Ethan Cook
1163071

Question 1

Use the definition of “big Oh” to prove that n is not O(1/n)

if n is O(1/n) there are positive constants c and n_0 such that: $n \leq c * \frac{1}{n}$, however it is not possible to satisfy the inequality, as no matter what as n grows the left side will grow and the right side will infinitely shrink. Therefore n is not O(1/n)

Question 2

(both parts)
Let f(n) and g(n) and h(n) be non-negative functions such that f(n) is O(g(n)). Use the definition of “big Oh” to prove that f(n) × h(n) is O(g(n) × h(n)).

if f(n) is known to be O(g(n)), therefore we just have to prove that h(n) is O(g(n), then we will know that f(n) *h(n) is O(g(n)* h(n)).

if f(n) is O(g(n)) then there exists some constant c and n_0 where $f(n) \leq c * g(n)$, if we introduce h(n) to both sides of the inequality, the inequality will still hold, as we've made the same changes to both sides. Therefore f(n) *h(n) is O(g(n)* h(n)).

Question 3

```
Algorithm IsSymmetric(A, n)
i <- 0
while (i < n) do
        X = A[i]
        j <- i + 1
        flag <- false
        while (j < n) do
                Y = A[j]
                if !(X[0] == Y[1] and X[1] == Y[0]) then j <- j+1
                else
                        flag <- true
                        break
        if flag == true then return -1
return 1
```

a) Explain what the worst case for the algorithm

The worst case happens when pair A[0] is symmetrical with A[n], A[1] with A[n-1] and

A[i] with A[n-i], where i is at most N/2-1, then the last pair at the midpoint (n/2) isn't symmetrical with any pair. This would cause every pair to be compared with every other pair, finding their match at the last possible check while the midpoint fails to find any match.

b) Compute the time complexity of the algorithm in the best case
The best case for this algorithm is when the first pair is not symmetrical with any other pair. In this situation, the loop will only ever have i=0, and j will go through the array n-1 times, then the program will exit. So the time complexity will be $O(n)$ in the best case.

INSERT_PROOF_HERE

Problem Set 2:

Question 1: Generate a Fibonacci series of n elements (25 marks)

I've attached the python file with both approaches and the code used to test, time and generate the graphs in my assignment submission. I've also embedded them here for ease of access

