Team members:
Cong Le Cle155@csu.fullerton.edu
Jack Nguyen nhannguyen7@csu.fullerton.edu
CPSC335-03: Algorithm Engineering
Professor Kevin Wortman
Due date: Friday, November 2, 1 pm.

Project 2- Empirical Analysis

The Hypotheses

This project will test the following hypotheses:

1. Exhaustive Optimization Algorithms are feasible to implement, and produce correct output.
2. Algorithms with exponential or factorial running times are extremely slow, probably too slow to be of practical use.

The Longest Increasing Subsequence Problem: **Input**: a vector *A* of *n* comparable elements. **Output**: a vector *R* containing the longest increasing subsequence of *A*

We will analyze the two algorithms (End-to-Beginning and Exhaustive) in details by examine their pseudocodes and time complexity.

Part I: Description of the Algorithms:

1. End-to-Beginning Algorithm:

- This algorithm is a straightforward approach to solve the problem and has O(n^2) time complexity.
- This algorithm can be described in plain English as following:
  - Initially, we need an additional vector *H* of length *n*, of non-negative integers.
  - The value *H[i]* will indicate how many elements, greater or equal to than *A[i]*, are further in the sequence *A* and have some special property.
  - The algorithm proceeds by attempting to increase the values of *H* starting with the previous to last element and going down to the first element.
  - Then a longest subsequence can be identified by selecting elements of *A* in decreasing order of the *H* values, starting with the element in *A* that has the largest *H* value.
- This algorithm can be described in sequential steps as following:

```
Step 1. Initiate vector H of length n and filled with all 0's
Step 2. Starting with H[n-2] and going down to H[0] try to increase the value
of H[i] as follows:
    Step 3. Starting with index (i+1) and going up to (n-1) (the last index in
    the vector A) repeat Steps 4 and 5:
            Step 4. See if any element is bigger than A[i] and has its H value
            also bigger to H[i].
            Step 5. If yes, then A[i] can be followed by that element in an
            increasing subsequence, thus set H[i] to be 1 plus the H value of
            that element.
    Step 6. Calculate the largest (maximum) value in vector H. By adding 1 to
    that value we have the length of a longest increasing subsequence.
    Step 7. Identify a longest subsequence by identifying elements in vector A
    that have decreasing H values, starting with the largest (maximum) value in
    vector H.
```

2. The Exhaustive Algorithm:

- This algorithm can solve the problem in O($n * 2^n$) time complexity.
- This approach is an exhaustive optimization algorithm.
- This approach will generate all possible subsequences of the vector A and will test each subsequence on whether it is in increasing order. The longest such subsequence is a solution to the problem.

- A subsequence can be uniquely identified by the set of indices in the vector A that are part of the subsequences.
- To generate all possible subsequences, we will generate the power set of {0, 1,…, n-1} and consider each subset of the power set as the set of indices in A of the subsequence.
- We will generate the power set of the set {1, 2,…, n} using an iterative algorithm that uses a stack to grow and shrink the set as needed.

Part II: Pseudocode for the Algorithms and Step Count:

1. **End-to-Beginning Algorithm:**

```
Define longest_increasing_end_to_beginning(std::vector<int> A)                1
        define n = A.size()                                                   1
        // populate the additional vector H of length n filled with all 0's
        std::vector<size_t> H(n, 0)                                           1
        // calculate the values of vector H
        for i from (n-2) going down to 0 in vector H:                         1          n-1
          for j from (i+1) going up to (n-1) in vector A:                     1          (n-1) – (i+1)
                if(A[i] < A[j]) and (H[i] < H[j])                             2          = n –i
                    H[i] = H[j] + 1                                           1          = n
        // calculate in maximum the length of the longest subsequence
        // by adding 1 to the largest (maximum) value in vector H
           auto maximum = *std::max_element(H.begin(), H.end() ) +1           n+1
        // allocate space for the subsequence vector R
        std::vector<int> R(maximum)                                           1
        // adding elements to vector R by whose H's values are in decreasing order,
        // starting with (maximum -1), and store in index the H values sought
        define index = maximum -1                                             1
        define j = 0                                                          1
        for i from 0 to n in vector H:                                        1          n
           if (H[i] == index )                                                1
                // adding A[i] to the sequence R
                R[j] = A[i]                                                   1
                decrement index                                               1
                increment j                                                   1
return std::vector<int> (R.begin(), R.end() + maximum )                       n+1
```

