Experimental Project:

Common Subsequence Algorithms

Maksim Yegorov

CSCI-665 @ RIT, Winter 2016

Prof. Stanisław Radziszowski

May 9, 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

I implement and investigate the performance of four algorithms that each calculate the length of and reconstruct a longest common subsequences shared by two 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 are the comparatively long execution times, as will become clear from the data below. 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 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
   ...
```

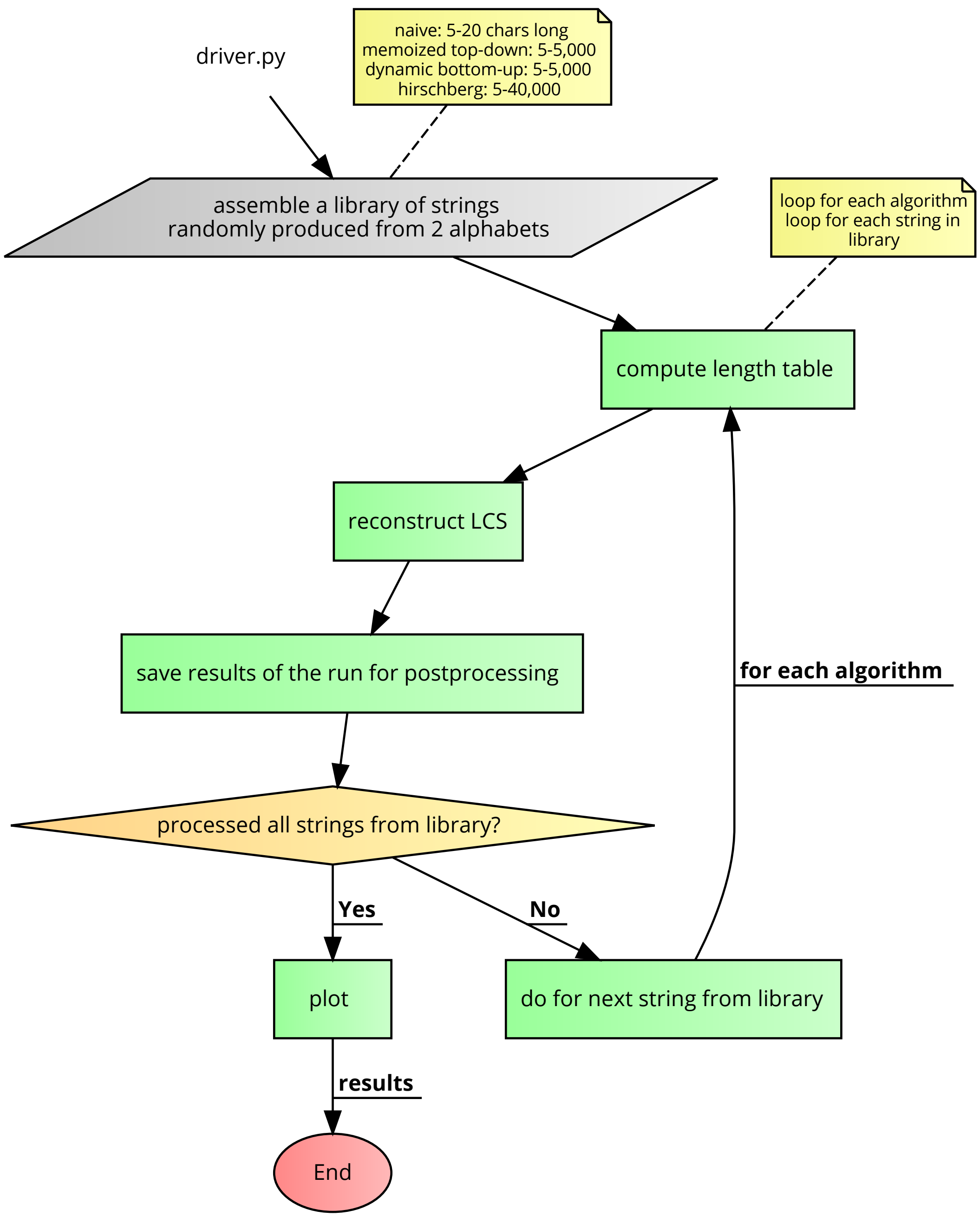
Prior to running the experiments, I set artificially high system limits on my stack size so as to prevent the program from failing prematurely from a too deep recursion and force any bottleneck in CPU or memory capacity instead:

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

and from `Python`:

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

The flowchart below shows the overall logic of the driver script (`driver.py`):



Two batches of experiments were run in total. The input strings were chosen from two alphabets: a binary alphabet $\{0, 1\}$ and a four-item alphabet representing a 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 are run wiht input strings from the same library randomly assembled for a given string length 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 custom code. 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 computing the length of an LCS, and for reconstructing an LCS – except for the *naive* algorithm, where the tasks cannot be separated.

Profiling the algorithms for time and memory consumption is done by wrapping the 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 of the above in a decorator that increments the recursion depth on each invocation. All of the profiling functions are defined in the `profilers.py` module.

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*:

