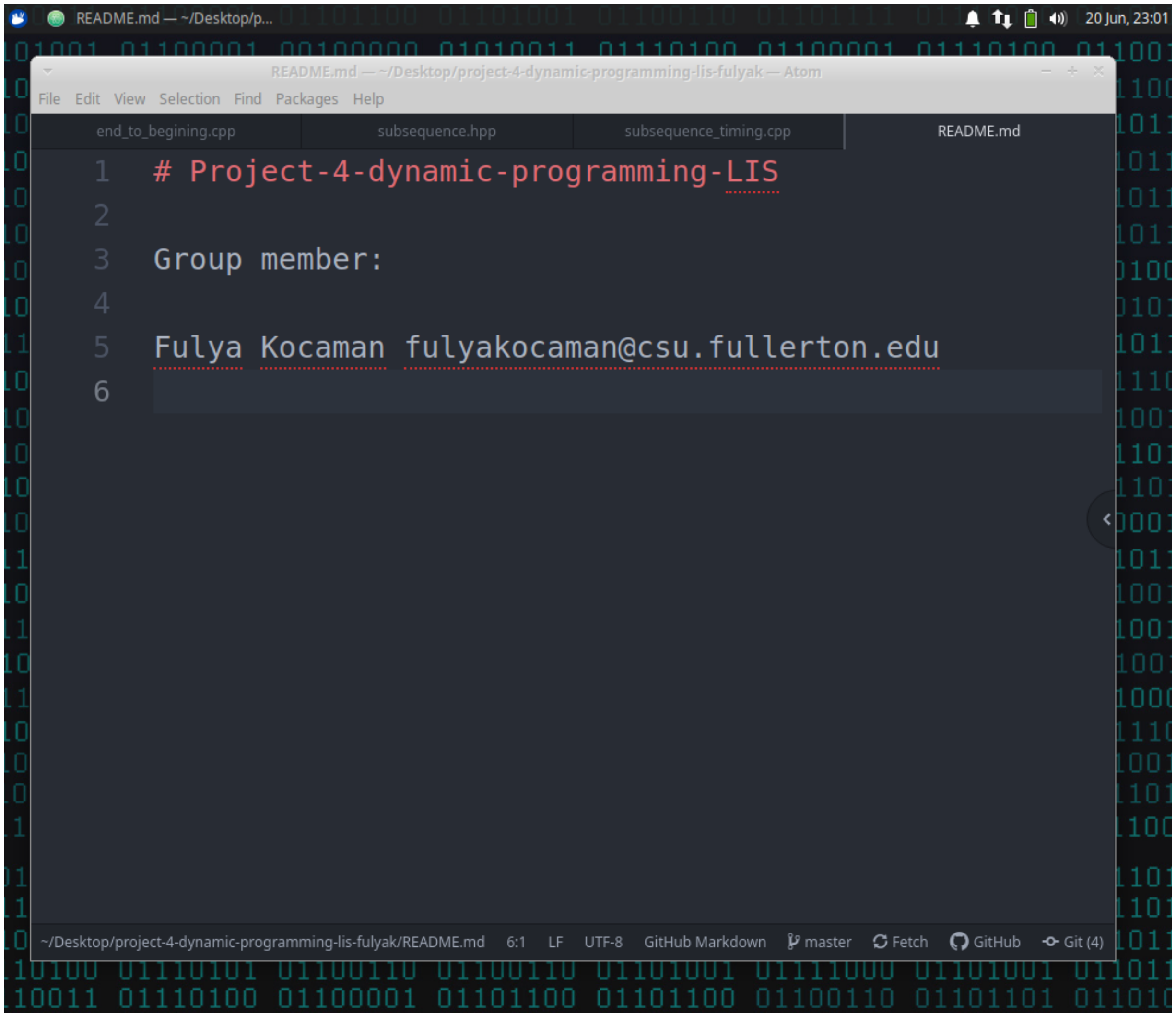
CPSC 335 **PROJECT 4 REPORT** **FULYA KOCAMAN**
SUMMER 2020 **fulyakocaman.csu.fullerton.edu**