2. The Exhaustive Algorithm:

```
Define longest_increasing_powerset(std::vector<int> A):                    1
    std::vector<int>  best   // best is None initially                     1
    for candidates in Powerset(A):                                         O(g) = n · 2^n and O(c) = 2^n
      if is_increasing(candidate):                                         O(v) = n
          if( best is None) or (candidate.size() > best.size() )           1 + O(b) = 1+1 = 2
                best = candidate                                           1
 return best                                                               1
```

##

<GENERATING CANDIDATES> (<INPUT>) and <VERIFIER>(candidate) are described in details as following:

<GENERATING CANDIDATES> (<INPUT>):

```
void Powerset (std::vector<int> A) {  // function to generate the power set of {1, .., n} and retrieve the best set
    define n = A.size()                                                    1
    // populate a stack of length (n+1) filled with all 0's
    std::vector<size_t> stack(n + 1, 0);                                   1
    define k = 0;                                                          1

    while(true) {                                                          1
        if (stack[k] < n) {                                                1
            stack[k+1] = stack[k] + 1;                                     1
            increment k;                                                   1
        }
        else {
            increment stack[k-1];                                          1
            decrement k;                                                   1
        }
        if (k==0) break;                                                   2

        std::vector<int> candidate                                         1
        for i from 1 going up to k inclusively in vector A:                1          n
            candidate.push_back(A[stack[i] – 1 ])                          4
    }
}
```

Since this loop is depend on the size **n** passed from the program, this loop will take 2^n

$$\therefore O(g) = n \cdot 2^n$$

<VERIFIER>(candidate):

```
bool is_increasing(candidate){
    for i from 1 going up to candidate.size():                             1          n
        if(candidate[i] < candidate[i-1])                                  1
            return false                                                   1
return true                                                                1
}
```

$$\therefore O(v) = n$$

<u>Part III</u>: Mathematical Analysis for the Algorithms:

1. End-to-Beginning Algorithm:

$$T(n) = 1 + 1 + 1 + (n-1)\cdot[1 + n\cdot(4)] + (n+1) + 1 + 1 + 1 + n\cdot(5) + (n+1)$$
$$= 3 + (n-1) + (n-1)\cdot(4n) + 2n + 5 + 5n$$
$$= n + 2 + 4n^2 - 4n + 7n + 5$$
$$= 4n^2 + 4n + 7$$
$$\in O(4n^2 + 4n + 7) \quad \text{(trivial)}$$
$$= O(4n^2) \quad \text{(dominated term)}$$
$$= O(n^2) \quad \text{(dropping the constant factor)}$$
$$\therefore \textit{The End} - \textit{to} - \textit{Beginning algorithm takes } \boldsymbol{O(n^2)} \textit{time}$$

2. The Exhaustive Optimization Algorithm:

Since we using the exhaustive optimization pattern to solve for this problem, we can use the following formula to analyze the time complexity class:

$$T(n) = O(g + c(v + b))$$
$$= O(n \cdot 2^n + 2^n(n + 1))$$
$$= O(n \cdot 2^n + n \cdot 2^n + 2^n)$$
$$= O(2n \cdot 2^n + 2^n)$$
$$= O(2n \cdot 2^n) \quad \text{(dominated term)}$$
$$= O(n \cdot 2^n) \quad \text{(dropping the constant factor)}$$

where,
Generating all candidates takes a total of $O(g) = n \cdot 2^n$ time
There are $O(c) = 2^n$ candidates
Verifying one candidate takes $O(v) = n$ time
Comparing two candidates takes $O(b) = 1$ time

$$\therefore \textit{The Exhaustive Optimization Algorithm takes } \boldsymbol{O(n \cdot 2^n)} \textit{ time}$$

Part IV: Graphs and Empirical Analysis on the Algorithms:

Empirical Timing Data	End-to-Beginning Algorithm Runtime	Exhaustive Optimization Algorithm Runtime
Instance Size n	**Elapsed Time (in seconds)**	
5	3.44E-06	4.760900E-05
10	4.85E-06	0.00389727
15	6.64E-06	0.103487
20	6.49E-06	5.02777
22	1.08E-05	9.73146
23	1.08E-05	20.9956
24	1.19E-05	42.576
25	1.11E-05	92.403

