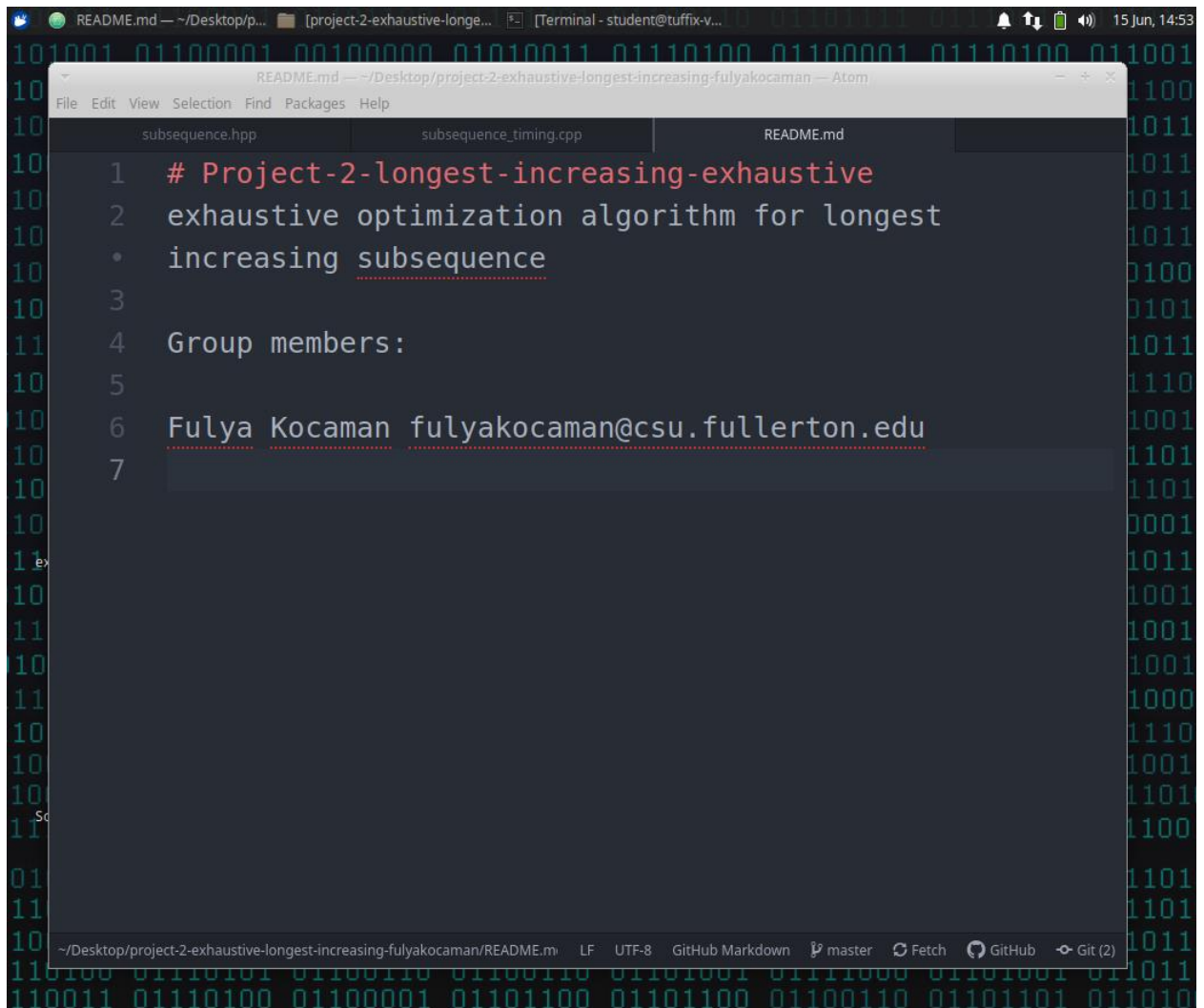


In this project I will be solving the longest increasing subsequent problem using the exhaustive optimization method. I will first write the pseudocode of the my algorithm from the C++ implementation, analyze it mathematically, measure its running time performance and prove that its time complexity is $O(n \cdot 2^n)$, 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. Algorithms with exponential or factorial running times are extremely slow, probably too slow to be of practical use.



```
1 # Project-2-longest-increasing-exhaustive
2 exhaustive optimization algorithm for longest
3   increasing subsequence
4
5 Group members:
6
7 Fulya Kocaman fulyakocaman@csu.fullerton.edu
```

1. Pseudocode:

The algorithm generates all possible subsequences of the array A and tests each subsequence on whether it is in increasing order. The longest such subsequence is a solution to the problem.