In this project I will be solving the longest increasing subsequent problem using the dynamic programming subsequence algorithm. I will first write the pseudocode of the my algorithm from the C++ implementation, analyze it mathematically, measure its running time performance and prove that its time complexity is O(n^2), and then analyze the algorithm empirically by running it for various input sizes and plotting the timing data, compare my experimental results with the efficiency class of my algorithm, and draw conclusions. My experiment will test the following hypotheses:

1. For large values of n, the mathematically-derived efficiency class of an algorithm accurately predicts the observed running time of an implementation of that algorithm.
2. Polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive search algorithms that solve the same problem.


README.md — ~/Desktop/p...
20 Jun, 23:01
README.md — ~/Desktop/project-4-dynamic-programming-lis-fulyak — Atom
File Edit View Selection Find Packages Help
end_to_begining.cpp
subsequence.hpp
subsequence_timing.cpp
README.md
~/Desktop/project-4-dynamic-programming-lis-fulyak/README.md 6:1 LF UTF-8 GitHub Markdown master Fetch GitHub Git (4)

```
# Project-4-dynamic-programming-LIS

Group member:

Fulya Kocaman fulyakocaman@csu.fullerton.edu

```

1. **Pseudocode:**

   Input: a vector V of n comparable elements

   Output: a vector R containing the longest increasing subsequence of V.

```
def longest_increasing_end_to_begining(A)
 n = size of A                                   1 step
 H: vector of n integers, all initialized to 0   n steps
 stack: vector of integers with size of n+1
 for i = n-2 to 0 do
    for j = i+1 to n-1 do
        if(A[i] < A[j] && H[i] <= H[j])          3 steps
            H[i] = H[j] + 1                      2 steps
        endif
    endfor
  endfor

  max = the maximum value in H +1                3 steps
  R: vector of integers with size of max

  index = max -1                                 2 steps
  j = 0                                          1 step

  for i=0 to n-1                                 n times
      if (H[i] == index)                         2 steps
         R[j] = A[i]                             1 step
          index--                                1 step
         j++                                     1 step

return sequence(R.begin(), R.begin() + max)      2 steps
```

2. Step Count = $1+n+\left[\sum_{i=n-2}^{0}\sum_{j=i+1}^{n-1}(3+\max(2,0))\right]+3+2+1+n*[2+\max(3,0)]$

$= \sum_{i=n-2}^{0}\sum_{j=i+1}^{n-1}(5) + 6n + 7$

$= (n-1)*(n-1)*5 + 6n + 7$

$= 5n^2 - 5 + 6n + 1 = 5n^2 + 6n - 4$

3. Time Complexity: I suggest that the big-O time complexity of this algorithm is $O(n^2)$ since $O(5n^2+6n-4) = O(5n^2) = O(n^2)$.

4. Proof of Efficiency: Now, I will prove that the time complexity of this algorithm is actually $\in O(n^2)$ using both the definition and the limit theorem.

Prove by definition: I need to show that $5n^2+6n-4 \in O(n^2)$ if $\exists$ c > 0 and $n_o \geq 0$ such that $5n^2+6n-4 \leq c\, n^2\ \forall n \geq n_o$.

I will choose c = 5+6+4 = 15. Then, I have $5n^2+6n-4 \leq 15n^2$.
Now, need to find n_o