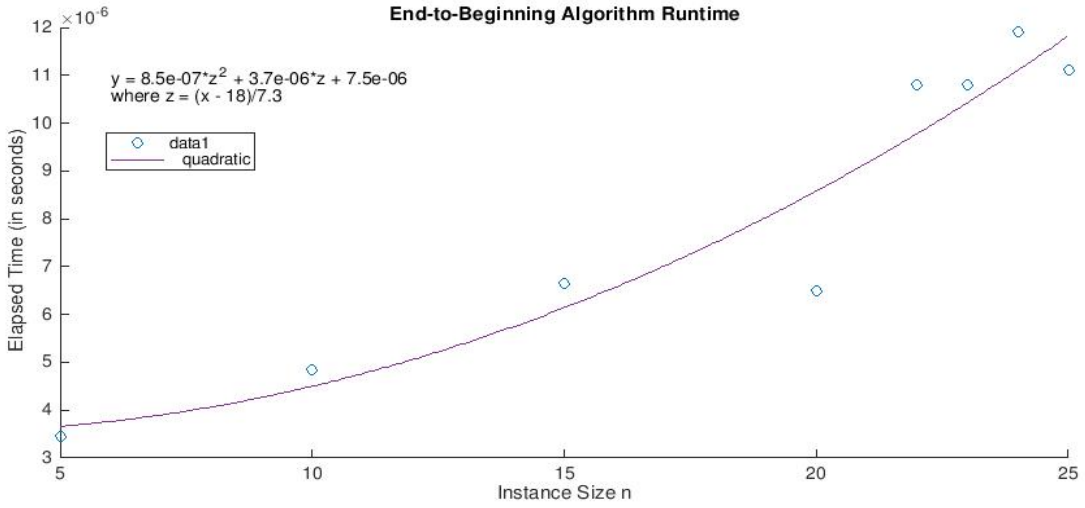

Figure1

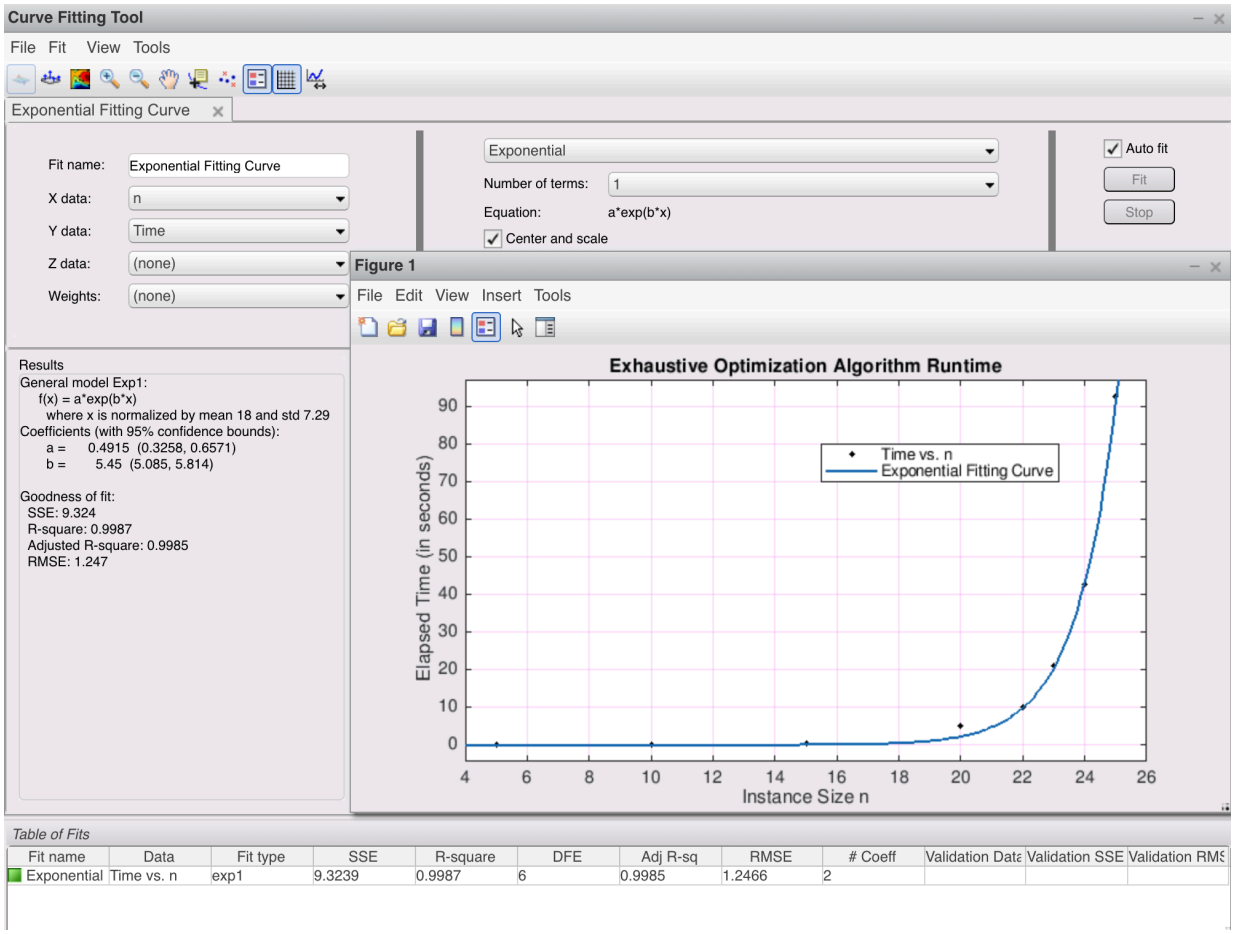

Figure2

To make our answers and conclusions on analyzing the two algorithm more robust, we create a third graph comparing runtime of both algorithms by using the same set of instance size n inputs as following:

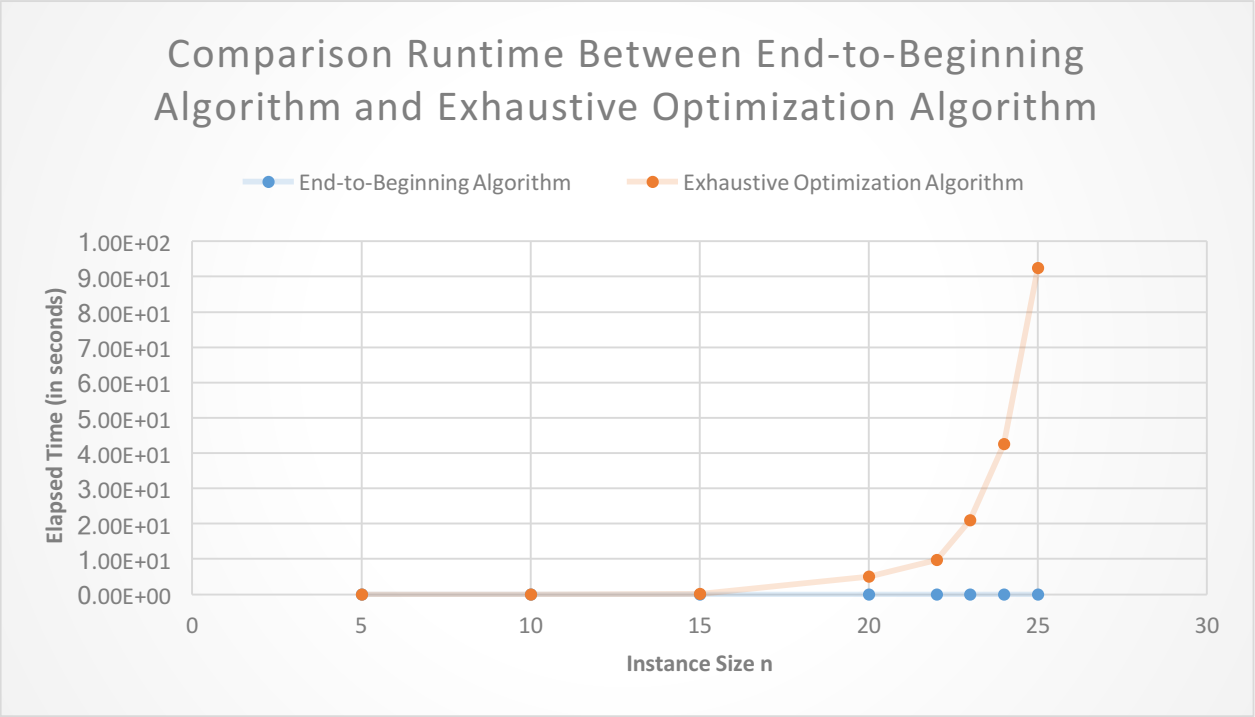

Figure3

Empirical Analysis

- We observe that the running speed for End-to-Beginning Algorithm is much more faster comparing with the Exhaustive Optimization Algorithm's. This observation is not surprise us because after mathematically analyzing, we already know that the efficiency class of End-to-Beginning Algorithm is O(n^2) time (quadratic time), which is faster than O($n * 2^n$) time (or, exponential time) of the Exhaustive Optimization Algorithm.
- The fit line on **Figure1** is a quadratic curve, which is ***consistence*** with the efficiency class of End-to-Beginning Algorithm as O(n^2) time, quadratic time.
- The fit line on **Figure2** is highly close to an exponential curve, which is ***consistence*** with the efficiency class of Exhaustive Optimization Algorithm as O($n * 2^n$) time, exponential time.
- Based on our evidences and observations on **Figure3**, we conclude that the initial hypotheses for this project is consistent with our experimental data:
  - Exhaustive Optimization Algorithm can provide a correct solution to the problem.
  - The runtime for Exhaustive Algorithm is super slow to be a practical use.