Project 3: String Matching with Dynamic Programming Part I

By:
Cristopher Hernandez (Bernstein)
cristopherh@csu.fullerton.edu

Carmine Infusino (Wortman)
c.infusino@csu.fullerton.edu

Christian Medina (Bernstein)
christian.medina@csu.fullerton.edu

Hypothesis

This experiment will test the following hypotheses:

1. Exhaustive longest common subsequence string matching is possible and will yield the correct result but even for short strings the algorithm may be too inefficient for real use.
2. Dynamic Programming solutions are very fast when compared to exhaustive algorithms and produce optimal solutions for string matching.

Mathematical Analysis of Pseudocode

Problem statement:

Longest common subsequence
input: two strings: S1 and S2. *output:* the length of the longest common subsequence shared between S1 and S2.

Exhaustive Algorithm:

```
exhaustive_longest_common_subsequence(S1, S2):
      all_subseqs1 = generate_all_subsequences(S1);
      all_subseqs2 = generate_all_subsequences(S2);
      best_score = 0
      for s1 in all_subseqs1
            for s2 in all_subseqs2
                  if (s1 = s2 and length(s1) > best_score)
                        best_score = length(s1)
      return best_score
```

Where generate_all_subsequences is defined as:

```
generate_all_subsequences(S)
      R = {}
      n = 2^size(S)
      for bits = 0 to n - 1
            subsequence = “”
            for j = 0 to size(S) - 1
                  if ((bits >> j) & 1) == 1
```

```
                    subsequence.append(S[j])
            R.insert(subsequence)
    return R
```

Looking at generate_all_subsequences first: a chronological step count gives us:

$$f(n) = 2^n * n + 3$$

To prove that generate_all_subsets $\in O(2^n * n)$ let $g(n) = 2^n * n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \to \lim_{n\to\infty} \frac{2^n*n+3}{2^n*n} \to \lim_{n\to\infty} \frac{2^n*n}{2^n*n} + \frac{3}{2^n*n} \to \lim_{n\to\infty} 1 + 0 \to 1$$

The result is a constant, therefore f(n) = g(n) as $n \to \infty$ and generate_all_subsets $\in O(2^n * n)$. [1]

Accounting for generate_all_subsets, we redefine f(n) and a chronological step count (discarding constant values) gives us:

$$f(n,m) = (2^n * n) + (2^m * m) + (2^{n+m})$$

As $n, m \to \infty$, we consider $m \in n$ as n will be dominating. As a result, the equation can be simplified to:

$$f(n) = 2^n * n + 2^n * n + 2^{2n}$$

Let $g(n) = 2^{2n}$ for the purpose of proving that $f(n) \in O(2^{2n})$.

$$\lim_{n\to\infty} \frac{2^n*n+2^n*n+2^{2n}}{2^{2n}} \to \lim_{n\to\infty} \frac{2^n*n}{2^{2n}} + \frac{2^n*n}{2^{2n}} + \frac{2^{2n}}{2^{2n}} \to \textit{L'Hopital's Rule} \to \frac{1}{ln(2)} \lim_{n\to\infty} \frac{1}{2^n} + \frac{1}{ln(2)} \lim_{n\to\infty} \frac{1}{2^n} + \lim_{n\to\infty} 1 \to 0 + 0 + 1$$
$$\to 1$$

Which is a non-negative, constant with respect to n.

Alternatively, we can prove the efficiency class, O(2^(2n), by using Properties of O:
$f(n) = (2^n * n) + (2^m * m) + 2^{(n+m)} \in O((2^n * n) + (2^m * m) + (2^{n+m}))$

Here, the first two terms would be dominated by the last term.

[1] See Algorithm Design in Three Acts (2017) p.163 for a corroborating claim for this algorithm's complexity

Thus, we can rewrite it as:

$O((2^{n+m})$

Since $m \in n$ as $n, m \to \infty$, it can also be simplified to:

$O(2^{n+n})$

$= O(2^{2n})$.

Thus, the result is the same and the exhaustive algorithm is found to be $O(2^{2n})$ time complexity.

