Eitan Romanoff
Mark Thill
Theory of Algorithms Update Report

1. Tools

All four of the LCS variants, as well as the profiling and time processing, has been implemented in python. Python was chosen for its powerful string slicing and simplicity in running on any platform. The higher level script that runs the profiled tests is also written in python, and makes use of python's subprocess module in order to run the LCS algorithms.

2. Naive

Shown below is the code for finding the longest common subsequence using the naive recursive algorithm.

```python
global_rcalls = 0


def resetGlobals():
    global global_rcalls
    global_rcalls = 0


def naiveGetLCS(first, second):
    """
    This function will return the longest common subsequence between two
    strings using a naive recursive approach with no performance boosts.
    """
    # global to count recursive calls
    global global_rcalls

    # base case
    if len(first) == 0 or len(second) == 0:
        return ''

    # match case
    elif first[-1:] == second[-1:]:
        global_rcalls += 1
        return naiveGetLCS(first[:-1], second[:-1]) + first[-1:]

    # split case
    else:
        c1 = naiveGetLCS(first, second[:-1])
        c2 = naiveGetLCS(first[:-1], second)
        global_rcalls += 2
        if len(c1) >= len(c2):
            return c1
        else:
            return c2
```

naiveGetLCS() as defined in lcs.py

The algorithm also uses global variables to keep track of the number of recursive calls made by the function. In order to keep these globals fresh for every run, a function for resetting them was implemented, and is used by the higher level script. The function listed above takes in two strings and returns the longest common subsequence.

3. Profiling

The higher level python script currently only profiles the naive approach, but can be easily modified in the future to analyze all four LCS implementations. The script generates two strings of length n, and profiles the function called until the computation time exceeds 10 CPU seconds. Profiling is implemented using python's fast built-in cProfile module. One challenge was that the cProfile module simply outputs the profile, or dumps binary data, to be parsed using a separate cProfile tool. In order to capture the profiler's output as a string, the profiling call was delegated as a separate script, and the printed output was piped into the higher level script. This output was parsed to see when the computation target was hit. Example output of the profiler is listed below.

```
eitanromanoff@pinklady RIT-LCS-Project $ python profilelcs.py
Finding length n strings to get close to ~10 seconds computation...
N(1)  => 0.0 CPU seconds
N(2)  => 0.0 CPU seconds
N(3)  => 0.0 CPU seconds
N(4)  => 0.0 CPU seconds
N(5)  => 0.001 CPU seconds
N(6)  => 0.003 CPU seconds
N(7)  => 0.01 CPU seconds
N(8)  => 0.038 CPU seconds
N(9)  => 0.177 CPU seconds
N(10)  => 0.55 CPU seconds
N(11)  => 2.044 CPU seconds
N(12)  => 7.749 CPU seconds

================================================
                Target hit!
Configuration for Naive LCS in 29.8 CPU seconds.
  String lengths: 13
  First:  0000000000000
  Second: 1111111111111
  LCS:
  Recursive Calls:  20801198

         78004497 function calls (57203299 primitive calls) in 29.800 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   29.800   29.800 <string>:1(<module>)
20801199/1   24.338    0.000   29.800   29.800 lcs.py:9(naiveGetLCS)
 57203296    5.462    0.000    5.462    0.000 {len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Example profiling run, halting when CPU time > 10 seconds

4. Understanding the Results

As is obvious from the above output, as n increases, so does computation time. The tests run above use strings that contain no common subsequence ({0}*n and {1}*n), such that the worst case is always hit, allowing the runtime to be monotonically increasing. Using random sequences will create large jumps in computation time depending on how many matches are found in searching for the LCS. For the strings described previously, given our implementation and the naive recursive approach, two strings of length 12 is the maximum length that can be achieved while staying below a CPU computation time of 10 seconds.

5. Next Steps

The profiling script will be altered such that it can be run from the command-line in a clean manner. Customization features include the following.

Computation Target Time
Strings (pre-determined, randomly generated, “fixed”)
Number of Repetitions
Verbose, Concise, Silent modes
Output can be tossed to a file