```
    # 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)
    waste, waste, memlog, lcs = \
           reconstruct_lcs("bbcaba", "cbbbaab",
                  lcs_table, lcs_length)
    assert lcs == "bbba"
```

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

Also, for the library of strings against which all algorithms were tested, I plot the length of LCS below. 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):

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

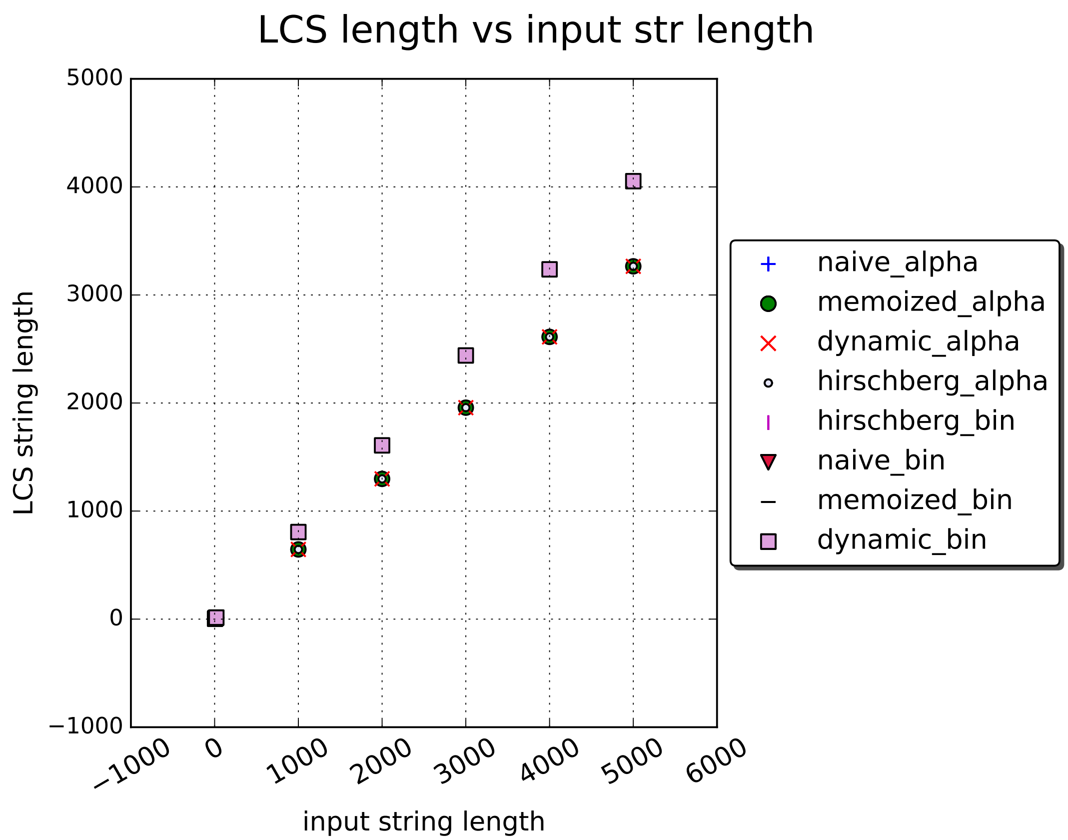


Figure 1.1: Sanity check: Verify all algorithms compute the same LCS length for a given pair of input strings

Chapter 2

Naive Algorithm

The naive recursive solution is based on recursion (15.9) in (Cormen & al. 2009) repeated here for clarity.

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

For the `Python` implementation, see listing in sec. 7.