Dynamic Programming Algorithm:

```
dynamicprogramming_longest_common_subsequence(S1, S2):
      n = size(S1)
      m = size(S2)
      D[n+1][m+1]
      for i = 0 to n
            D[i][0] = 0
      for j = 0 to m
            D[0][j] = 0

      for i = 1 to n
            for j = 1 to m
                  up = D[i-1][j]
                  left = D[i][j-1]
                  diag = D[i-1][j-1]
                  if (S1[i-1] = S2[j-1])
                        diag = diag + 1
                  D[i][j] = max(up, left, diag)

      return D[n][m]
```

Using chronological step counting on dynamicprogramming_longest_common_subsequence we get (while assuming functions like size() and max() have O(1) complexity):

$$f(n, m) = n * m + n + m + 4$$

To prove that $f(n, m) \in O(n * m)$ let $g(n) = n * m$

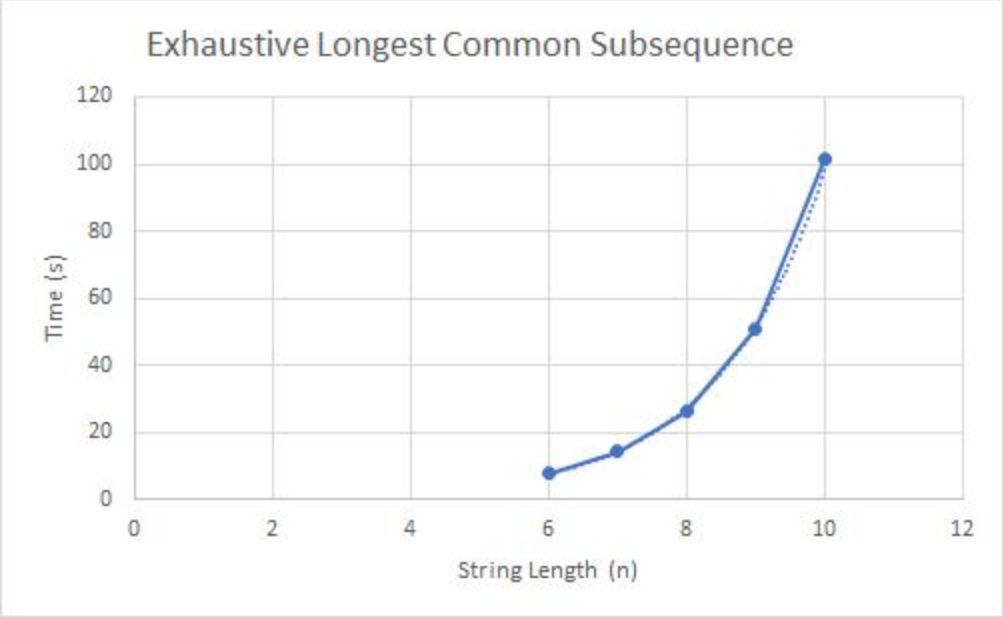
$$\lim_{(n,m)\to(\infty,\infty)} \frac{n*m+n+m+4}{n*m} \to \lim_{(n,m)\to(\infty,\infty)} \frac{n*m}{n*m} + \frac{n}{n*m} + \frac{m}{n*m} + \frac{4}{n*m} \to \lim_{(n,m)\to(\infty,\infty)} 1 + 0 + 0 + 0 \to 1$$

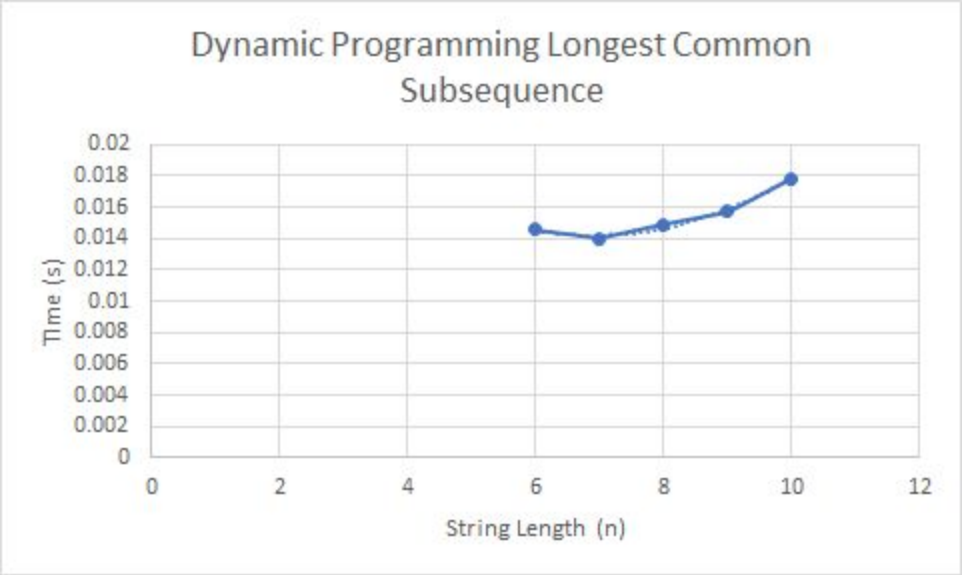
The result is a constant, therefore f(n,m) = g(n,m) as $n, m \to \infty, \infty$ and $f(n, m) \in O(n * m)$