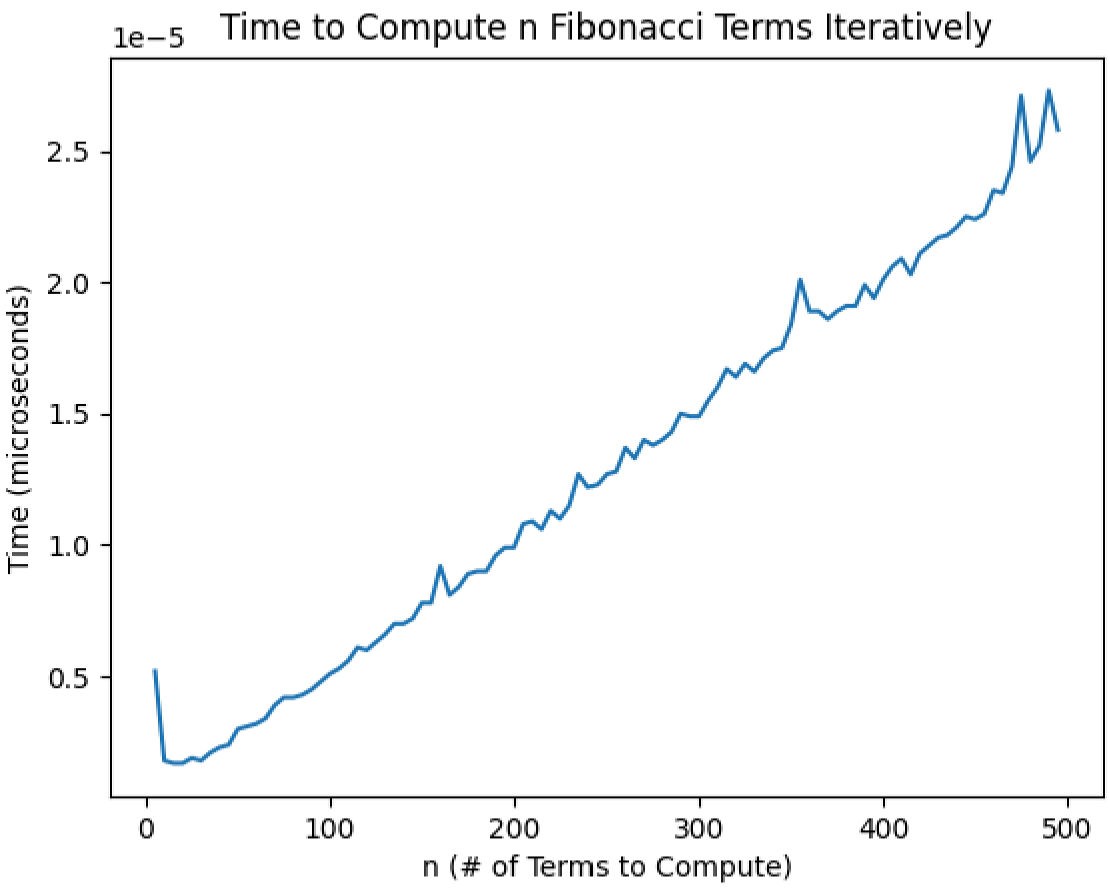
Approach 1: Iteration Based

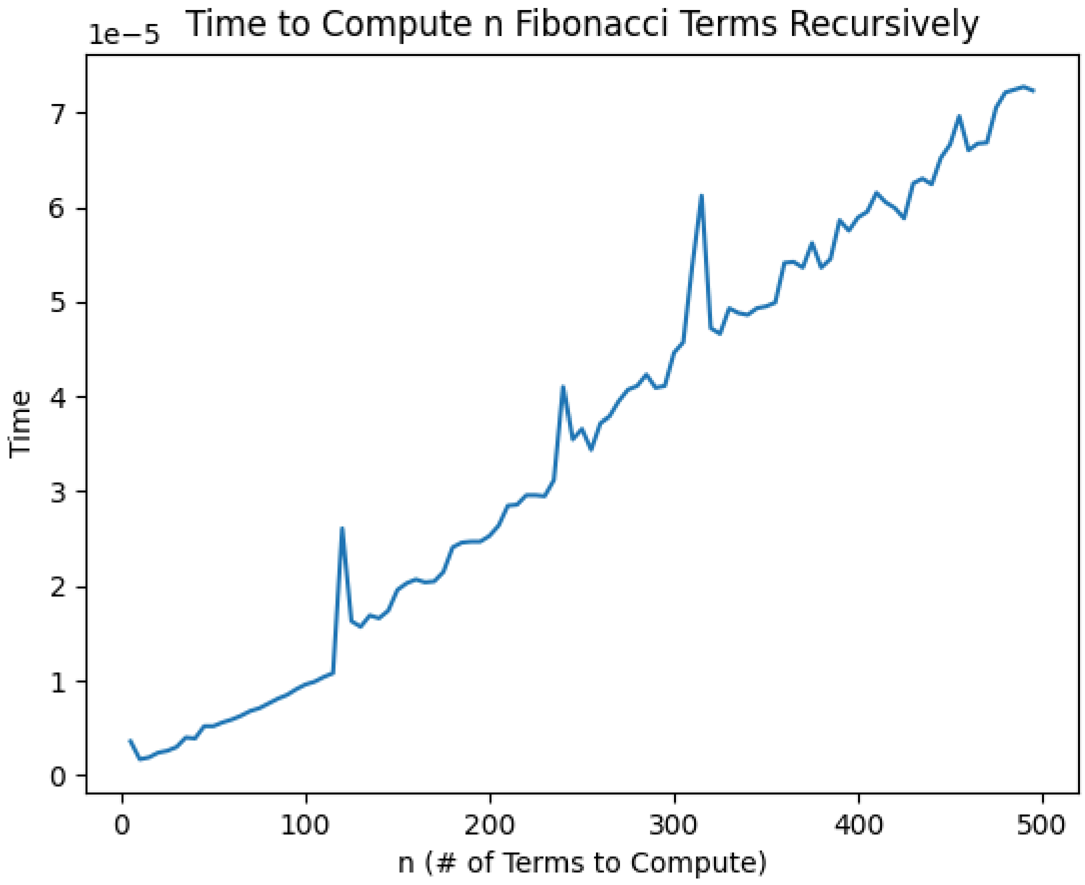
```python
def itter_fibb(n):
    range_ = range(3,n+1)
    arr = [0, 1]
    if n <= 2:
        return arr[:abs(n)]
    for i in range_:
        arr.append(arr[-1] + arr[-2])
    return arr
```

Approach 2: Recursive Function Based

```python
def recursive_fibb(n, x=2, arr=None):
    if arr is None:
        arr = [0, 1]
    if x == n:
        return arr
    arr.append(arr[-1] + arr[-2])
    recursive_fibb(n, x + 1, arr)
    return arr
```

Analyze both approaches in terms of time taken:


Time to Compute n Fibonacci Terms Iteratively
1e−5
2.5
2.0
1.5
1.0
0.5
Time (microseconds)
0
100
200
300
400
500
n (# of Terms to Compute)


Time to Compute n Fibonacci Terms Recursively
1e−5
7
6
5
4
3
2
1
0
Time
0
100
200
300
400
500
n (# of Terms to Compute)

These two graphs show the time to compute n terms of the Fibonacci sequence, using an iterative and recursive approach. The spikes are unavoidable due to CPU caching, other programs running and various uncontrollable reasons. However if we compare the two graphs we can see that the Iterative approach was far faster, peaking out at 2.5 microseconds compared to the recursive approaches 7.

Based on these tests, I think the iterative approach is superior in this case, although there might be other situations where a recursive approach is preferred.