For a feasible run time, only strings of length up to 20 were used. Asymptotic complexity suggests that the recursive algorithm would take exponential time to compute the length of (and coincidentally reconstruct) of an LCS of two strings. This is indeed supported 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 exponential algorithms.

Figure **??** shows a recursion depth profiler run using the naive algorithm and character alphabet for strings of varying length and structure. For a feasible run time, only strings of length 10 and 15 were used.

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.

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 $O(m+n)$ and quadratic space $O(mn)$ for reconstructing an LCS, given the table computed beforehand.

It will be seen in sec. 6 that the *memoized algorithm* has indeed polynomial execution time and memory performance for sizing an LCS, and linear time for reconstructing an LCS.

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

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

It will be seen 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 differs, in that 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.

Figure **??** shows a run time profiler output for a string of size 10,000.

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. where the former two algorithms excel in the time requirements for recovery, *Hirschberg* excels in the space requirements for recovery.

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

Chapter 6

Summary of results

In this section, I compare experimental runs side by side. The following plot excludes *naive algorithm* results so as to distinguish sizing LCS from reconstruction. It can be seen that the execution time of the remaining three algorithms is polynomial 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.

Figure **??** shows a run time profiler output for a string of size 10,000.

The following plot 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 polynomial nature of

Hirschberg's reconstruction scheme (for the CPU time, as opposed to memory usage).

For the recursive *memoized algorithm* (*naive* not shown, as it performs reconstruction coupled with sizing the LCS), one can see the polynomial 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, the following plot shows the polynomial 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.

I performed a second run, with essentially identical results. For better resolution, the following plots exclude the runs for inputs of size above 5,000 (*Hirschberg algorithm*).

