Experimental Project:

Common Subsequence Algorithms

Maksim Yegorov

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.

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

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

6

and character alphabet for strings of varying length and structure. For a feasible run time, only strings of length 10 and 15 were used.

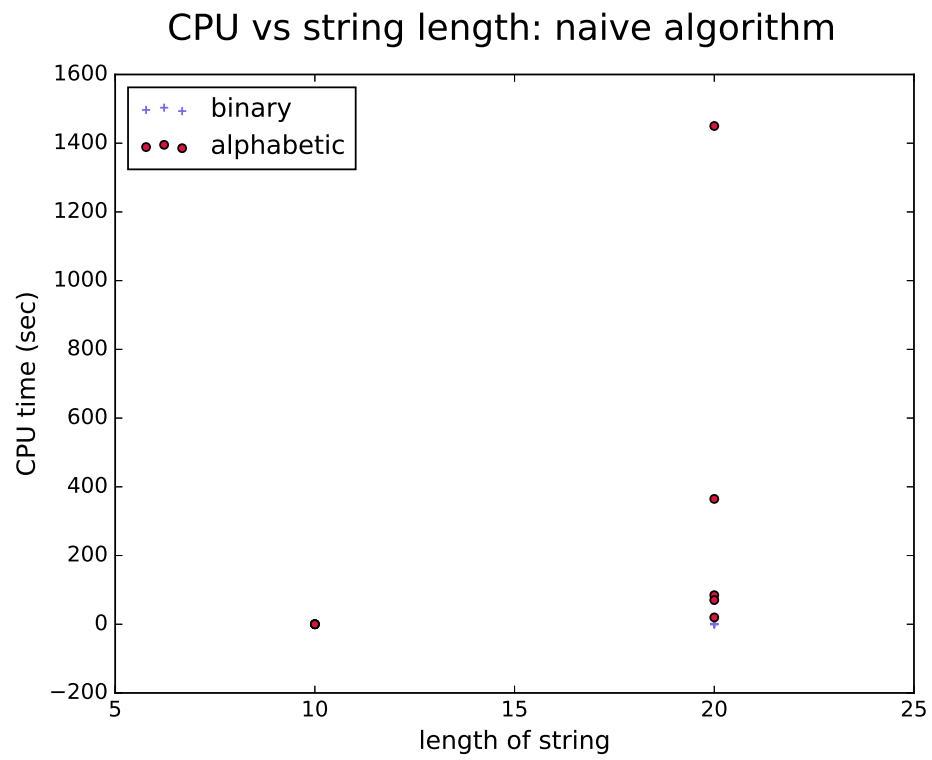Figure 2.1: CPU time vs input string length: naive algorithm.
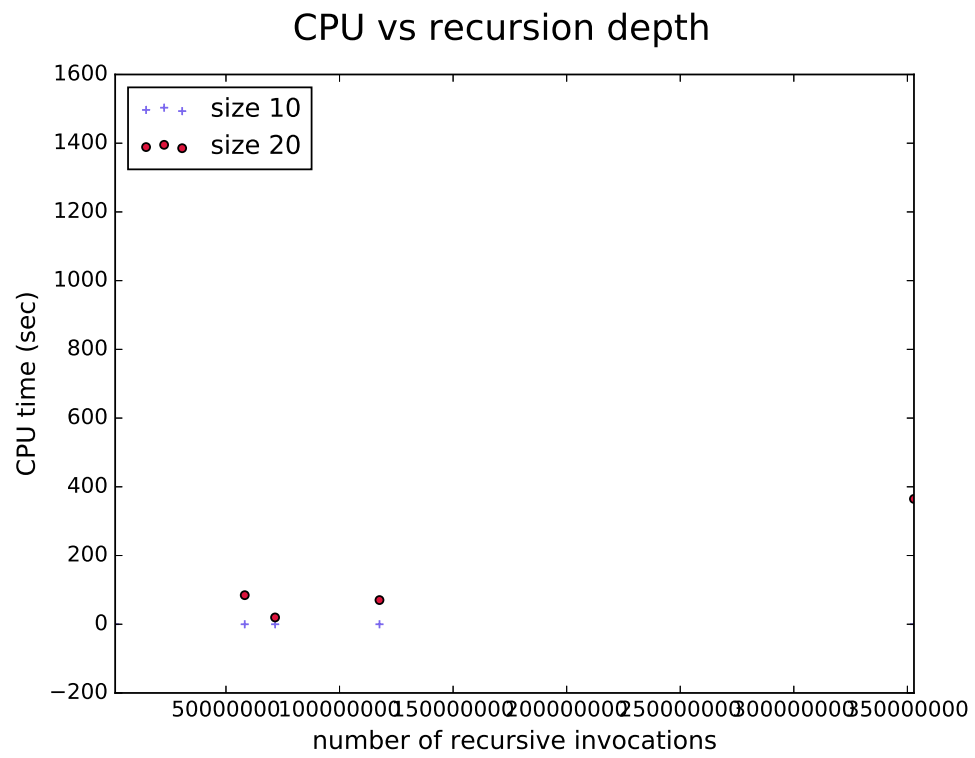
Figure 2.2: CPU time vs recursion depth: naive algorithm.

