Experimental Project: Common Subsequence Algorithms

Maksim Yegorov

CSCI-665 @ RIT, Winter 2016

Prof. Stanisław Radziszowski

2016-04-07 Thu 02:50 PM

Abstract

Four algorithms that establish the length of a longest common subsequence (LCS) of two arbitrary strings are implemented and compared side by side. The preliminary submittal contains the implementations of the naive, memoized and bottom-up dynamic algorithms to calculate the length of the LCS, and a sample run of the main driver program that measures and plots the run time of select implementations. Not yet implemented are the Hirschberg algorithm, as well as the portions of each algorithm (except naive) dealing with LCS reconstruction.

Table of Contents

••

List of Figures

..

List of Tables

Chapter 1

Introduction

The implementation of all algorithms except Hirschberg's quadratic-time linear-space algorithm is based on (Cormen & al. 2009).

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. 6.

Figure 2.1 shows a run time profiler results using the naive algorithm for strings of varying length and structure. For a feasible run time, only strings of length 10 and 15 were used. Both binary and character alphabets are compared.

Figure 2.2 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.

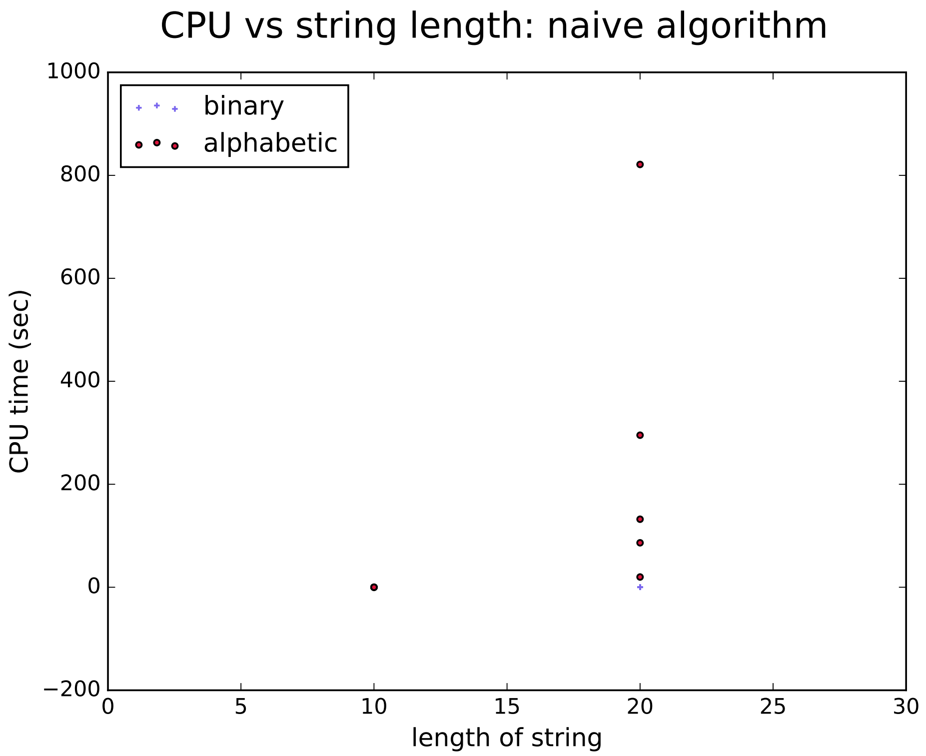

Figure 2.1: CPU time vs input string length: naive algorithm.