All graphs are clearly polynomial in the length of input. Interestingly, alphabetic input matching is less efficient than binary. *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 x n-sized lists from Python's Standard Library instead of using arrays from the outside `numpy` library that are much more compact and efficient.

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

For recursive algorithms, the polynomial relationship between recursion vs input length mirrors that between CPU time vs input length. The two are clearly related.

This observation is reinforced by the following plot that shows CPU runtime vs recursion depth. For the recursive *memoized* algorithm, the relationship is clearly linear. What is interesting to note is 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.

Finally, the memory usage is also polynomial 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.

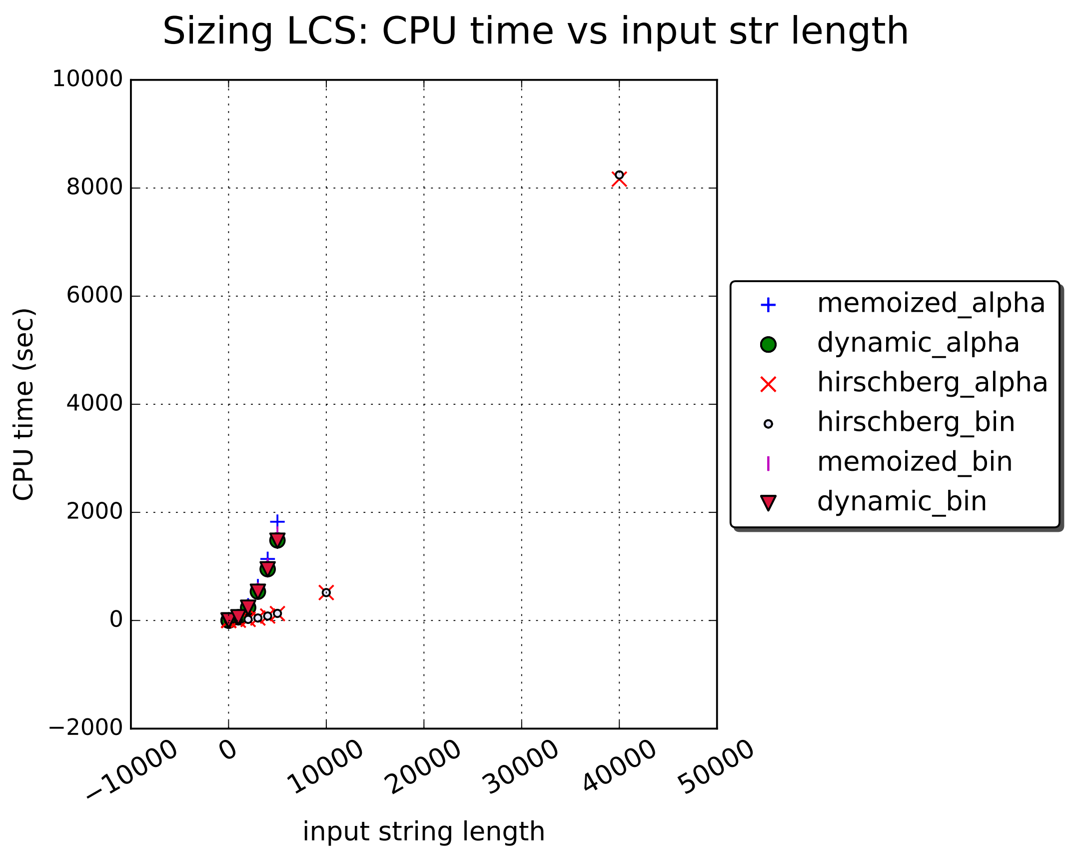


Figure 6.1: Runtime vs input length: sizing LCS

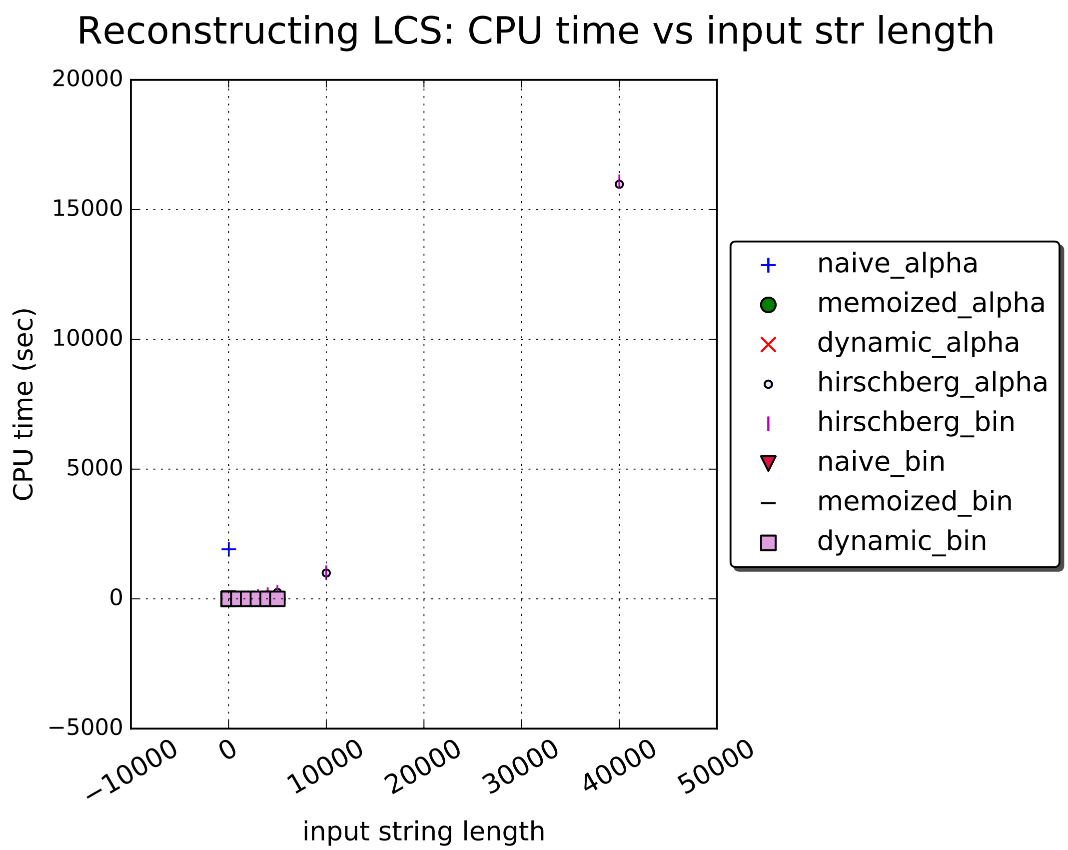