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.

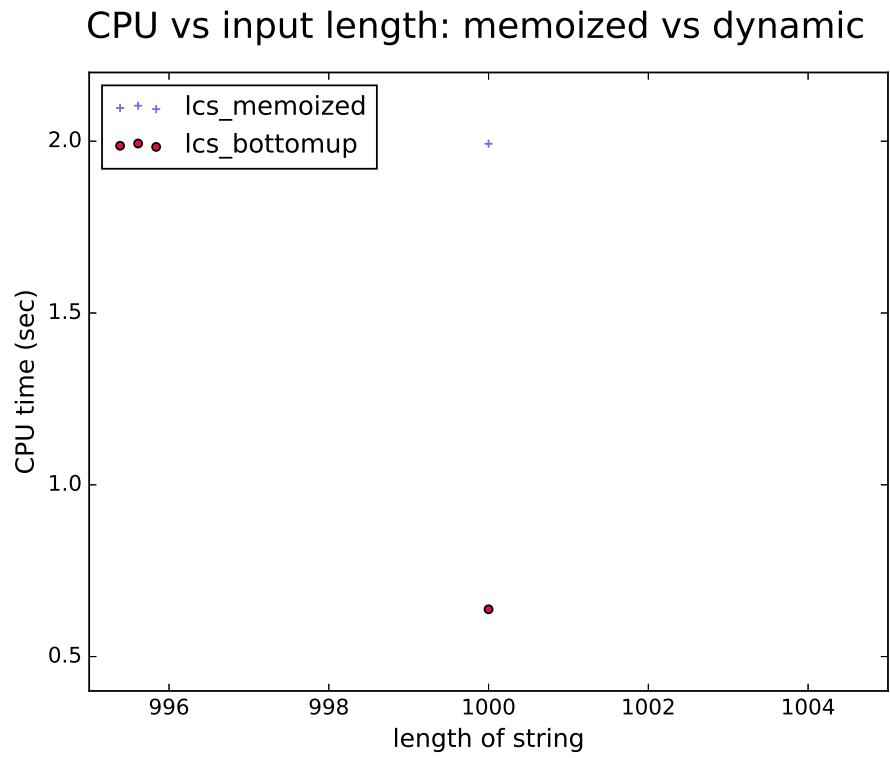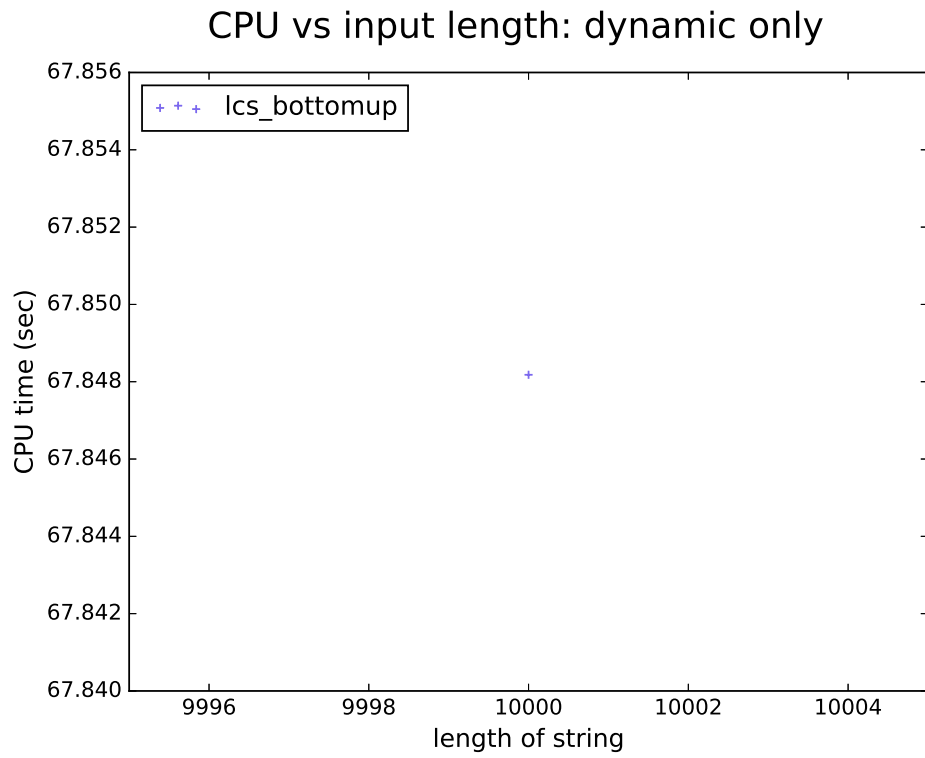Figure 4.2: CPU time for a long input string: dynamic algorithm.

