Homework 3 : Algorithms and Data Structures

Digdarshan Kunwar

February 2019

Problem 3.1 Fibonacci Numbers

!! NOTE !!
Run file named main.py from the zip file.
In the zip file within the folder **Plots** are the testcases graph of different algorithms.

(a) The various implementation of finding the Fibonacci are given below and are also in zip folder:

```
import math
from math import sqrt

#Recursive or the naive appoach of finding fibonacci number
def frecursive(number):
    if number==0:
        return 0
    if number==1:
        return 1
    return frecursive(number-1)+frecursive(number-2)


#This is the bottom up approach of finding the nth fibonacci number
def fbottomUp (number):
    if number==0:
        return 0
    if number==1:
        return 1
    n1=0
    n2=1
    for _ in range(1,number):
        n2=n1+n2
        n1=n2-n1
    return n2

#Closed form Implementation
def fformula(number):
    return (((1+math.sqrt(5))/2)**number / math.sqrt(5))

#The Matrix Implementation

#Fibonacci Numbers using matrix multiplication

class Matrix:
    def __init__(self,a,b,c,d):
        self.a=a
```

```
            self.b=b
            self.c=c
            self.d=d
        def multiply(self,matb):
            new=Matrix(0,0,0,0)
            new.a=self.a*matb.a +self.b*matb.c
            new.b=self.a*matb.b +self.b*matb.d
            new.c=self.c*matb.a +self.d*matb.c
            new.d=self.c*matb.b +self.d*matb.d
            return new

    def mul(n):
        if n==1:
            return Matrix(1,1,1,0)
        return Matrix(1,1,1,0).multiply(mul(n-1))

    def fmatrix(n):
        if(n==0):
            return 0
        return mul(n).b
```

(b) The graph are attached in the zip file.
The combined graph is named plot_all.png
The table is in testfile.txt and in Table Data.png

(c) Here all the other method except the closed form give us the same fibonacci number. This is because for the large nth Fibonacci number the value (double/float) that is returned from the sqrt(x) function has a certain precision but always fails to meet the perfect accuracy as to gain perfect accuracy the precision should be as large as possible.

(d) Here the recursive method for larger n takes the highest amount of time for any nth fibonacci number to be calculated.
The formula method has a straight line which implies it has it has constant time.As for the bottom up approach we have linear line.
The graph of different methods are in Plots directory.

Problem 3.2

Divide Conquer and Solving Recurrences

(a) Here we know that,
Addition, subtraction, and bit shifting can be done in linear time that is $\Theta(n)$
So according to the brute force implementation of multiplication it means that a number **a** with n bits when gets multiplied with another number **b** with n bits then we have n bit shift that occurs.After every bit shift there is the multiplication of the bit for n times with every n bits in **a**.
Thus there is a loop for going through every bit of number A for every n bit of number B.
Thus we have

$$
\begin{aligned}
T(n) &= n\Theta(n) + n\Theta(n) \\
&= 2n\Theta(n) \\
&= \Theta(n^2)
\end{aligned}
$$

(b) Let a number A and B be n bits.
According to Karatsuba algorithm

A $= \boxed{\mathrm{a}_L \mid \mathrm{a}_R} = a_L * 2^{\frac{n}{2}} + a_R$

B $= \boxed{\mathrm{b}_L \mid \mathrm{b}_R} = b_L * 2^{\frac{n}{2}} + b_R$

Then we know that,

$$
\begin{aligned}
A.B &= (a_L * 2^{\frac{n}{2}} + a_R) * (b_L * 2^{\frac{n}{2}} + b_R) \\
&= (a_L b_L * 2^n + 2^{\frac{n}{2}} * (a_R b_L + a_L b_R) + a_R b_R)
\end{aligned}
$$

Here we have four multiplication of $n/2$ two bit numbers and 3 addition and 2 shift operation for the multiplication by 2^n *or* $2^{n/2}$
So we have the cost as :

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n)$$

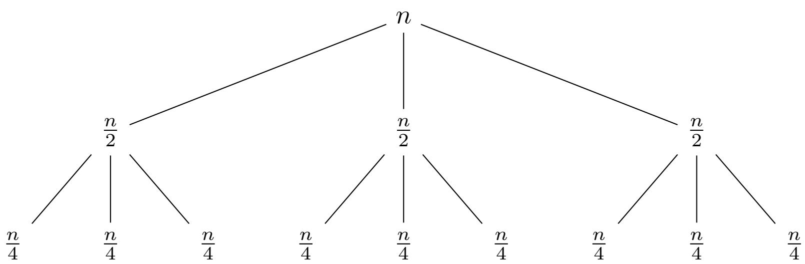
The time complexity can be reduced by following the algorithm.

```
def multiply(A,B):
    n=max(its in A,bits in B)
    if (n==1):
        return x*y

    a_L,a_R = left[n/2] bits of A,right[n/2] bits of A
    b_L,b_R = left[n/2] bits of B,right[n/2] bits of B

    #Now multiplying
    M1=multiply(a_L,b_L)
    M2=multiply(a_R,b_R)
    M3=multiply((a_L+b_R),(b_L+a_R))

    return M1*2^n+(M3-M2-M1)*2^(n/2)+M2
```

(c) As the multiply function is called three times within the function with half of the n as the new parameter.
Looking at the algorithm above we can imply that :

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

(d) Here we have the tree,



So from the tree:

$$h = log\ _2(n)$$

where h is the height of the recursive tree.
Here at any level k the cost is $\left(\frac{3}{2}\right)^k.O(n)$
At the bottom of the tree,
We have

$$\left(\frac{n}{2^k}\right) = 1 \quad where\ k = h$$

Then,

$$\begin{aligned} Cost &= \left(\frac{3}{2}\right)^{log\ _2(n)} O(n) \\ &= O\left(\left(\frac{3}{2}\right)^{log\ _2(n)} * n\right) \\ &= O\left(3^{log\ _2(n)}\right) \\ &\texttt{This can be rewritten as} \\ &= O\left(n^{log\ _2(3)}\right) \end{aligned}$$

(e) Here using the master method:

T(n)=3$(\frac{n}{2}) + O(n)$
According to the master theorem : a=3 b=2

f(n)=$O(n^{\log_2 3} - \epsilon)$ $\epsilon = \log_2 3 - 1 \implies \epsilon = 0.5849$

Therefore, $O(n^{\log_2 3})$