Project 3: polynomial versus exponential time

Group members: Brad Dodds bradleydodds@csu.fullerton.edu
CWID: 889763546

Introduction

In this project I have created and compared two algorithms that solve similar problems. I have investigated the difference between polynomial and exponential time complexities of the two algorithms. The first algorithm solves the longest common substring problem with expected time complexity of O(n^3). The second algorithm solves the longest common subsequence problem with an expected time complexity of O(2^n * n).

The Hypotheses

This experiment will test the following hypotheses:

1. Exhaustive search algorithms are feasible to implement, and produce correct outputs.
2. Algorithms with exponential running times are extremely slow, probably too slow to be of practical use.

Problem1: longest common substring
input: a string a of length m and a string b of length n
output: the longest string s such that s is a substring of both a and b; in the case of ties, use the substring that appears first in a

Problem2: longest common subsequence
input: a string a of length m and a string b of length n
output: the longest string s such that s is a subsequence of both a and b; in the case of ties, use the substring that appears first in a

Analyze Time Complexity

Longest Common Substring: O(n^3)

```
string substring = "";
string best = "";

for(int i = 0; i < a.length(); i++) {

    for(int j = 1; j <= a.length() - i; j++){

        substring = a.substr(i, j);

        cout << substring << endl;

        if(b.find(substring) != string::npos && substring.length() > best.length()){

            best = substring;

            }

        }//END Inner-Loop
}//END Outer-Loop

cout << best << endl;
```

-The two nested for loops represent O(n^2) time.

-b.find(substring) represent O(n) time.

-Total time of O(m(m-1)n)=O(m2n)

-When m = n, O(m2n)=O(n^3)

Longest Common Subsequence: O(2^n * n)

```
std::string candidateSubset = "";
std::string shorter = "";
std::string longer = "";
std::string best = "";
std::vector<string> candidate;

if(a.length() > b.length()) {
    longer = a;
    shorter = b;
}

else {
    shorter = a;
    longer = b;
}

candidate = subsets(shorter);

for(int index = 0; index < candidate.size(); index++){

    if(detect_subsequence(candidate[index], longer) == true && candidate[index].length() > best.length()) {

        best = candidate[index];

    }

}

cout << "Final Best: " << best << endl;
```

-There are 2 input string's which lengths represent 2^n candidates

-detect_subsequence roughly takes O(n) time

-comparing length takes constant O(1) time

Subsequence Detection

```
Def detect_subsequence(candidate_subsequence, candidate_supersequence):

loopCounter = 0;

for i in candidate_subsequence.length()

    for j in candidate_supersequence.length()

    if candidate_subsequence[i] == candidate_supersequence[j]
    loopCounter = j + 1;
     break;


if counter == candidate_subsequence.length()

return true;

else:

return false
```

Description

The subsequence_detection function will detect if a subsequence exists or not. It will compare the current inputs:

1.) current subset of shorter string: (candidate_subsequence)
2.) the longest string: (candidate_supersequence)