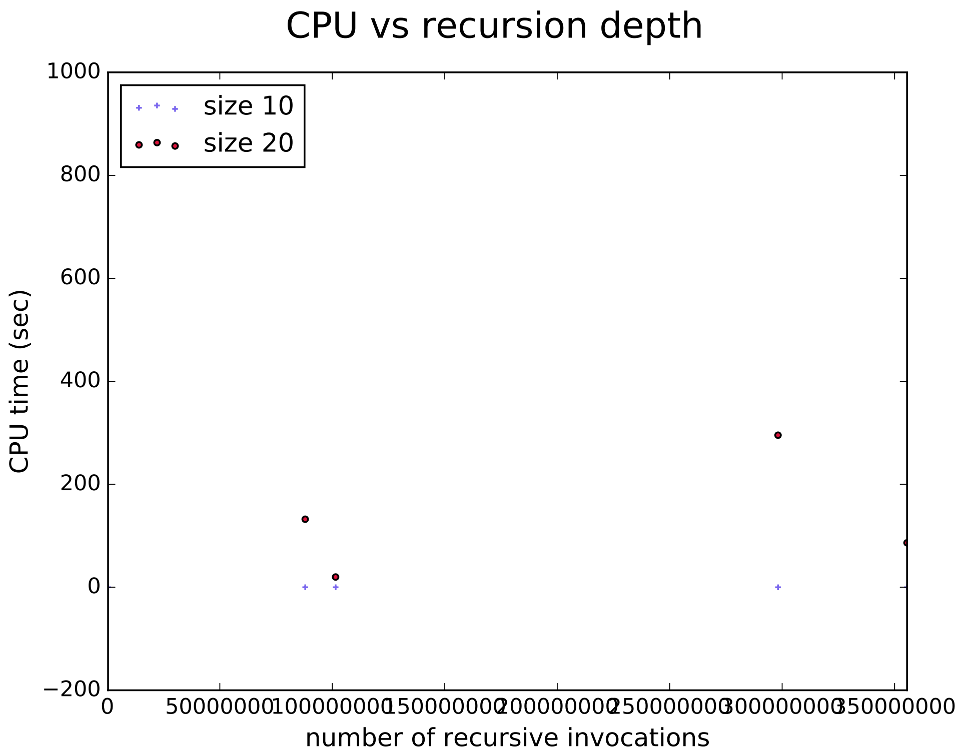

Figure 2.2: CPU time vs recursion depth: naive algorithm.

Chapter 3

Top-Down Memoized Algorithm

The memoized implementation uses top-down recursion essentially identical to the naive approach in sec. 6, except that performed computations are saved in a table. *TODO*: add pseudo-code.

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

Chapter 4

Bottom-Up Dynamic Programming Algorithm

The DP implementation uses bottom-up iterative approach in `Fig. 15.8` in (Cormen & al. 2009). *TODO*: add pseudo-code.

For the `Python` implementation, see listing in sec. 8.

Figure 4.1 shows a run time profiler output comparing the bottom-up dynamic vs top-down memoization algorithm performance for the same string input.

Figure 4.2 shows a run time profiler output for a string of size 10,000.

