ADS Homework 5

Name - Tewodros Adane
Email - tadane@jacobs-university.de

Problem 4.1 – Fibonacci Numbers

a) The implementations of the different methods of generating Fibonacci numbers in python.

Naive Recursion:

```
def naiveFibonacci(n):
    if n < 2:
        return n
    else:
        return naiveFibonacci(n - 1) + naiveFibonacci(n - 2)
```

Bottom Up:

```
def bottomUpFibonacci(n):
    if n == 0:
        return 0
    num1 = 0
    num2 = 1
    for i in range(1, n):
        new_num = num1 + num2
        num1 = num2
        num2 = new_num
    return num2
```

Closed Form:

```
def closedFormFibonacci(n):
    a = (1 + math.sqrt(5)) / 2
    f = power(a, n) / math.sqrt(5)
    return math.ceil(f)

def power(a, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return power(a, n//2) * power(a, n//2)
    else:
        return power(a, n//2) * power(a, n//2) * a
```

Matrix Representation:

```
def matrixFibonacci(n):
    a = [[1, 1], [1, 0]]
    if n == 0:
        return 0

    powerMatrix(a, n)
    return a[0][1]

def powerMatrix(a, n):
    if n < 2:
        return

    b = [[1, 1], [1, 0]]
    powerMatrix(a, n // 2)
    multiplyMatrices(a, a)

    if (n % 2 != 0):
        multiplyMatrices(a, b)

def multiplyMatrices(a, b):
    c00 = a[0][0]*b[0][0] + a[0][1]*b[1][0]
    c01 = a[0][0]*b[0][1] + a[0][1]*b[1][1]
    c10 = a[1][0]*b[0][0] + a[1][1]*b[1][0]
    c11 = a[1][0]*b[0][1] + a[1][1]*b[1][1]

    a[0][0] = c00
    a[0][1] = c01
    a[1][0] = c10
    a[1][1] = c11
```

b) The time taken by the different algorithms for finding the nth Fibonacci number is shown by the following tables. Execution was stopped when the time exceeded 100 seconds.