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
```

```
44                    string)
45        """
46
47        if i < 0 or j < 0:
48            return lcs
49        else:
50            if seq1[i] == seq2[j]:
51                return _lcs_naive(seq1, seq2, i-1, j-1, \
52                        seq1[i] + lcs)
53            else:
54                return max(_lcs_naive(seq1, seq2, i-1, j, \
55                        lcs),
56                        _lcs_naive(seq1, seq2, i, j-1, lcs),
57                        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:

```python
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]
```

```
41
42  @len_recursion
43  def _lcs_memoized(seq1, seq2, i, j, lcs_table):
44      """Recursive solution with memoization to LCS problem.
45      See CLRS ex. 15.4-3.
46
47      Args:
48          seq1 (string): a string sequence generated by
49                          generate_string.strgen()
50          seq2 (string): another random string sequence like seq1
51          i (int): index into seq1
52          j (int): index into seq2
53          lcs_table (2D list): a matrix of LCS length for
54                          [i, j] prefix
55      Returns:
56          None: modifies in place LCS length table
57      """
58
59      if i < 0 or j < 0:
60          return 0
61      else:
62          if lcs_table[i][j] is not None:
63              return lcs_table[i][j]
64          else:
65              if seq1[i] == seq2[j]:
66                  val = 1 + \
67                      _lcs_memoized(seq1, seq2, i-1, \
68                                      j-1, lcs_table)
69              else:
70                  val = max(_lcs_memoized(seq1, seq2, \
71                                      i-1, j, lcs_table),
72                          _lcs_memoized(seq1, seq2, i, \
73                                      j-1, lcs_table))
74
75              lcs_table[i][j] = val
76              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:

```python
17  from profilers import len_recursion, time_profiler, registry
18  from generate_string import strgen
19
20  @time_profiler(repeat=1)
21  def lcs_bottomup(seq1, seq2):
22      """Calls helper function to calculate an LCS.
23
24      Args:
25          seq1 (string): a random string sequence generated by
26                         generate_string.strgen()
27          seq2 (string): another random string sequence like seq1
28      Returns:
29          LCS length (int)
30
31      """
32      len1 = len(seq1)
33      len2 = len(seq2)
34
35      # store length of LCS[i,j] in lcs_table
36      lcs_table = [[0 for j in range(len2+1)] \
37                     for i in range(len1+1)]
38      _lcs_bottomup(seq1, seq2, len1+1, len2+1,
39                    lcs_table)
40      return lcs_table[len1][len2]
41
```

```
42
43  # @to_profile
44  def _lcs_bottomup(seq1, seq2, i, j, lcs_table):
45      """Iterative bottom-up dynamic programming solution to
46      LCS problem. See CLRS p.394.
47
48      Args:
49          seq1 (string): a string sequence generated by
50                          generate_string.strgen()
51          seq2 (string): another random string sequence
52                          like seq1
53          i (int): number of rows in LCS table
54                          (=len(seq1) + 1)
55          j (int): number of columns in LCS table
56                          (=len(seq2) + 1)
57          lcs_table (2D list): a matrix of LCS length for
58                          [i-1, j-1] prefix
59      Returns:
60          None: modifies in place LCS length table
61      """
62
63      for row in range(1, i):
64          for col in range(1, j):
65              if seq1[row-1] == seq2[col-1]:
66                  lcs_table[row][col] = \
67                      lcs_table[row-1][col-1] + 1
68              elif lcs_table[row-1][col] \
69                      >= lcs_table[row][col-1]:
70                  lcs_table[row][col] = \
71                      lcs_table[row-1][col]
72              else:
73                  lcs_table[row][col] = \
74                      lcs_table[row][col-1]
```

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

20

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:

```
23  import os.path, sys
24  import importlib
25  from plot import plot_scatter
26  from generate_string import strgen
27
28  import naive
29  import memoized
30  import dynamic
31  #import hirschberg
32
33
34  # increase recursion limit
35  sys.setrecursionlimit(100000)
36
37  # set up directory refs
38  CURDIR = os.path.abspath(os.path.curdir)
39  FIGDIR = os.path.join(os.path.dirname(CURDIR),\
40          'docs/source/figures')
41  LOGDIR = os.path.join(CURDIR, 'logs')
42
43  # alphabets
44  ALPHAS = {'bin': ['0', '1'],
45          'alpha': ['A','C','G','T']}
46
47
48  if __name__ == "__main__":
49
50      ## (1) for each algorithm
51      # plot CPU time vs string length for several
52      # inputs at each length
53      # set legend by length of LCS or alphabet
54      str_lens = [10, 20]
```

```
55      title = 'CPU vs string length: naive algorithm'
56      series = []
57
58      set1 = {'x':[], 'y':[]}
59      seq1_bin = [strgen(alphabet=ALPHAS['bin'], \
60              size=strlen) for i in range(5) \
61              for strlen in str_lens]
62      seq2_bin = [strgen(alphabet=ALPHAS['bin'], \
63              size=strlen) for i in range(5) \
64              for strlen in str_lens]
65      for (str1, str2) in zip(seq1_bin, seq2_bin):
66          algo_name, time_elapsed, lcs = \
67                  naive.lcs_naive(str1, str2)
68          set1['x'].append(len(str1))
69          set1['y'].append(time_elapsed)
70      series.append('binary')
71
72      #store CPU vs length
73      set2 = {'x':[], 'y':[]}
74      set3 = {'x':[], 'y':[]}
75      set4 = {'x':[], 'y':[]}
76      seq1_alpha = [strgen(alphabet=ALPHAS['alpha'], \
77                  size=strlen) for i in range(5) \
78                  for strlen in str_lens]
79      seq2_alpha = [strgen(alphabet=ALPHAS['alpha'], \
80                  size=strlen) for i in range(5) \
81                  for strlen in str_lens]
82
83      for (str1, str2) in zip(seq1_alpha, seq2_alpha):
84          strlen = len(str1)
85          algo_name, time_elapsed, lcs = \
86                  naive.lcs_naive(str1, str2)
87          set2['x'].append(len(str1))
88          set2['y'].append(time_elapsed)
89          if strlen == 10:
90              set3['x'].append(\
91                      naive.registry['_lcs_naive'])
92              set3['y'].append(time_elapsed)
93          elif strlen == 20:
94              set4['x'].append(\
95                      naive.registry['_lcs_naive'])
96              set4['y'].append(time_elapsed)
97      series.append('alphabetic')
98      plot_scatter(set1, set2, series, title, \
99              xlabel = 'length of string', \
100             ylabel = 'CPU time (sec)')
101
102     print("-> done with naive algorithm runs")
103
104     # (2) plot CPU time vs number of recursive
105     # invocations for several lengths
106     plot_scatter(set3, set4, ['size 10', 'size 20'], \
107             title = 'CPU vs recursion depth', \
108             xlabel = 'number of recursive invocations',\
109             ylabel = 'CPU time (sec)')
110
111     print("-> done with CPU vs recursion depth plots")
```

23

```
112
113        # (3) TODO: plot memory use vs string length for
114        # several inputs at each length
115
116
117        ## (4) algorithm comparison
118        # for given string, plot CPU time for each
119        # algorithm; do this for several strings
120        # (vary length and alphabet)
121        title = 'CPU vs input length: memoized vs dynamic'
122        str1 = strgen(alphabet=ALPHAS['alpha'], size=1000)
123        str2 = strgen(alphabet=ALPHAS['alpha'], size=1000)
124
125        set1 = {'x':[], 'y':[]} #store CPU vs length: memoized
126        labels = []
127        algo_name, time_elapsed, lcs_len = \
128                memoized.lcs_memoized(str1, str2)
129        set1['x'].append(len(str1))
130        set1['y'].append(time_elapsed)
131        labels.append(algo_name)
132
133        #store CPU vs length: bottom-up dynamic
134        set2 = {'x':[], 'y':[]}
135        algo_name, time_elapsed, lcs_len = \
136                dynamic.lcs_bottomup(str1, str2)
137        set2['x'].append(len(str1))
138        set2['y'].append(time_elapsed)
139        labels.append(algo_name)
140        plot_scatter(set1, set2, labels, title, \
141                xlabel = 'length of string', \
142                ylabel = 'CPU time (sec)')
143
144
145        print("-> done with CPU vs length plots")
146
147        ## (5) show time for strlen == 40000
148        title = 'CPU vs input length: dynamic only'
149        str1 = strgen(alphabet=ALPHAS['alpha'], \
150                size=10000)
151        str2 = strgen(alphabet=ALPHAS['alpha'], \
152                size=10000)
153
154        set1 = {'x':[], 'y':[]} #store CPU vs length: dynamic
155        labels = []
156        algo_name, time_elapsed, lcs_len = \
157                dynamic.lcs_bottomup(str1, str2)
158        set1['x'].append(len(str1))
159        set1['y'].append(time_elapsed)
160        labels.append(algo_name)
161
162        plot_scatter(set1, [], labels, title, \
163                xlabel = 'length of string', \
164                ylabel = 'CPU time (sec)')
165
166
167        print("-> done with plot for dynamic @ 10K long string")
```

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

Chapter 11

Appendix 6: Plotter Program

Listing for the plotting routine:

```python
1  import matplotlib.pyplot as plt
2  import os.path
3
4  CURDIR = os.path.abspath(os.path.curdir)
5  DOCDIR = os.path.join(os.path.dirname(CURDIR), \
6          'docs/source/figures')
7
8  def plot_scatter(set1, set2, labels, title, xlabel, ylabel):
9      """Save 2D scatter plot of numplots sets of data.
10
11     Args:
12         set1 (dict): dict of coordinate lists; set1 =
13                      {'x':[list of x-coords],
14                      'y':[list of y-coords]}
15         set2: ditto
16         labels (list of strings): series labels
17                                   for each data set
18         title (string): plot title
19         xlabel, ylabel (string): axes labels
20     """
21
22     fig = plt.figure()
23     axes = plt.gca()
24     axes.set_xlim([set1['x'][0] - 5, \
25             set1['x'][-1] + 5])
26     plt.ticklabel_format(style='plain', axis='both', \
27             useOffset = False)
28     fig.suptitle(title, fontsize=20)
29     plt.xlabel(xlabel, fontsize=14)
30     plt.ylabel(ylabel, fontsize=14)
31     if set2:
32         plt.scatter(set1['x'], set1['y'], s=20,
```

```
33                c='mediumslateblue',
34                marker='+', label = labels[0])
35          plt.scatter(set2['x'], set2['y'], s=20, c='crimson',
36                marker='o', label = labels[1])
37     else:
38          plt.scatter(set1['x'], set1['y'], s=20,
39                c='mediumslateblue',
40                marker='+', label = labels[0])
41
42     plt.legend(loc='upper left')
43     fname = '_'.join(title.replace(':','_').split()) + '.ps'
44     fig.savefig(os.path.join(DOCDIR,fname))
45     #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:

```python
#!/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 13

Appendix 8: Performance Profiler Program

Listing for runtime, recursion depth and memory profilers:

```python
19  import time, sys
20  from memory_profiler import profile as mem_profiler
21  from collections import defaultdict
22
23  # keep track of recursive function calls
24  registry = defaultdict(int)
25
26  def len_recursion(func):
27      """Decorator that counts the number of function
28      invocations.
29
30      Args:
31          func: decorated function
32      Returns:
33          decorated func
34      Caveats:
35          does not account for repeated runs!
36      """
37      # count number of invocations
38      def inner(*args, **kwargs):
39          """Increments invocations and returns the
40          callable unchanged."""
41
42          registry[func.__name__] += 1
43          return func(*args, **kwargs)
44      return inner
45
```

```python
46
47  def time_profiler(repeat = 1):
48      """Decorator factory that times the function
49      invocation. A function is timed over 'repeat' times
50      and then runtime is averaged.
51
52      Args:
53          repeat (int): number of repeat runs to average
54                              runtime over.
55      Returns:
56          decorated func
57      """
58      def decorate(func):
59          """Decorator.
60
61          Args:
62              func: decorated function
63          """
64          def inner(*args, **kwargs):
65              """Sets timer and returns the elapsed time
66              and result of original function.
67
68              Returns:
69                  func.__name__, elapsed_time,
70                      original_return_value (tuple)
71              """
72
73              start = time.perf_counter()
74              for i in range(repeat):
75                  return_val = func(*args, **kwargs)
76              finish = time.perf_counter()
77              elapsed = finish - start
78
79              return (func.__name__, elapsed, return_val)
80          return inner
81      return decorate
```

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

References

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