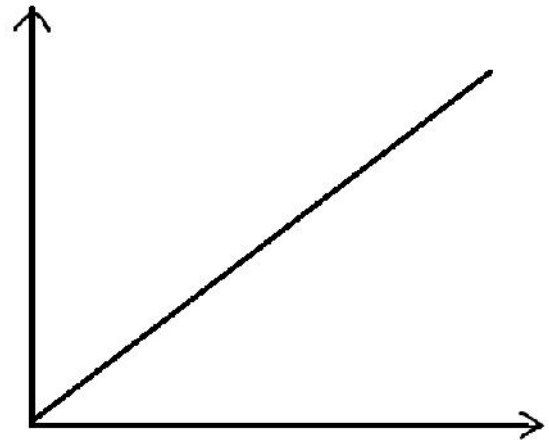
Team Gaspy

Emin Arakelian
Mike Galczynski

CE 290I Assignment 2

Fibonacci Iterative Linear Time Algorithm



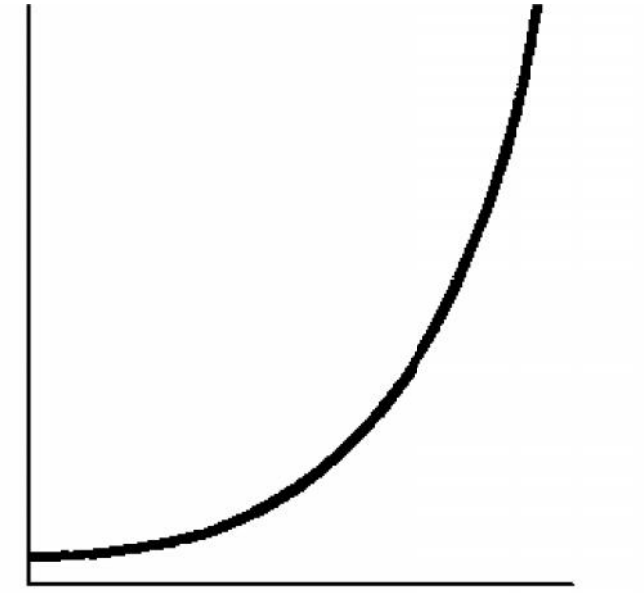
```
1 Fibonacci Numbers
[0]
2 Fibonacci Numbers
[0, 1]
3 Fibonacci Numbers
[0, 1, 1]
4 Fibonacci Numbers
[0, 1, 1, 2]
5 Fibonacci Numbers
[0, 1, 1, 2, 3]
6 Fibonacci Numbers
[0, 1, 1, 2, 3, 5]
7 Fibonacci Numbers
[0, 1, 1, 2, 3, 5, 8]
```

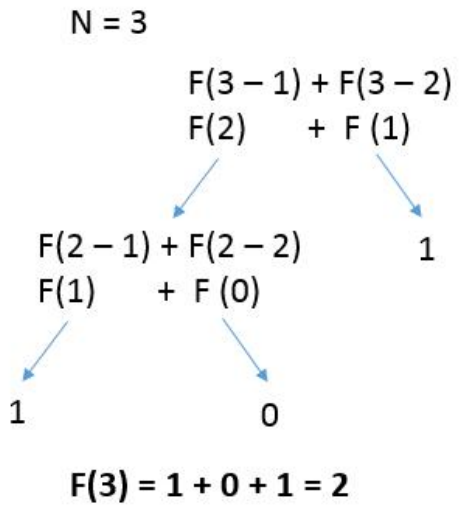
The first algorithm that we created was a linear time Fibonacci algorithm. The operation of this algorithm is relatively simple. First a list / array of the first two Fibonacci numbers is created: [0,1]. To calculate the nth Fibonacci number, the program adds the last number of the list, and the next to last number of the list and places the result of this addition at the end of the list. An example of this function is shown above next to the linear graph.

The reason that this program will result in a linear time complexity algorithm, O(n), is because it must essentially complete a successive number of addition operations to compute the nth Fibonacci number; the higher the value of n to be computed, the higher the number of additions that must be completed. An excerpt from the DLX code that we wrote to run this algorithm can be found below.

```
ADDI 1 0 1
STW 1 1 0
ADDI 1 0 1
STW 1 2 0

LDW 1 1 0
LDW 2 2 0

ADD 1 1 2
STW 1 3 0
ADD 2 1 2
STW 2 4 0

ADD 1 1 2
STW 1 5 0
ADD 2 1 2
STW 2 6 0

ADD 1 1 2
STW 1 7 0
ADD 2 1 2
STW 2 8 0

ADD 1 1 2
STW 1 9 0
ADD 2 1 2
STW 2 10 0

END
```

Fibonacci Recursive Exponential Time Algorithm





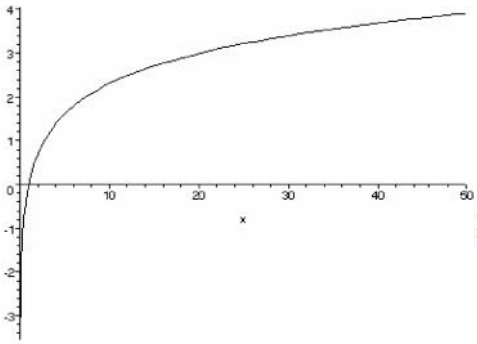
The second algorithm we tried to create was a recursive exponential time Fibonacci algorithm. The code we developed for this in python is as follows:

```
def F(n):
    if n == 0: return 0
    elif n == 1: return 1
    else: return F(n-1)+F(n-2)
```

Essentially this code will continually call on itself to calculate the nth Fibonacci number. An example of this process is shown above for n = 3, i.e. the 3rd Fibonacci number. As one might imagine, every increase in the value of n, causes an increasing number of branches and calculations to be created. This behavior will result in an algorithm that takes increasingly longer and longer to compute the value of n (an exponential time complexity algorithm).

*Unfortunately this algorithm is still in the debugging phase as it has proven extremely difficult.

Fibonacci Recursive Logarithmic Algorithm



$$F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n \sqrt{5}}.$$

The third algorithm we tried to create was a recursive logarithmic time Fibonacci algorithm. The code we developed for this in python is as follows:

```
def F(n):
    fib = int(((1+(5**0.5))**n-(1-(5**0.5))**n)/(2**n*(5**0.5)))
    return fib
```

The code mirrors the equation shown above, and calculates the nth Fibonacci number using a formula that is based off of the value n. Because the calculation of Fn requires only a single calculation to compute the nth Fibonacci number, the time complexity of the algorithm will be logarithmic. As n gets larger, the algorithm will only take marginally longer to compute, because of the extra digits and higher order numbers that must be calculated. We attempted to make this code in the DLX language, which is shown below for n = 2:

```
ADDI 1 0 3.2360679775
ADDI 2 0 2
ADDI 3 0 -1.2360679775
ADDI 4 0 2.2360679775

MULI 5 1 10.472135955
MULI 6 3 -1.527864045
SUB 7 5 6
MULI 5 2 4
MUL 6 5 4
DIV 5 7 6
Check 3 1
```

*Unfortunately we have not yet been able to implement this algorithm into DLX code as we realized that our DLX emulator will only accept integer values, and thus our equation which uses floating points cannot work. We are continuing to work on another algorithm. We will try to use a recursive matrix method instead.