Homework 5

P1

(a) see code in vscode

(b) see code in vscode(the program timed out for n = 50, it takes long to get the results for this value, but they eventually show up so we can also consider this value for plotting. This time out is stemming from the fact that Naive Search method recalculates values over and over again instead of storing them)

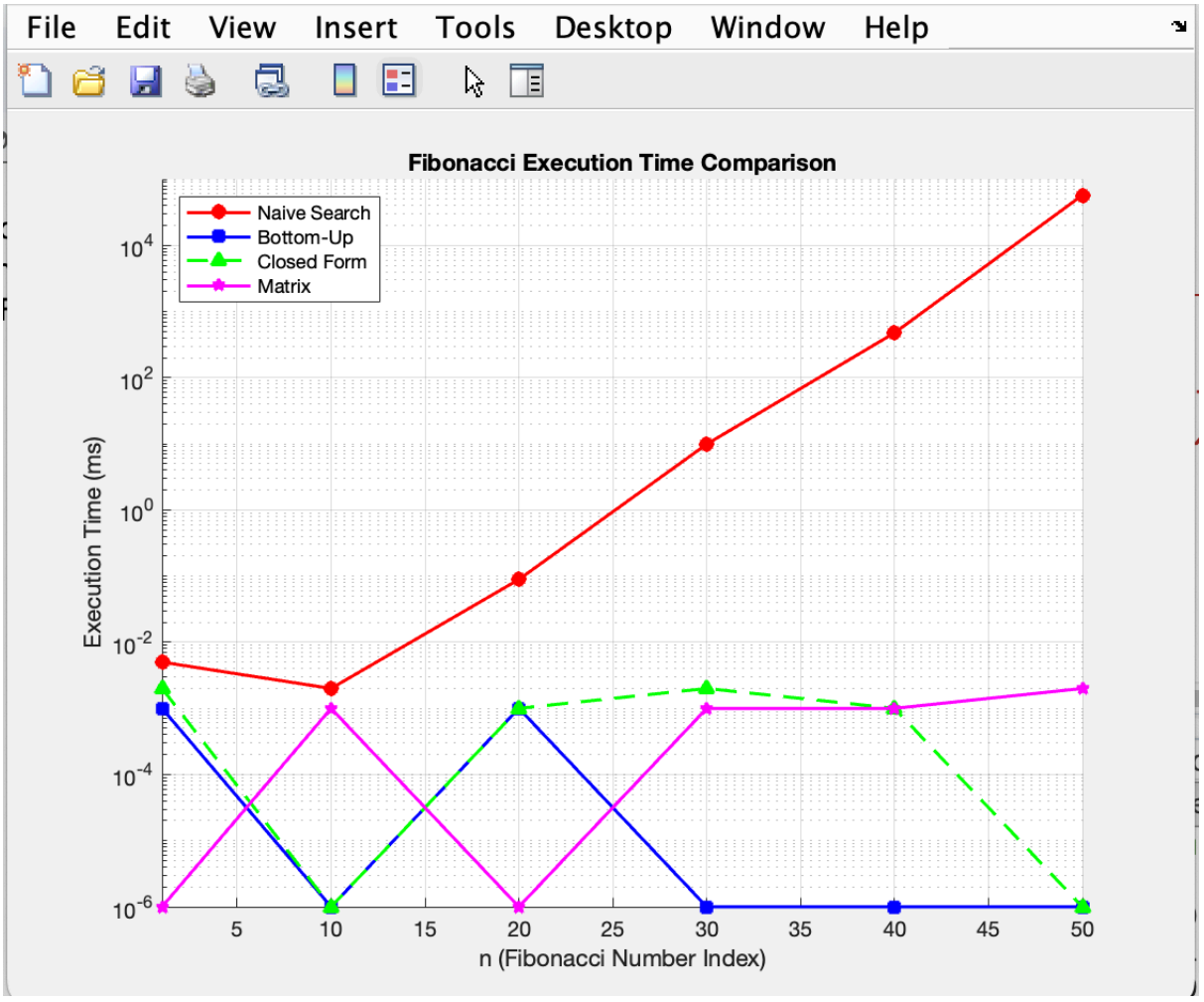
n	Naive Search	BottomUp	ClosedForm	MatrixRepresentation
0	0.009	0.003	0.003	0.003
1	0.003	0.002	0.004	0.004
2	0.003	0.002	0.003	0.003
3	0.002	0.003	0.003	0.003
4	0.002	0.002	0.003	0.003
5	0.002	0.002	0.003	0.003
6	0.003	0.020	0.005	0.003
8	0.003	0.003	0.003	0.003
10	0.003	0.002	0.006	0.003
13	0.007	0.003	0.003	0.003
16	0.015	0.002	0.004	0.014
20	0.091	0.003	0.002	0.002
25	0.995	0.003	0.003	0.003
32	24.592	0.004	0.002	0.003
40	466.097	0.002	0.002	0.003
50	55428.1	0.002	0.002	0.003