$$1 + 2*n^2 + 1 + 1$$

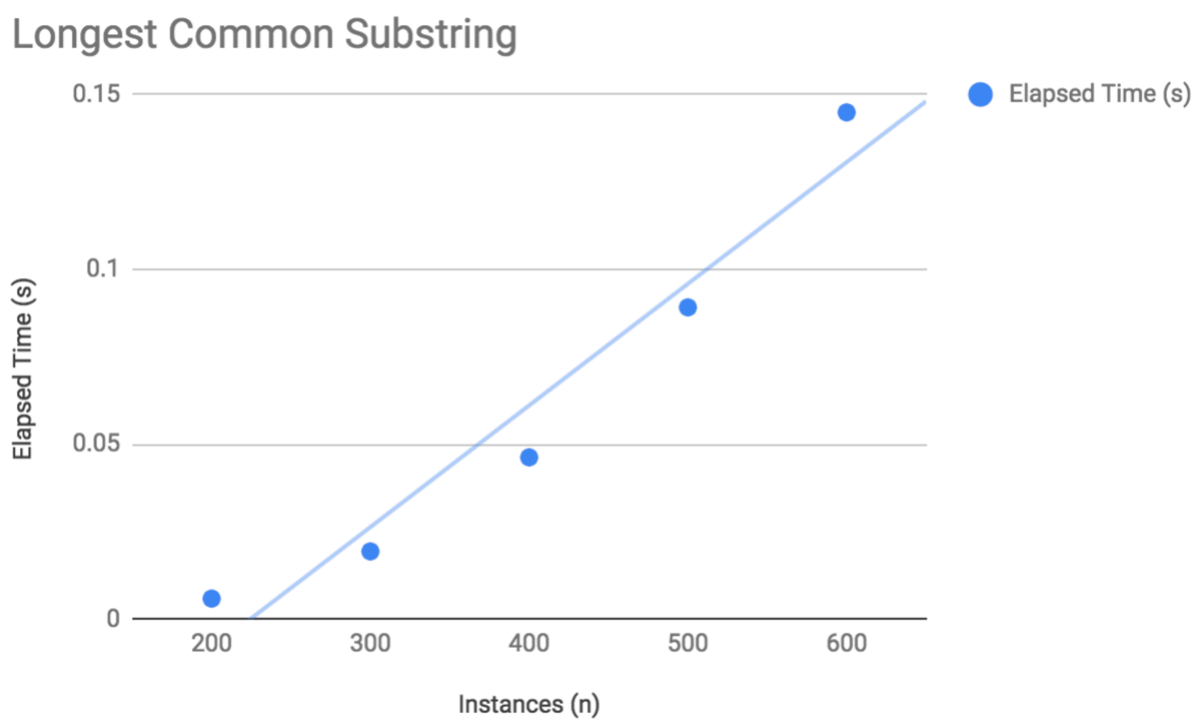
$$1 + 2n^2 +2$$

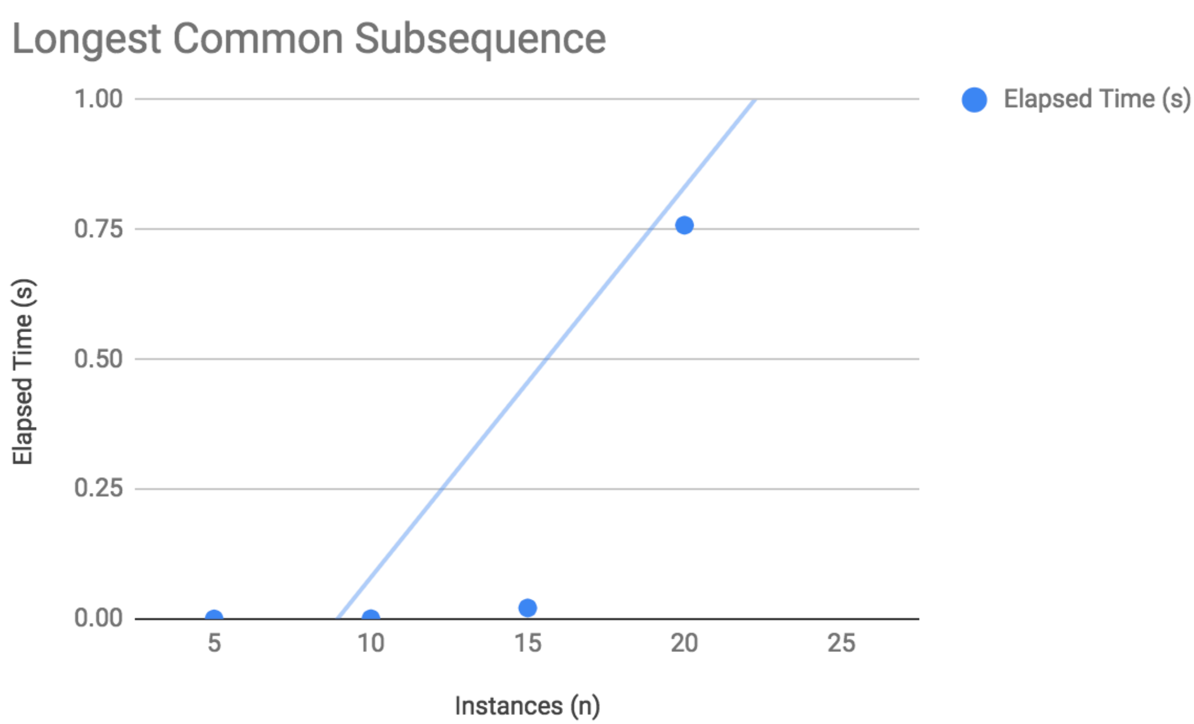
$$2n^2 + 3$$

$$= O(n^2)$$

The time complexity for subseuence_detection is O(n^2) time.

Performance





Comparison:

There is a very noticeable difference between the two algorithms. The longest common substring problem ran extremely faster than the longest common subsequence problem. As I increased the instance size (n) on both algorithms the longest common substring problem incremented at a steady pace. When the instances were incremented on problem 2 the time (s) jumped at a much larger noticeable rate.

For example:

Problem 1: Time stayed under 1.0 (second) as n incremented by the 100's.(n = 100…)

Problem 2: for 20 instances of n it ran in 0.75 seconds. For just 5 more instances of (n = 25) it ran around 31.2 seconds.

I expected that the subsequence problem would run slower than problem 1. However, I was surprised to see such a drastic increase in time just by incrementing n by 5.

Empirical vs. Mathematical:

The empirical analysis of this project is consistent with the initial mathematical analysis. Knowing the time complexities of the pseudocode and justifying the time complexities through execution of the algorithms shows a comparison between both approaches.

Conclusion:

After implementing and comparing both algorithms to solve a similar problem, I can say that polynomial and exponential run times with large inputs are very slow when it comes to execution.

By executing exhaustive search algorithms in the problems at hand I was able to gain the correct output. Based on the outcome of the algorithms I can say that hypothesis one is correct.

For hypothesis two, I agree that exponential running times are extremely slow with large inputs. However, depending on the problem at hand and the size of the input exponential functions may be feasible to solve that current problem. Overall, the hypothesis is correct when it comes to exponential time complexities running at a very slow rate.