Figure 6.2: Runtime vs input length: reconstructing LCS

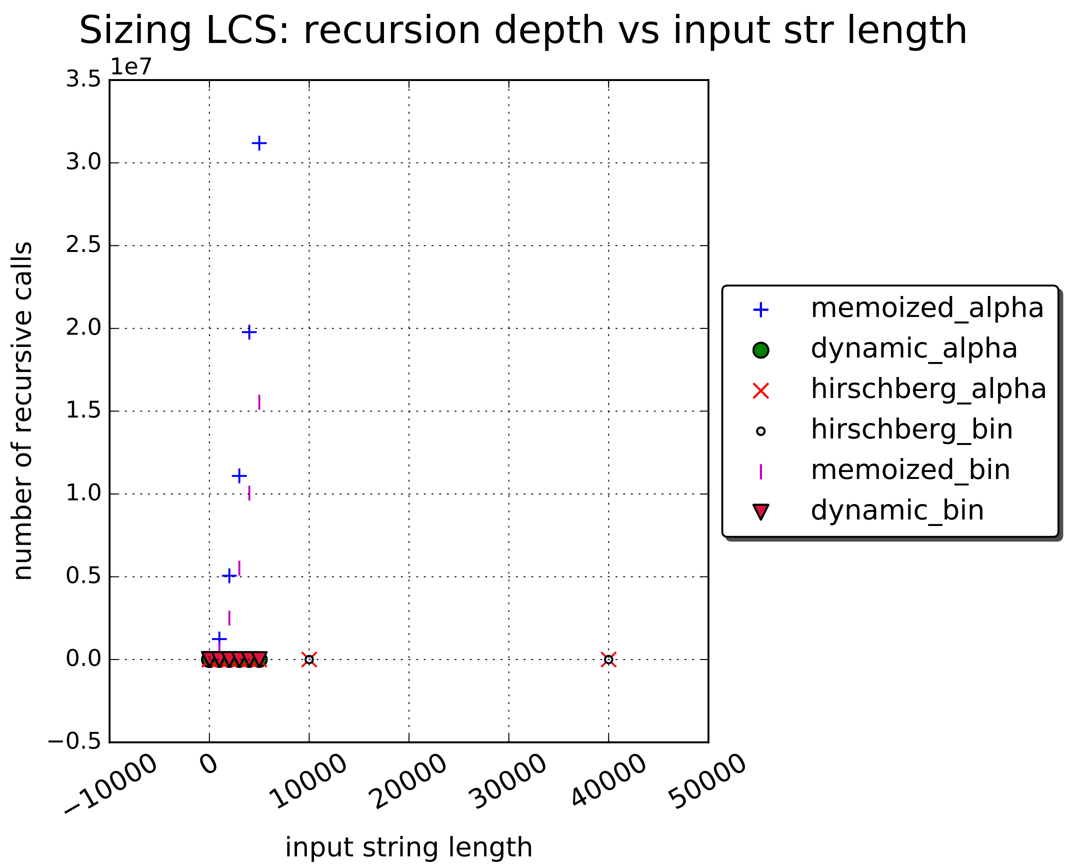


Figure 6.3: Recursion depth vs input length: sizing LCS

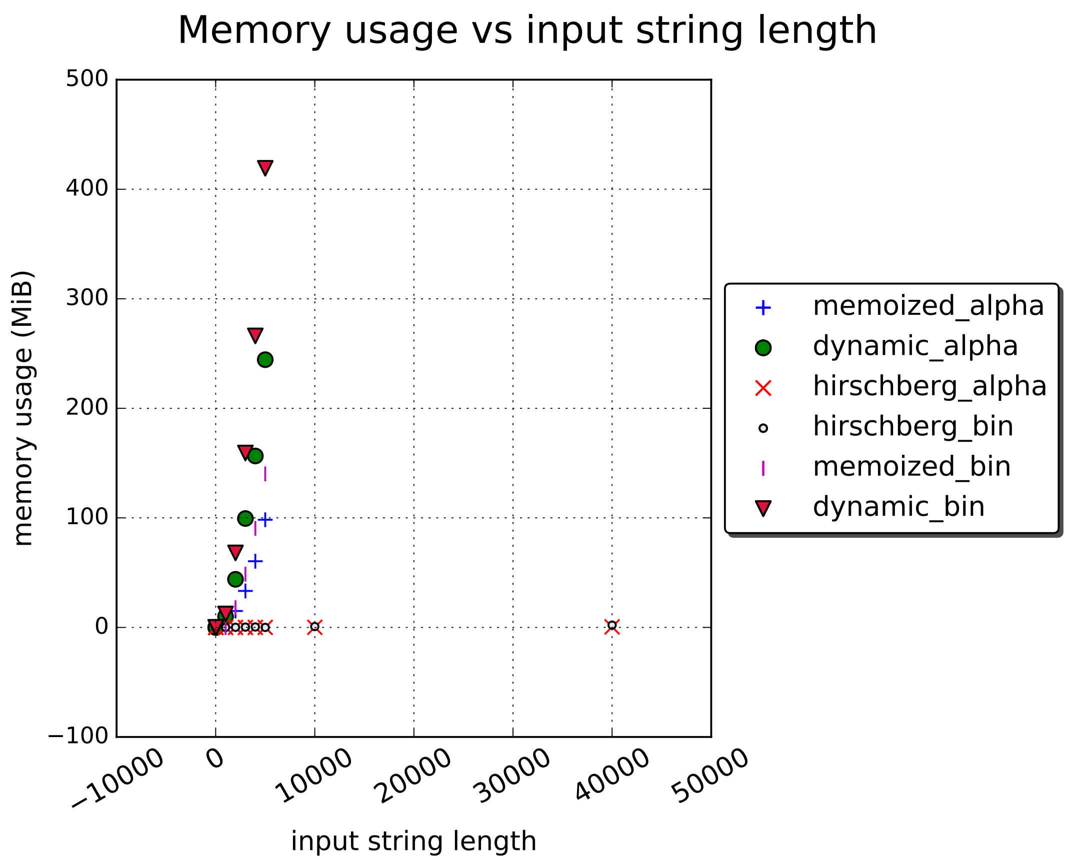


Figure 6.4: Memory usage

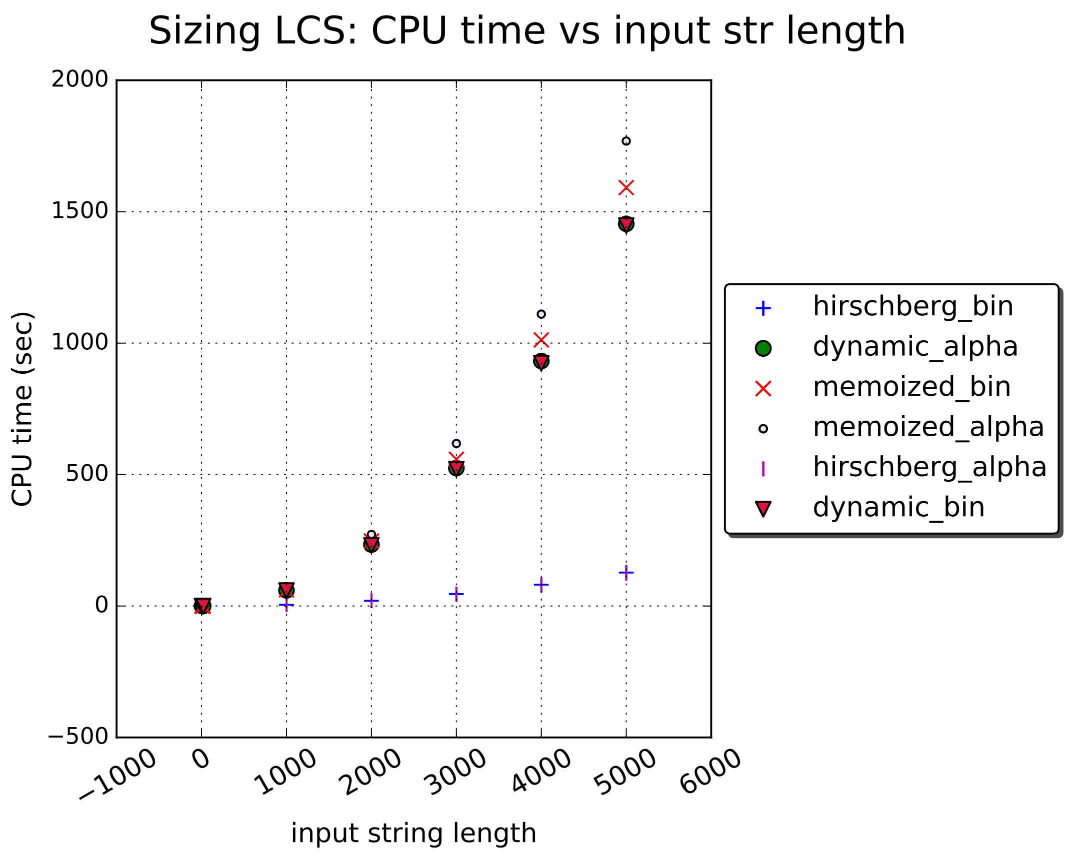


Figure 6.5: Runtime vs input length: sizing LCS

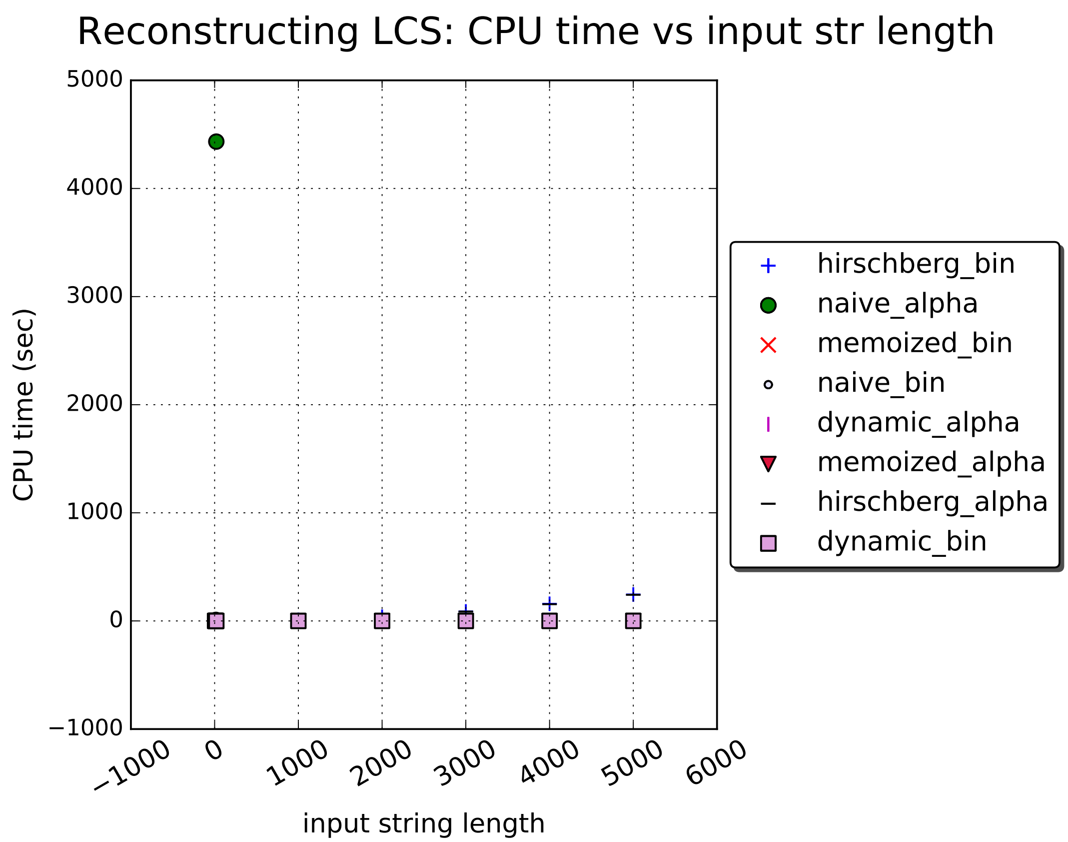


Figure 6.6: Runtime vs input length: reconstructing LCS

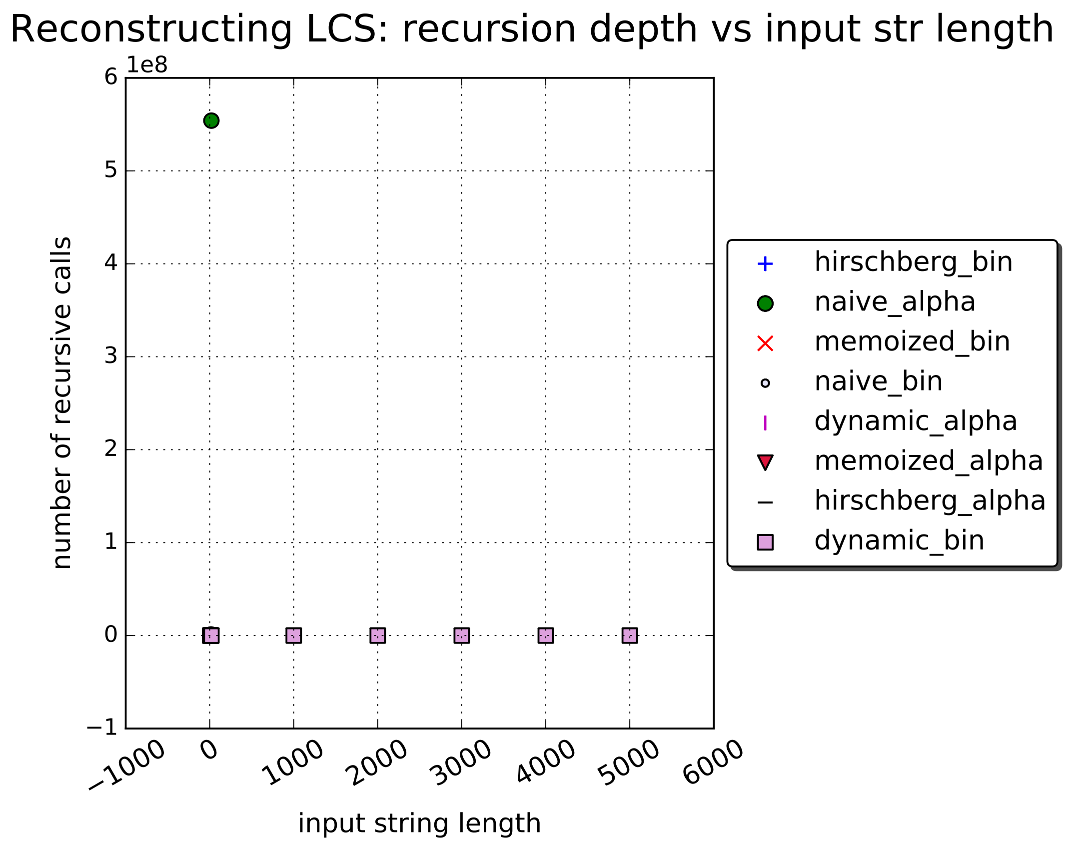


Figure 6.7: Recursion depth vs input length: reconstructing LCS

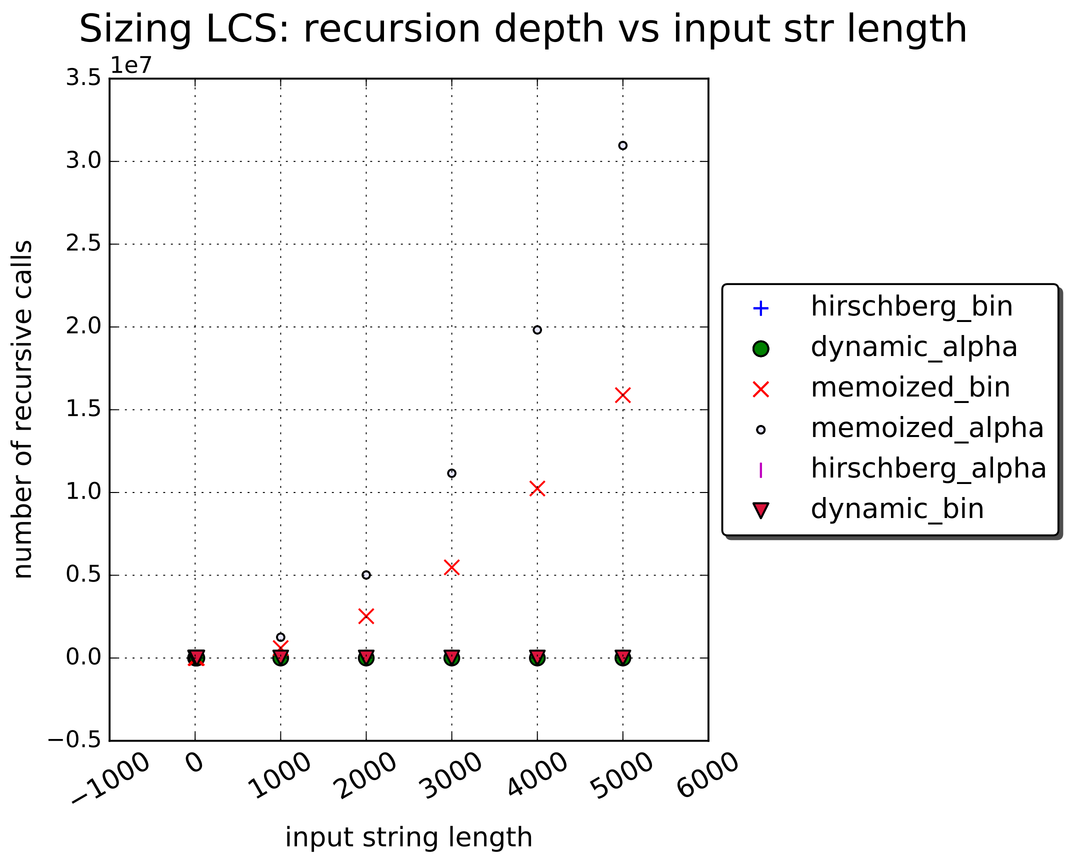


Figure 6.8: Recursion depth vs input length: sizing LCS

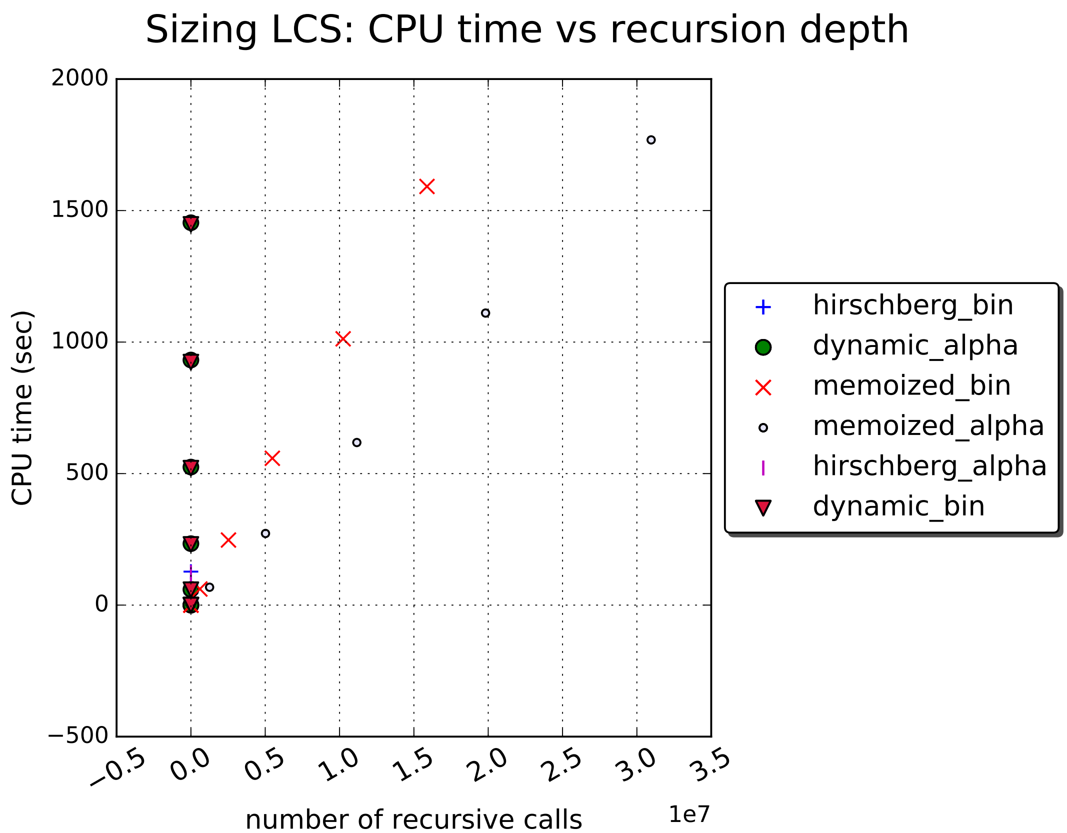


Figure 6.9: Runtime vs recursion depth: sizing LCS

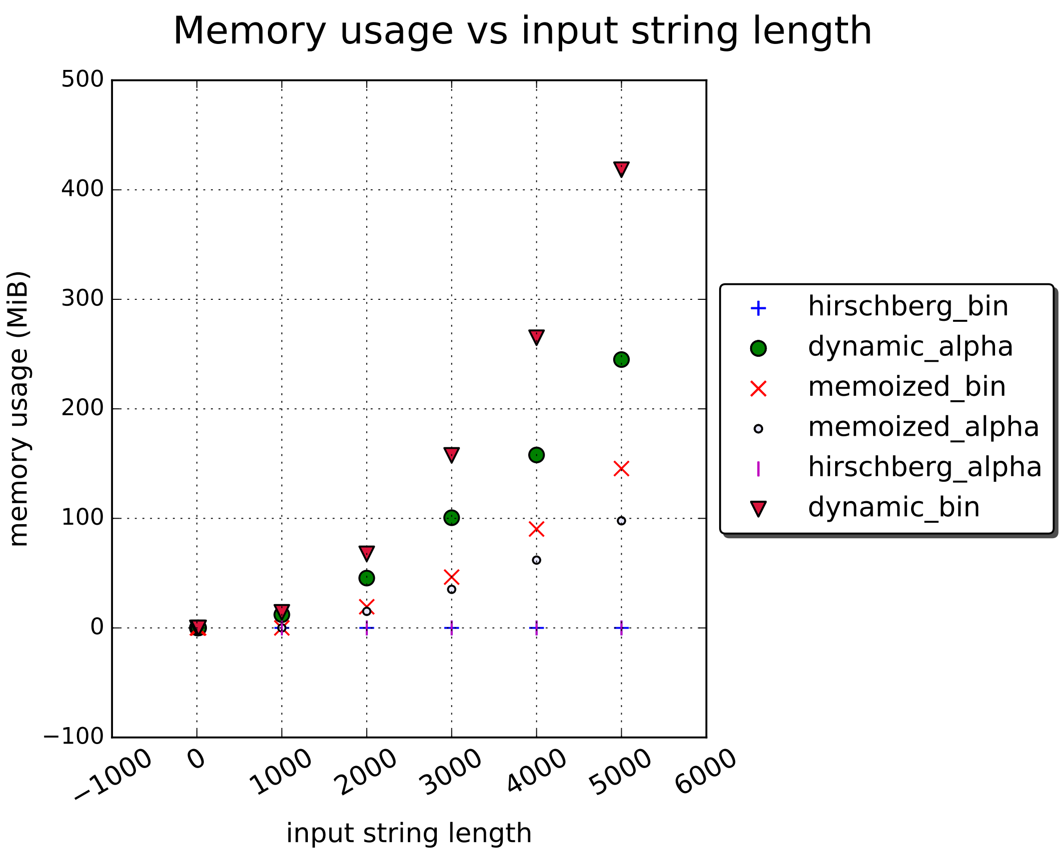


Figure 6.10: Memory usage

Chapter 7

Appendix 1: Naive Algorithm Implementation

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