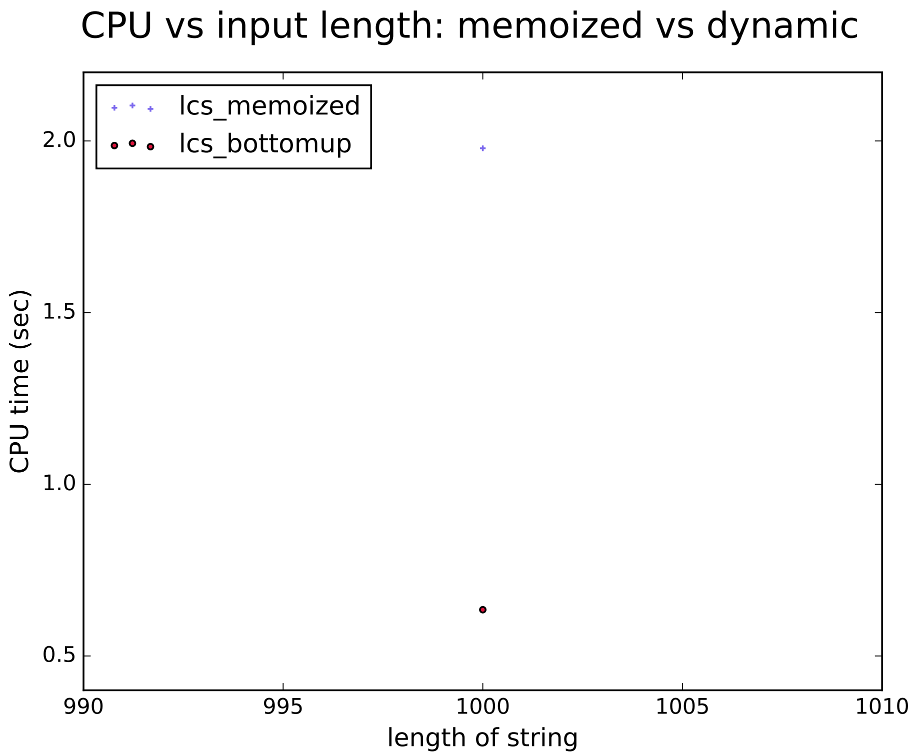

Figure 4.1: CPU time vs input string length: memoization vs dynamic algorithms.

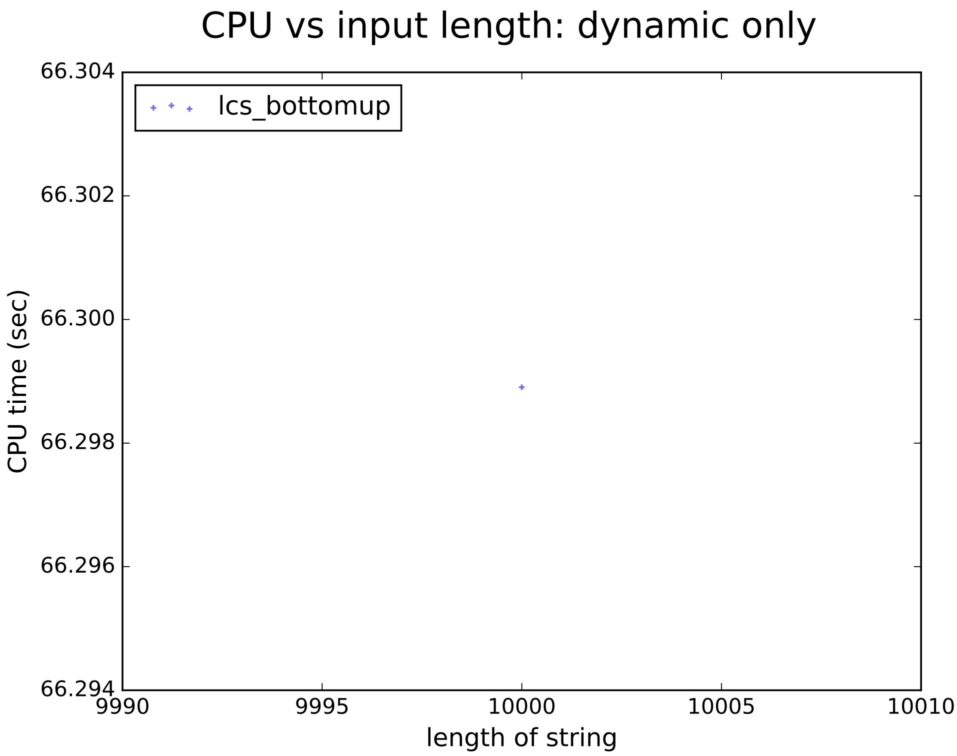

Figure 4.2: CPU time for a long input string: dynamic algorithm.

Chapter 5

Hirschberg Linear Space Dynamic Programming Algorithm

TODO: This is a stub for a future section.

For the `Python` implementation, see listing in sec. 9.

Chapter 6

Appendix 1: Naive Algorithm Implementation

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

```python
from profilers import len_recursion, time_profiler, registry
from generate_string import strgen


@time_profiler(repeat = 1)
def lcs_naive(seq1, seq2):
    """Calls helper function to calculate an LCS."""

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


@len_recursion
def _lcs_naive(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 _lcs_naive(seq1, seq2, i-1, j-1, \
                    seq1[i] + lcs)
        else:
            return max(_lcs_naive(seq1, seq2, i-1, j, \
                    lcs),
                _lcs_naive(seq1, seq2, i, j-1, lcs),
                key=len)
```