(c)Yes, for the same n, all of the methods always return the same Fibonacci number. Naive Recursive, Bottom-up, Closed Form and Matrix Exponentiation always have the same output n as long as the implementation was correct. The reason is that all of these methods were designed to calculate the same Fibonacci sequence, and even though they use different approaches, the mathematical concepts behind them are exactly the same. The fibonacci sequence is well-defined, and as long as the correct formulas are being used, the same correct result will be guaranteed.

The naive Recursive Method is using directly the definition of the Fibonacci sequence:

F(n) = F(n-1) + F(n-1);(it might not be the best choice performance wise because it has an exponential time complexity because of the numerous recursive calls).
Bottom-Up Approach computes iteratively the Fibonacci numbers from the base case upwards.

Closed Form: We used an approximation of the Binet's formula $(\phi^{n}/\sqrt{5})$ where $\phi = (1 + \sqrt{5})/2)$. This method provides an exact result, assuming that floating-point precision does not interfere(this might be a concern for very large n, but s should still return the correct Fibonacci number with a reasonable range).
Matrix Exponentiation: this method uses the fact that Fibonacci numbers can be computed efficiently using matrix exponentiation. The n th Fibonacci number can be derived from the matrix raised to the power of n (The searched for element(F(n)) is at position [0][1]).
(d)



The plot may not be accurate in some cases, and we will discuss those situations while breaking down each approach separately:
1.Naive Search(red line)

- ☐ this method has $O(2^{n})$ time complexity, meaning it grows exponentially with n.
- ☐ for small values of n execution time is negligible
- ☐ as n increases, the number of recursive calls escalates, leading to an abrupt rise in execution time

This had an expected behaviour

2.Bottom-up(blue line):

- ☐ this method has an O(n) time complexity, meaning that the execution time grows linearly with n
- ☐ in practice -> much faster than the naive approach as it avoids redundant computations
- ☐ we can observe that it remains really low on the graph, even for large values of n

It sometimes has an unexpected behaviour, especially when n is small because clock()'s precision does not capture its speed accuracy, which might be the cause of inconsistent plotting. clock() measures in milliseconds, which may not be enough for extremely fast calculations

3. Closed-Form formula(green line)

- ☐ this approach is known to have O(1) complexity, making it theoretically the fastest
- ☐ however, due to floating-point precision limits, results may become slightly inaccurate for large n
- ☐ floating-point operations, such as (pow() and sqrt(0) can introduce small computational errors, leading to variations in the execution time

We encounter unexpected behaviour for large values for n because of the floating-point operations that are not always executed in constant time because of hardware optimizations(caching and CPU optimizations may cause the function to run inconsistently fast).

4.Matrix Exponentiation(Pink line)

- ☐ this method has O(log n) complexity, making it highly efficient for a large value of n
- ☐ due to overhead of matrix multiplications, it may be slower than expected for a small n. Unlike the Closed-Form method, it does not suffer from precision errors but may not always outperform the Bottom-Up approach at small values of n.