Input: An array A of integers.

Output: A longest increasing subsequence of the array A.

def longest_increasing_powerset(A)

n = size of A	1 step
best, increasing_best: vector of integers	
stack: vector of integers with size of n+1	
best_length = 0	1 step
k = 0	1 step
while(1)	2 ⁿ times(generating every power set)
if (stack[k] < n) then	1 step
stack[k + 1] = stack[k] + 1	3 steps
++k	1 step
else	
stack[k - 1]++	2 steps
k--	1 step
endif	
if (k == 0) then	1 step
break	
endif	
for each candidate i = 1 to k do	n times
push A[stack[i] - 1] in candidate	2 steps
endfor	
if (k > best_length) then	1 step
increasing = true	1 step
i = 0	1 step
if (k == 1) then	1 step
else	
while (increasing && i < (k - 1)) do	n-1 times
if (candidate[i] >= candidate[i + 1]) then	2 steps
increasing = false	1 step
endif	
i++	1 step
endwhile	
endif	

```

if (increasing) then
    best_length = k                                1 step
    for i = 0 to i < best_length do                 n times
        push candidate[i] in increasing_best        1 step
    endfor
endif
endif
endwhile

for i = (size of increasing_best - best_length) to i < size of increasing_best do    n times
    push increasing_best[i] in best                                                  1 step
endfor

return best

```

2. Step Count = $3 + 2^n[1 + \max(4, 3) + 1 + (n \cdot 2) + 1 + 2 + 1 + \max(0, (n-1) \cdot (2+1+1)) + 1 + (n \cdot 1)] + (n \cdot 1)$

$$= 3 + 2^n[1 + 4 + 1 + 2n + 4 + 4(n-1) + 1 + n] + n$$

$$= 3 + 2^n[7n + 7] + n$$

$$= 7n \cdot 2^n + 7 \cdot 2^n + n + 3$$

3. Time Complexity: I suggest that the big-O time complexity of this algorithm is $O(n \cdot 2^n)$ since $O(7n \cdot 2^n + 7 \cdot 2^n + n + 3) = O(7n \cdot 2^n) = O(n \cdot 2^n)$.

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 $7n \cdot 2^n + 7 \cdot 2^n + n + 3 \in O(n \cdot 2^n)$ if $\exists c > 0$ and $n_0 \geq 0$ such that $7n \cdot 2^n + 7 \cdot 2^n + n + 3 \leq c n \cdot 2^n \forall n \geq n_0$.

I will choose $c = 7 + 7 + 1 + 3 = 18$. Then, I have $7n \cdot 2^n + 7 \cdot 2^n + n + 3 \leq 18 n \cdot 2^n$.

Now, need to find n_0

$$18 n \cdot 2^n - 7n \cdot 2^n - 7 \cdot 2^n - n - 3 \geq 0$$

$$11 n \cdot 2^n - 7 \cdot 2^n - n - 3 \geq 0 \text{ satisfies } \forall n \geq 1.$$

Therefore, choosing $c = 18$ and $n_0 = 1$ shows that $7n \cdot 2^n + 7 \cdot 2^n + n + 3 \in O(n \cdot 2^n)$.

Prove by the limit theorem: I need to show that $7n \cdot 2^n + 7 \cdot 2^n + n + 3 \in O(n \cdot 2^n)$ using the limit theorem.

$$\lim_{n \rightarrow \infty} (7n \cdot 2^n + 7 \cdot 2^n + n + 3) / (n \cdot 2^n) \text{ dividing each term in the numerator by } n \cdot 2^n$$