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

Chapter 7

Appendix 2: Memoized Algorithm Implementation

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

```
from profilers import len_recursion, time_profiler, registry
from generate_string import strgen


@time_profiler(repeat=1)
def lcs_memoized(seq1, seq2):
    """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 length (int)

    """
    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)]
    _lcs_memoized(seq1, seq2, len1-1, len2-1, \
                  lcs_table)
    return lcs_table[len1-1][len2-1]
```

```
@len_recursion
def _lcs_memoized(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 + \
                    _lcs_memoized(seq1, seq2, i-1, \
                                  j-1, lcs_table)
            else:
                val = max(_lcs_memoized(seq1, seq2, \
                                  i-1, j, lcs_table),
                        _lcs_memoized(seq1, seq2, i, \
                                  j-1, lcs_table))

            lcs_table[i][j] = val
            return val
```
/home/max/classes/16_spring/algorithms/project/pylib/memoized.py

Chapter 8

Appendix 3: Bottom-Up DP Algorithm Implementation

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

```
from profilers import len_recursion, time_profiler, registry
from generate_string import strgen

@time_profiler(repeat=1)
def lcs_bottomup(seq1, seq2):
    """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 length (int)

    """
    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)]
    _lcs_bottomup(seq1, seq2, len1+1, len2+1,
                    lcs_table)
    return lcs_table[len1][len2]

```

```python

# @to_profile
def _lcs_bottomup(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]
```

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

Chapter 9

Appendix 4: Hirschberg DP Algorithm Implementation

TODO: this is a stub for a future section.

Chapter 10

Appendix 5: Driver Program

Listing for the overall driver program:

```
import os.path, sys
import importlib
from plot import plot_scatter
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')
LOGDIR = os.path.join(CURDIR, 'logs')

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


if __name__ == "__main__":

    ## (1) for each algorithm
    # plot CPU time vs string length for several
    # inputs at each length
    # set legend by length of LCS or alphabet
    str_lens = [10, 20]
```

```
    title = 'CPU vs string length: naive algorithm'
    series = []

    set1 = {'x':[], 'y':[]}
    seq1_bin = [strgen(alphabet=ALPHAS['bin'], \
               size=strlen) for i in range(5) \
               for strlen in str_lens]
    seq2_bin = [strgen(alphabet=ALPHAS['bin'], \
               size=strlen) for i in range(5) \
               for strlen in str_lens]
    for (str1, str2) in zip(seq1_bin, seq2_bin):
        algo_name, time_elapsed, lcs = \
                naive.lcs_naive(str1, str2)
        set1['x'].append(len(str1))
        set1['y'].append(time_elapsed)
    series.append('binary')

    #store CPU vs length
    set2 = {'x':[], 'y':[]}
    set3 = {'x':[], 'y':[]}
    set4 = {'x':[], 'y':[]}
    seq1_alpha = [strgen(alphabet=ALPHAS['alpha'], \
                 size=strlen) for i in range(5) \
                 for strlen in str_lens]
    seq2_alpha = [strgen(alphabet=ALPHAS['alpha'], \
                 size=strlen) for i in range(5) \
                 for strlen in str_lens]

    for (str1, str2) in zip(seq1_alpha, seq2_alpha):
        strlen = len(str1)
        algo_name, time_elapsed, lcs = \
                naive.lcs_naive(str1, str2)
        set2['x'].append(len(str1))
        set2['y'].append(time_elapsed)
        if strlen == 10:
            set3['x'].append(\
                    naive.registry['_lcs_naive'])
            set3['y'].append(time_elapsed)
        elif strlen == 20:
            set4['x'].append(\
                    naive.registry['_lcs_naive'])
            set4['y'].append(time_elapsed)
    series.append('alphabetic')
    plot_scatter(set1, set2, series, title, \
            xlabel = 'length of string', \
            ylabel = 'CPU time (sec)')

    print("-> done with naive algorithm runs")

    # (2) plot CPU time vs number of recursive
    # invocations for several lengths
    plot_scatter(set3, set4, ['size 10', 'size 20'], \
            title = 'CPU vs recursion depth', \
            xlabel = 'number of recursive invocations',\
            ylabel = 'CPU time (sec)')

    print("-> done with CPU vs recursion depth plots")
```

```
    # (3) TODO: plot memory use vs string length for
    # several inputs at each length


    ## (4) algorithm comparison
    # for given string, plot CPU time for each
    # algorithm; do this for several strings
    # (vary length and alphabet)
    title = 'CPU vs input length: memoized vs dynamic'
    str1 = strgen(alphabet=ALPHAS['alpha'], size=1000)
    str2 = strgen(alphabet=ALPHAS['alpha'], size=1000)

    set1 = {'x':[], 'y':[]} #store CPU vs length: memoized
    labels = []
    algo_name, time_elapsed, lcs_len = \
           memoized.lcs_memoized(str1, str2)
    set1['x'].append(len(str1))
    set1['y'].append(time_elapsed)
    labels.append(algo_name)

    #store CPU vs length: bottom-up dynamic
    set2 = {'x':[], 'y':[]}
    algo_name, time_elapsed, lcs_len = \
           dynamic.lcs_bottomup(str1, str2)
    set2['x'].append(len(str1))
    set2['y'].append(time_elapsed)
    labels.append(algo_name)
    plot_scatter(set1, set2, labels, title, \
              xlabel = 'length of string', \
              ylabel = 'CPU time (sec)')


    print("-> done with CPU vs length plots")

    ## (5) show time for strlen == 40000
    title = 'CPU vs input length: dynamic only'
    str1 = strgen(alphabet=ALPHAS['alpha'], \
           size=10000)
    str2 = strgen(alphabet=ALPHAS['alpha'], \
           size=10000)

    set1 = {'x':[], 'y':[]} #store CPU vs length: dynamic
    labels = []
    algo_name, time_elapsed, lcs_len = \
           dynamic.lcs_bottomup(str1, str2)
    set1['x'].append(len(str1))
    set1['y'].append(time_elapsed)
    labels.append(algo_name)

    plot_scatter(set1, [], labels, title, \
              xlabel = 'length of string', \
              ylabel = 'CPU time (sec)')


    print("-> done with plot for dynamic @ 10K long string")
```