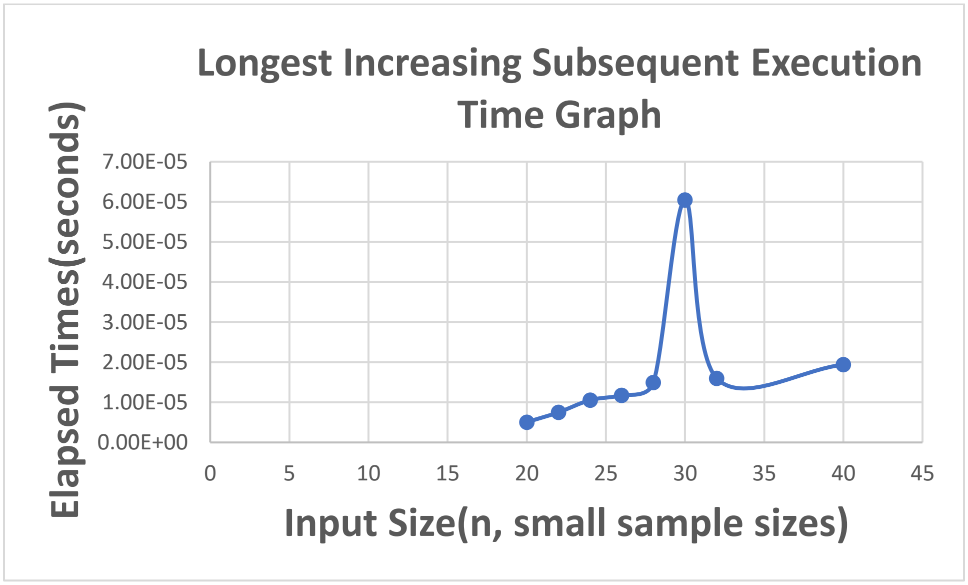
$15n^2 - 5n^2 - 6n + 4 \geq 0$

$10n^2 - 6n + 4 \geq 0$ satisfies $\forall n \geq 0$.

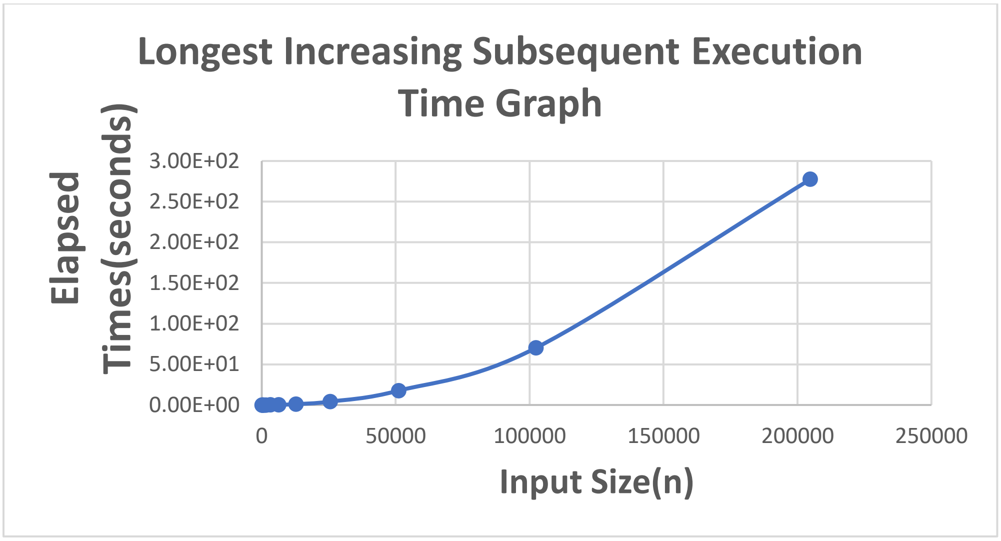
Therefore, choosing c = 15 and $n_o = 0$ shows that $5n^2+6n-4 \in O(n^2)$.

Prove by the limit theorem: I need to show that $5n^2+6n-4 \in O(n^2)$ using the limit theorem.
lim_{n->inf} $(5n^2+6n-4)/(n^2)$ dividing each term in the numerator by n^2
= lim_{n->inf} $(5 + 6/n - 4/n^2)$
= lim_{n->inf} (5) + lim_{n->inf} (6/n) - lim_{n->inf} $(4/n^2)$
= $5 + 0 + 0 = 5 \geq 0$.
Therefore, $5n^2+6n-4 \in O(n^2)$.