We encounter unexpected behaviour for small values of n, because matrix multiplication introduces constant-time overhead, making it slower than expected for very small n
For a large n, it outperforms the naive significantly but remains close to the closed-form method.

ADS Homework 5

P5.2

(a) For a brute-force multiplication (the basics of multiplication) of 2 really large n-bits integers, we can look over the classical long multiplication algorithm, which has the following mechanism behind it. → we multiply each bit of a with each of b

→ then, we shift and add the intermediate results accordingly (intermediate results as in the partial products that we get at each multiplication process before summing them up)

Complexity Analysis:

- Each n-bit number has n digits (in binary representation) we multiply each bit of a with each bit of b, leading to $n \times n = n^2$ bit-wise multiplications. Each of these multiplications of single bits takes $O(1)$ time (we have to keep in mind that bit shifts and additions take at most $O(n)$ per term).

Since we perform $\Theta(n^2)$ multiplications and n additions we at most $\Theta(n^2)$ in total, the overall time-complexity of brute-force multiplication is $O(n^2)$ (as the number of bits n increases $\Rightarrow$ the runtime grows quadratically).

1(c) Supposing that a and b are both n-bit numbers, we can divide each of them in two parts (where n is a power of 2):

$a = a_1 \cdot 2^{n/2} + a_0$ } where a_1, b_1 → most significant bits
$b = b_1 \cdot 2^{n/2} + b_0$ } a_0, b_0 → least significant bits;

$$a \cdot b = (a_1 \cdot 2^{n/2} + a_0)(b_1 2^{n/2} + b_0) =$$
$$= a_1 b_1 2^n + a_1 2^{n/2} b_0 + a_0 b_1 2^{n/2} + a_0 b_0$$
$$= a_1 b_1 2^n + (a_1 b_0 + a_0 b_1) 2^{n/2} + a_0 b_0$$

we have 4 multiplications of $n/2$ - bit numbers: $a_1 b_1$, $a_1 b_0$, $a_0 b_1$, $a_0 b_0$ - combining those will only require shifts and additions, which will take $\Theta(n)$ time.

(2) Recalling that above we have splitted the problem into 2 subproblems and made 4 recursive calls on number of size $(n/2)$:

$$T(n) = 4T(n/2) + \Theta(n)$$

(d) Recursion tree method:

We can begin by expanding the recurrence $T(n) = 4T(n/2) + \Theta(n)$:

$$T(n) = 4[4T(n/4) + \Theta(n/2)] + \Theta(n)$$
$$T(n) = 16\,T(n/4) + 4\Theta(n/2) + \Theta(n)$$

This can be generalized to:

$$T(n) = 4^k T(n/2^k) + \Theta(kn)$$

Stop when the subproblem reaches 1 (when $n/2^k = 1$), so $k = \log_2(n)$.

substituting $a = \log_2(n)$ into the equation:

$T(n) = 4^{\log_2(n)} \cdot T(1) + \Theta(n \log_2(n))$

Since $4^{\log_2(n)} = n^2$ and $T(1) = \Theta(1)$, we get:

$T(n) = \Theta(n^2) + \Theta(n \log_2(n))$

Thus, the final result is dominated by the n^2 term

$T(n) = \Theta(n^2)$

(c) $T(n) = 4T(n/2) + \Theta(n)$;

$a = 4$ (number of subproblems)

$b = 2$ (factor by which the problem size is reduced in each subproblem)

$d = 1$

$n^{\log_b a}$? $f(n)$

$n^{\log_2 4}$? $f(n)$

n^2 ? $\Theta(n)$

n^2 is polynomially larger $\Theta(n) \Rightarrow f(n) = O(n^{\log_b a - \epsilon})$

$\Rightarrow T(n) = \Theta(n^2)$