```
from profilers import log_recursion, 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, "")

@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:
```

/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:

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 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
```

```python

    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] = \
```

Chapter 10

Appendix 4: Hirschberg DP Algorithm Implementation

TODO: this is a stub for a future section.

Chapter 11

Appendix 5: Driver Program

Listing for the overall driver program:

```
"""

__author__ = "Maksim Yegorov"
__date__ = "2016-05-07 Sat 04:26 PM"

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':['41', '49']},
            'dynamic': {'size':['39', '46']},
            'hirschberg': {'lcs':['102', '116']}
            }

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'
                space_size = None # negligible for vector
        else: # algorithm == 'naive'
            time_size = None
            space_size = None
            recursion_depth_size = None

        # reconstruct actual LCS
        echo(" |--> reconstructing an LCS")
```

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

Chapter 12

Appendix 6: Plotter Program

Listing for the plotting routine:

```
#!/usr/bin/env python3

"""
plot.py

Plot experimental results.

Usage (meant to be run from a build script):
    python3 .py
"""
__author__ = "Maksim Yegorov"
__date__ = "2016-05-06 Fri 01:30 AM"


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()
```

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

Chapter 13

Appendix 7: String Generator Program

Listing for the string generator routine:

```
#!/usr/bin/env python3
"""
generate_string.py

Generate a string given alphabet and length of string.

Usage:
    python3 generate_string.py
"""

__author__ = "Maksim Yegorov"
__date__ = "2016-04-06 Wed 08:06 PM"

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
```

/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:

```
__author__ = "Maksim Yegorov"
__date__ = "2016-04-28 Thu 02:38 PM"

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!
    """
    # 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()
```

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. Available at: http://doi.acm.org/10.1145/360825.360861.