Chapter 11

Appendix 6: Plotter Program

Listing for the plotting routine:

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

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

def plot_scatter(set1, set2, labels, title, xlabel, ylabel):
    """Save 2D scatter plot of numplots sets of data.

    Args:
        set1 (dict): dict of coordinate lists; set1 =
                    {'x':[list of x-coords],
                     'y':[list of y-coords]}
        set2: ditto
        labels (list of strings): series labels
                                for each data set
        title (string): plot title
        xlabel, ylabel (string): axes labels
    """

    fig = plt.figure()
    axes = plt.gca()
    axes.set_xlim([set1['x'][0] - 5, \
           set1['x'][-1] + 5])
    plt.ticklabel_format(style='plain', axis='both', \
           useOffset = False)
    fig.suptitle(title, fontsize=20)
    plt.xlabel(xlabel, fontsize=14)
    plt.ylabel(ylabel, fontsize=14)
    if set2:
        plt.scatter(set1['x'], set1['y'], s=20,
```

```
                c='mediumslateblue',
                marker='+', label = labels[0])
        plt.scatter(set2['x'], set2['y'], s=20, c='crimson',
                marker='o', label = labels[1])
    else:
        plt.scatter(set1['x'], set1['y'], s=20,
                c='mediumslateblue',
                marker='+', label = labels[0])

    plt.legend(loc='upper left')
    fname = '_'.join(title.replace(':','_').split()) + '.ps'
    fig.savefig(os.path.join(DOCDIR,fname))
    #plt.show()
```

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

Chapter 12

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

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 13

Appendix 8: Performance Profiler Program

Listing for runtime, recursion depth and memory profilers:

```python
import time, sys
from memory_profiler import profile as mem_profiler
from collections import defaultdict

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

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

    Args:
        func: decorated function
    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_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
    """
    def decorate(func):
        """Decorator.

        Args:
            func: decorated function
        """
        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)
            """

            start = time.perf_counter()
            for i in range(repeat):
                return_val = func(*args, **kwargs)
            finish = time.perf_counter()
            elapsed = finish - start

            return (func.__name__, elapsed, return_val)
        return inner
    return decorate
```

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

References

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