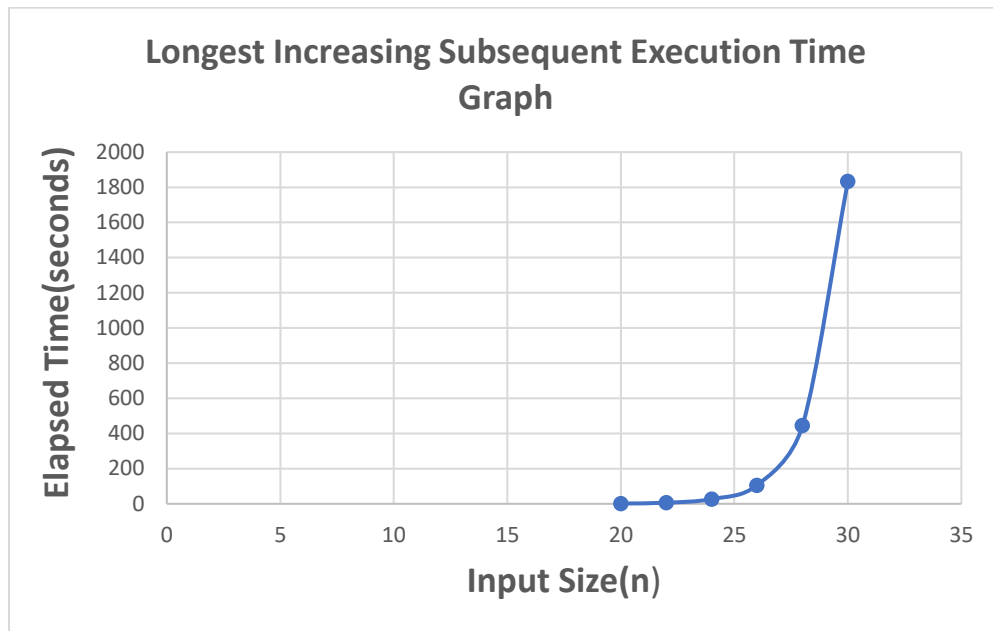
$$= \lim_{n \rightarrow \infty} (7 + 7/n + 1/2^n + 3/(n \cdot 2^n))$$

$$= \lim_{n \rightarrow \infty} (7) + \lim_{n \rightarrow \infty} (7/n) + \lim_{n \rightarrow \infty} (1/2^n) + \lim_{n \rightarrow \infty} (3/(n \cdot 2^n))$$

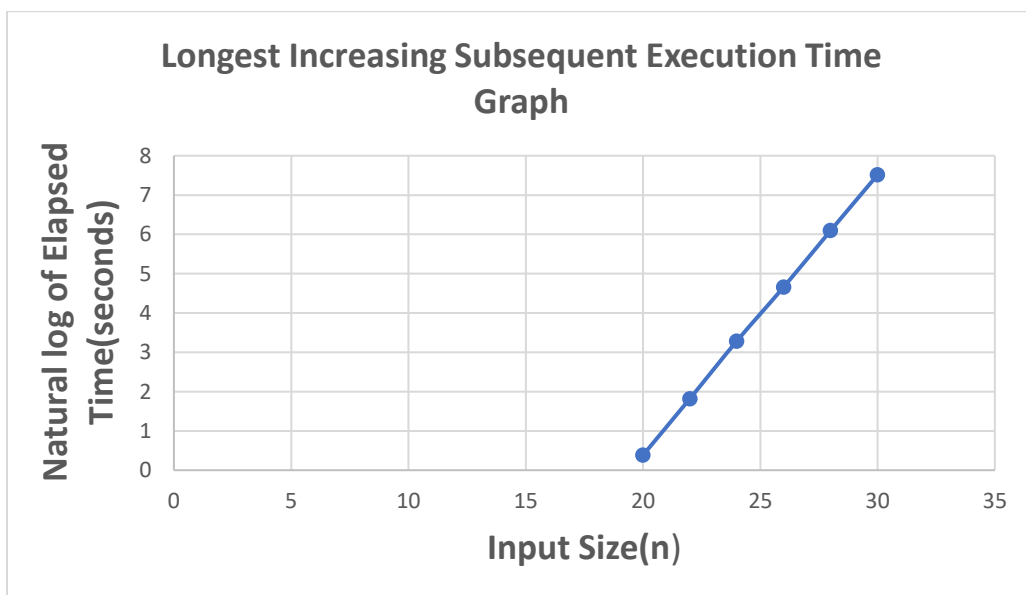
$$= 7 + 0 + 0 + 0 = 7 \geq 0.$$

Therefore, $7n \cdot 2^n + 7 \cdot 2^n + n + 3 \in O(n \cdot 2^n)$.

5. Scatter plot with the best-fit curve: The graph below shows the elapsed times(seconds) based on the sample sizes from 20 to 30 produced by running my algorithm implementation.



6. Conclusion: Empirically-observed time efficiency data shown above seems exponential. To justify that I drew another graph using the natural log of the same elapsed times and discovered that it created a linear function. Thus, the time efficiency data shown above is an exponential 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 $O(n \cdot 2^n)$, exponential. They both show that my algorithm is an exponential algorithm.

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. Even though I used my last sample size as 30 which may not seem that much of a large sample size, it took more than 30 minutes for my exponential algorithm to execute. Therefore, based on my observations and findings, I can confidently say that algorithms with exponential running times are extremely slow, probably too slow to be of practical use.