Empirical Analysis: Scatterplots and Data of Implemented Functions

Exhaustive Longest Common Subsequence	
String Length	Time
6	7.89852
7	14.1969
8	26.3761
9	50.9967
10	101.261

Dynamic Programming Longest Common Subsequence	
String Length	Time
6	0.0145453
7	0.0139953
8	0.0148899
9	0.0157501
10	0.0178533





Experiment Results

```
TOTAL SCORE = 21 / 21

[c.infusino@jupiter project-3-team-c-master]$ ./experiment
------------------ Exhaustive Method ------------------
String to Match = QSDITV
sp|P32469|DPH5_YEAST Diphthine synthase OS=Saccharomyces cerevisiae (strain ATCC 204508 / S288c) GN=DPH5 PE=1 SV=1
7.89852
String to Match = KDITVXR
sp|P32469|DPH5_YEAST Diphthine synthase OS=Saccharomyces cerevisiae (strain ATCC 204508 / S288c) GN=DPH5 PE=1 SV=1
14.1969
String to Match = YKSDTWRN
sp|P32469|DPH5_YEAST Diphthine synthase OS=Saccharomyces cerevisiae (strain ATCC 204508 / S288c) GN=DPH5 PE=1 SV=1
26.3761
String to Match = AYKDIRNLX
sp|Q08032|CDC45_YEAST Cell division control protein 45 OS=Saccharomyces cerevisiae (strain ATCC 204508 / S288c) GN=CDC45 PE=1 SV=1
50.9967
String to Match = BQSITVARGL
sp|P32469|DPH5_YEAST Diphthine synthase OS=Saccharomyces cerevisiae (strain ATCC 204508 / S288c) GN=DPH5 PE=1 SV=1
101.261
------------------ Dynamic Programming ------------------
String to Match = QSDITV
sp|P32469|DPH5_YEAST Diphthine synthase OS=Saccharomyces cerevisiae (strain ATCC 204508 / S288c) GN=DPH5 PE=1 SV=1
0.0145453
String to Match = KDITVXR
sp|P32469|DPH5_YEAST Diphthine synthase OS=Saccharomyces cerevisiae (strain ATCC 204508 / S288c) GN=DPH5 PE=1 SV=1
0.0139953
String to Match = YKSDTWRN
sp|P32469|DPH5_YEAST Diphthine synthase OS=Saccharomyces cerevisiae (strain ATCC 204508 / S288c) GN=DPH5 PE=1 SV=1
0.0148899
String to Match = AYKDIRNLX
sp|Q08032|CDC45_YEAST Cell division control protein 45 OS=Saccharomyces cerevisiae (strain ATCC 204508 / S288c) GN=CDC45 PE=1 SV=1
0.0157501
String to Match = BQSITVARGL
sp|P32469|DPH5_YEAST Diphthine synthase OS=Saccharomyces cerevisiae (strain ATCC 204508 / S288c) GN=DPH5 PE=1 SV=1
0.0178533
```

Four out of the five proteins matched by the algorithms corresponded to sp|P32469|DPH5_YEAST Diphthine synthase. The single protein which is different is the same for both: sp|Q08032|CDC45_YEAST Cell division control protein 45. While it was given that the expected result should be uniform to the same protein, this result may still be correct for the

algorithms being used. This is because insertions and deletions are allowed in the strings being matched. If the algorithms were to use penalties to take these into account that may alter the results.

Conclusion

Based off of our empirical data, we can conclude that our observed time efficiency is consistent with the mathematically-derived big-O efficiency classes we developed through our step-count and analyses. The plot for our exhaustive algorithm appears to follow an exponential trendline which is consistent with the expected slow, exponential time complexity we derived. Our empirical analysis yielded a surprising result for the 7 character long since it was faster than the 6 character long string for the dynamic programming algorithm. However, even with this outlier, our results still are consistent since our plot follows a polynomial trendline which was expected.