n	20	b	14	83572			0.7059		
n	20	a	13	230000000			1910.6828		
m	5	b	3	6	27	0.0246	0.0143		0
m	5	a	4	5	24	0.02	0.015		0
m	10	b	9	10	34	0.0294	0.0135		0
m	10	a	7	12	105	0.0272	0.0146		0
m	15	b	11	16	143	0.0382	0.0136		0
m	15	a	8	18	330	0.0376	0.0152		0
m	20	b	14	25	271	0.0513	0.0132		0
m	20	a	13	27	499	0.0504	0.0155		0
m	1000	b	806	1190	577240	62.1108	0.0413		0
m	1000	a	645	1355	1235391	69.2252	0.0393		0
m	2000	b	1608	2392	2501587	251.2945	0.1162		18
m	2000	a	1298	2700	5060499	278.9918	0.0658		15
m	3000	b	2440	3560	5520115	569.9373	0.0806		48.6
m	3000	a	1957	4041	11000000	634.1554	0.0914		33.3
m	4000	b	3237	4758	10000000	1017.0957	0.2871		90.4
m	4000	a	2612	5387	20000000	1138.6572	0.1141		60.4

algo	input	alpha	match	recur	recur	time	time	space	space
m	5000	b	4053	5942	16000000	1606.4914	0.3979		140.2
m	5000	a	3265	6726	31000000	1826.7663	0.4117		98.3

15.3 Test Set 2

Table 15.2: Test set 2 result summary

algo	input	alpha	match	recur	recur	time	time	space	space
	(len)		(len)	(rcnstr)	(size)	(size)	(rcnstr)	(rcnstr)	(size)
d	5	a	2	6	1	0.4485	0.0136		0
d	5	b	4	7	1	0.0218	0.0134		0
d	10	a	4	12	1	0.0285	0.0146		0
d	10	b	8	12	1	0.0303	0.0147		0
d	15	a	8	20	1	0.0398	0.0137		0
d	15	b	10	21	1	0.0369	0.0151		0
d	20	a	10	28	1	0.0464	0.0151		0
d	20	b	16	25	1	0.0514	0.0154		0
d	1000	a	643	1356	1	59.4236	0.0538		12.1
d	1000	b	808	1193	1	58.6192	0.0423		14.2
d	2000	a	1297	2702	1	233.6667	0.1309		45.5
d	2000	b	1611	2390	1	230.7633	0.0826		67.6
d	3000	a	1949	4050	1	524.8506	0.1325		100.6
d	3000	b	2446	3555	1	520.6707	0.0883		157.5

algo	input	alpha	match	recur	recur	time	time	space	space
d	4000	a	2611	5381	1	931.8733	0.2031		157.9
d	4000	b	3225	4772	1	923.8998	0.1830		264.9
d	5000	a	3255	6745	1	1453.7706	0.1652		245
d	5000	b	4061	5939	1	1447.4913	0.4058		418.3
n	5	a	2	227			0.0214		
n	5	b	4	30			0.0132		
n	10	a	4	37412			0.3186		
n	10	b	8	234			0.0144		
n	15	a	8	1207326			9.6630		
n	15	b	10	3151			0.0456		
n	20	a	10	554020891			4433.3023		
n	20	b	16	136317			1.1002		
m	5	a	2	8	41	0.0220	0.0135		0
m	5	b	4	6	27	0.0204	0.0149		0
m	10	a	4	14	136	0.0268	0.0145		0
m	10	b	8	11	61	0.0288	0.0143		0
m	15	a	8	17	270	0.0396	0.0141		0
m	15	b	10	19	167	0.0360	0.0134		0
m	20	a	10	28	620	0.0498	0.0148		0
m	20	b	16	24	371	0.0508	0.0135		0
m	1000	a	643	1355	1257162	67.9600	0.0452		0
m	1000	b	808	1192	603178	61.2487	0.0400		0

algo	input	alpha	match	recur	recur	time	time	space	space
m	2000	a	1297	2702	5017475	272.3344	0.0655		15.1
m	2000	b	1611	2389	2526915	247.3882	0.0619		19.3
m	3000	a	1949	4048	11161409	618.1745	0.0877		35.2
m	3000	b	2446	3554	5478036	558.2729	0.0837		46.4
m	4000	a	2611	5385	19821104	1110.3767	0.2876		61.9
m	4000	b	3225	4772	10239993	1012.3914	0.2972		90.3
m	5000	a	3255	6744	30960754	1768.7142	0.4066		97.8
m	5000	b	4061	5937	15880614	1591.7180	0.3978		145.5
h	5	a	2	9	1	0.0213	0.0139	0	
h	5	b	4	9	1	0.0163	0.0162	0	
h	10	a	4	13	1	0.0176	0.0191	0	
h	10	b	8	19	1	0.0166	0.0188	0	
h	15	a	8	23	1	0.0175	0.0216	0	
h	15	b	10	27	1	0.0153	0.0212	0	
h	20	a	10	37	1	0.0197	0.0261	0	
h	20	b	16	37	1	0.0179	0.0262	0	
h	1000	a	643	1877	1	5.0141	9.6747	0	
h	1000	b	808	1951	1	4.9947	9.8500	0	
h	2000	a	1297	3721	1	19.8064	38.6532	0	
h	2000	b	1611	3881	1	20.2065	39.1011	0	
h	3000	a	1949	5631	1	44.6806	86.9596	0	
h	3000	b	2446	5869	1	45.2797	87.8965	0	

algo	input	alpha	match	recur	recur	time	time	space	space
h	4000	a	2611	7471	1	79.1319	154.6820	0	
h	4000	b	3225	7741	1	81.3705	158.7095	0	
h	5000	a	3255	9279	1	123.8846	241.5663	0	
h	5000	b	4061	9731	1	127.3422	244.8492	0	

References

Cormen & al., 2009. *Introduction to Algorithms*, Cambridge, Mass.: The MIT Press.

Hirschberg, D.S., 1975. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6), pp.341–343.