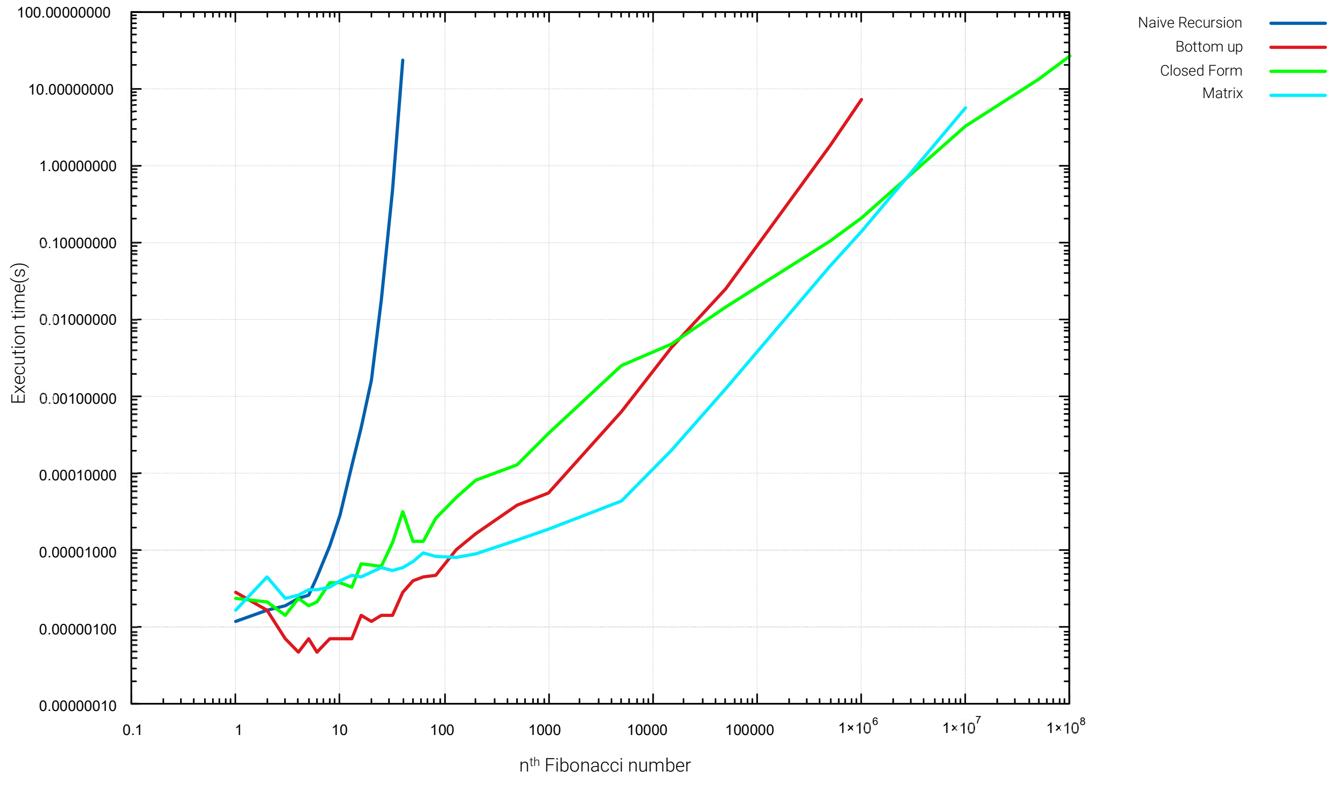
N	Execution time (s)			
	Naive recursion	Bottom up	Closed Form	Matrix
0	1.67E-06	1.67E-06	5.25E-06	1.43E-06
1	1.19E-06	2.86E-06	2.38E-06	1.67E-06
2	1.67E-06	1.67E-06	2.15E-06	4.53E-06
3	1.91E-06	7.15E-07	1.43E-06	2.38E-06
4	2.38E-06	4.77E-07	2.38E-06	2.62E-06
5	2.62E-06	7.15E-07	1.91E-06	3.10E-06
6	4.53E-06	4.77E-07	2.15E-06	3.10E-06
8	1.14E-05	7.15E-07	3.81E-06	3.34E-06
10	2.88E-05	7.15E-07	3.81E-06	4.05E-06
13	0.0001266	7.15E-07	3.34E-06	4.77E-06
16	0.000392199	1.43E-06	6.68E-06	4.53E-06
20	0.001623392	1.19E-06	6.44E-06	5.25E-06
25	0.017766476	1.43E-06	6.20E-06	5.96E-06
32	0.474479914	1.43E-06	1.26E-05	5.48E-06
40	23.07508683	2.86E-06	3.19E-05	5.96E-06
50	Exceeded time limit	4.05E-06	1.31E-05	7.15E-06
63	Exceeded time limit	4.53E-06	1.31E-05	9.30E-06
83	Exceeded time limit	4.77E-06	2.62E-05	8.34E-06
130	Exceeded time limit	1.03E-05	4.86E-05	8.11E-06
200	Exceeded time limit	1.65E-05	8.18E-05	9.06E-06
500	Exceeded time limit	3.86E-05	0.000129461	1.36E-05
1000	Exceeded time limit	5.56E-05	0.000331163	1.88E-05
5000	Exceeded time limit	0.000632048	0.002519846	4.39E-05
15000	Exceeded time limit	0.004290819	0.004788637	0.000198364
50000	Exceeded time limit	0.024843454	0.014472961	0.001255751
500000	Exceeded time limit	1.796675205	0.103965044	0.049222946
1000000	Exceeded time limit	7.103663445	0.206573963	0.138466597
10000000	Exceeded time limit	Exceeded time limit	3.210908175	5.586149454
50000000	Exceeded time limit	Exceeded time limit	13.03990984	Exceeded time limit
100000000	Exceeded time limit	Exceeded time limit	26.14154172	Exceeded time limit

c) By choosing **n** to be 8 and calculating the 9th Fibonacci number with the different algorithms we get:

```
Naive Recursion = 21
Bottom Up       = 21
Closed Form     = 22
Matrix          = 21
```

This shows that the closed form doesn't always return the exact value. This is because of the imprecision in the way floating point numbers are calculated on digital computers. We can further see this when we make the closed form function return without rounding the float value to the next integer. It returns 21.009519494, which is closer to 21 rather than 22.

d) The following is the graph of the execution times for the functions. The x-axis and y-axis are in a logarithmic scale.



From the graph above we can observe that naive recursion is the fastest method for values of n that are less than 2 and the slowest after values greater than 6. Naive recursion method is very slow for values of n larger than the 30s. Closed form method beats all the other functions for extremely large values of n even though it doesn't always return the correct value. The matrix method is faster than bottom up method for values of n greater than 100.