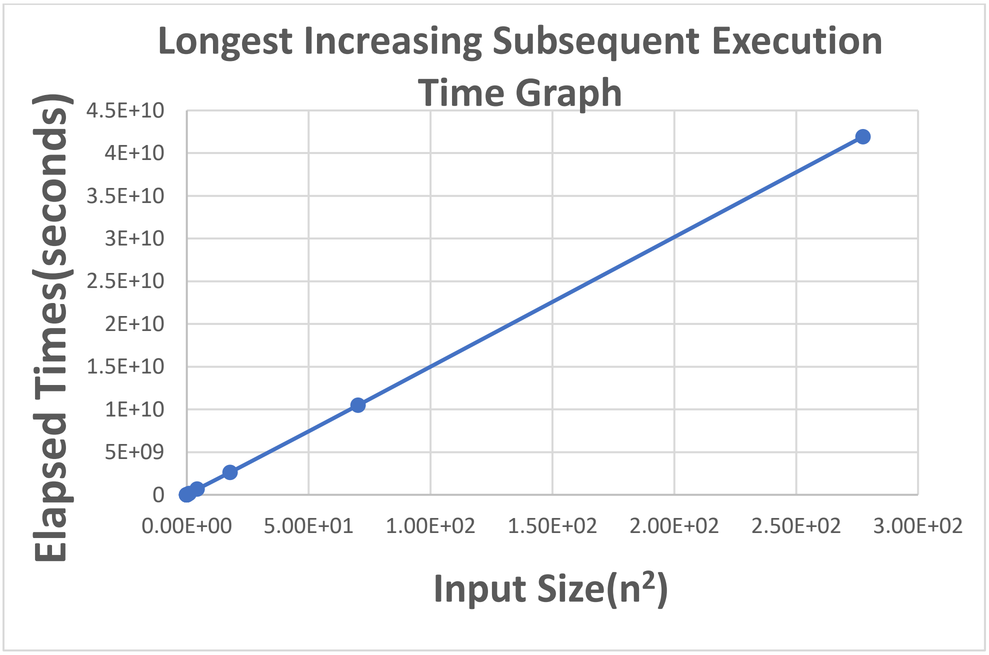
5. Scatter plot with the best-fit curve: I initially drew the graph below that shows the elapsed times(seconds) based on the sample sizes ranges from 20 to 40 produced by running my algorithm implementation.



This graph did not look a polynomial n^2 function. Thus, I drew another graph below that shows the elapsed times(seconds) based on the larger sample sizes ranges from 100 to 204800 produced by running my algorithm implementation. This new graph seems to represent the observed running time better.



6. Conclusion: Empirically-observed time efficiency data shown above seems polynomial n^2 function (quadratic function). To justify that I drew another graph using the squared of the same sample sizes and discovered that it created a linear function. I also ran the regression analysis on this set of data and got r squared as 0.925 which is very high. This also suggest that there is a linear association between the squared sample size and the elapsed times. Thus, I can easily say that the time efficiency data shown above is a quadratic function.



Therefore, the empirically-observed time efficiency data that I gathered is consistent with my mathematically-derived big-*O* efficiency class since the efficiency class of my algorithm according to my own mathematical analysis was O(n^2). They both show that my algorithm is a polynomial n^2 algorithm. This algorithm's speed noticeable faster than the powerset algorithm in Project 2 which has an efficiency of O(2^n . n). In fact, this polynomial function is 2^n/n times faster than the exponential function. This result was expected because polynomial functions are more efficient than the exponential functions.

Moreover, the findings from both empirical and mathematical data give us enough evidence to conclude that they are consistent with the first hypothesis given above: For large values of n, the mathematically-derived efficiency class of an algorithm accurately predicts the observed running time of an implementation of that algorithm.

My findings are also consistent with the second hypothesis. Therefore, based on my observations and findings, I can confidently say that polynomial-time dynamic programming algorithms are more efficient than exponential-time exhaustive search algorithms that solve the same problem.