Problem 4.2 – Divide & Conquer and Solving Recurrences

a) The brute force algorithm for multiplying two numbers **a** and **b**, which are both **n** bits long, can be approached like the way we multiply numbers on paper. By iterating through each bit of **b** from right to left and multiplying each digit with all the **n** bits of **a**, with each iteration shifting the product to the left by one more bit than the last shift. This step will take $\Theta(n^2)$ time this is because iterating through **n** bits of **b** and multiplying **n** times for each bit in **a** has a complexity of $n * n$. Then, summing up the products of the previous step takes $\Theta(n)$ because addition and subtraction take linear time.

$$Therefore, \qquad T(n) = \Theta(n^2)$$

b) For the divide and conquer algorithm we can split the digits in the middle and reduce it into smaller problems. Let's take two numbers a and b that are n bits long(n is a power of 2).

a can be split into a_{Left} and a_{Right}, b can also be split the same way.

$$a = a_{left} * 2^{\frac{n}{2}} + a_{right}$$

$$b = b_{left} * 2^{\frac{n}{2}} + b_{right}$$

When the numbers cannot be divided any further, we can multiply them.

$$a * b = a_{left}b_{left} * 2^{\frac{n}{2}} * 2^{\frac{n}{2}} + a_{left}b_{right} * 2^{\frac{n}{2}} + a_{right}b_{left} * 2^{\frac{n}{2}} + a_{right}b_{right}$$

$$a * b = a_{left}b_{left} * 2^n + \left(a_{left}b_{right} + a_{right}b_{left}\right) * 2^{\frac{n}{2}} + a_{right}b_{right}$$

From this result there's still have 4 multiplications which still has $\Theta(n^2)$, so the number of multiplications should be reduced. The coefficient of $2^{\frac{n}{2}}$ can be replaced so that it will only take 1 multiplication instead of 2.

$$(a_{left}b_{right} + a_{right}b_{left}) = (a_{left} + a_{right}) * (b_{left} + b_{right}) - a_{left}b_{left} - a_{right}b_{right}$$

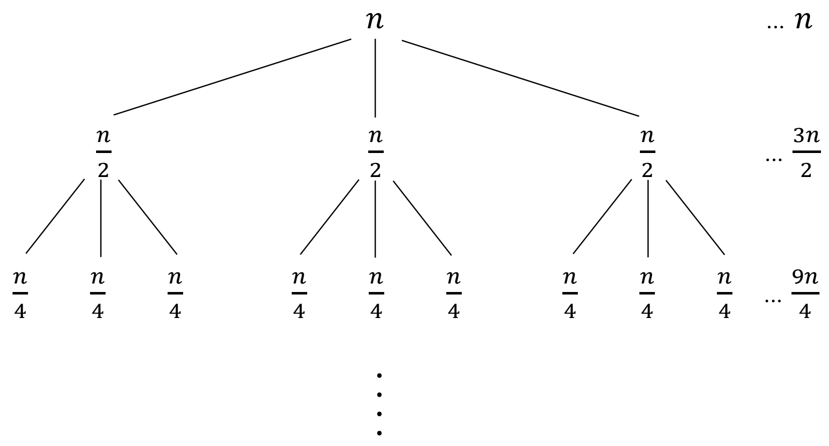
Even though this seems like we have 3 multiplications the last two terms don't need to be calculated because they're already done in the previous step. So only one multiplication is done.

$$a * b = a_{left}b_{left} * 2^n + \left((a_{left} + a_{right}) * (b_{left} + b_{right}) - a_{left}b_{left} - a_{right}b_{right}\right) * 2^{\frac{n}{2}} + a_{right}b_{right}$$

c) We divide the problem in $\frac{n}{2}$ every time and we have three multiplications. We also have two bit-shiftings, four additions, and two subtractions which are the combination steps take $\Theta(n)$.

$$T(n) = 3T(n/2) + \Theta(n)$$

d) Solving the recurrence using recursion tree method:



n
$\ldots n$
$\frac{n}{2}$ $\frac{n}{2}$ $\frac{n}{2}$
$\ldots \frac{3n}{2}$
$\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$
$\ldots \frac{9n}{4}$

The height of this tree is going to be $\log(n)$ because were dividing the problem into 2 on each step.
The cost at height h is given by:

$$cost = n\left(\frac{3}{2}\right)^h$$

And the complexity:

$$complexity = O\left(n * \left(\frac{3}{2}\right)^{lo\,g(n)}\right)$$

$$complexity = O(3^{\log_2 n})$$

$$complexity = O(n^{\log_2 3}) = O(n^{1.585})$$

e) Solving the recurrence using master method:

$$T(n) = 3T(n/2) + \Theta(n)$$

$$a = 3, b = 2, \ \log_2 3 = 1.585$$

$$f(n) = O(n^{1.585-\varepsilon})$$

$$for\ \varepsilon = 0.585,$$

$$f(n) = O(n)$$

$$Therefore,\ T(n) \in \Theta(\mathrm{n}^{1.585})$$