3/1/2019

Homework 3

Algorithms and Data Structure

Mario Alberto hernandez salamanca
JACOBS UNIVERSITY

1.Fibonacci Algorithms

1.1Implementation

naive recursive

```
unsigned int fibonacci(unsigned int n){
  if(n<2){
          return n;
  }
  else{
          return fibonacci(n-1) + fibonacci(n-2);
  }
}
```

bottom up

```
unsigned int fibonacci(unsigned int n)
{
  unsigned int f[n+1];
  int i;
  f[0] = 0;   f[1] = 1;
  for (i = 2; i <= n; i++) {
      f[i] = f[i-1] + f[i-2];
  }

  return f[n];
}
```

closed form

```
const long  double Phi = 1.6180339887498948;
const long double rooo_of_5 = 2.2360679775;
unsigned int fibonacci(unsigned int n){
  long double result;
  result = pow(Phi,n) / rooo_of_5;
  return round(result);
}
```

matrix representation

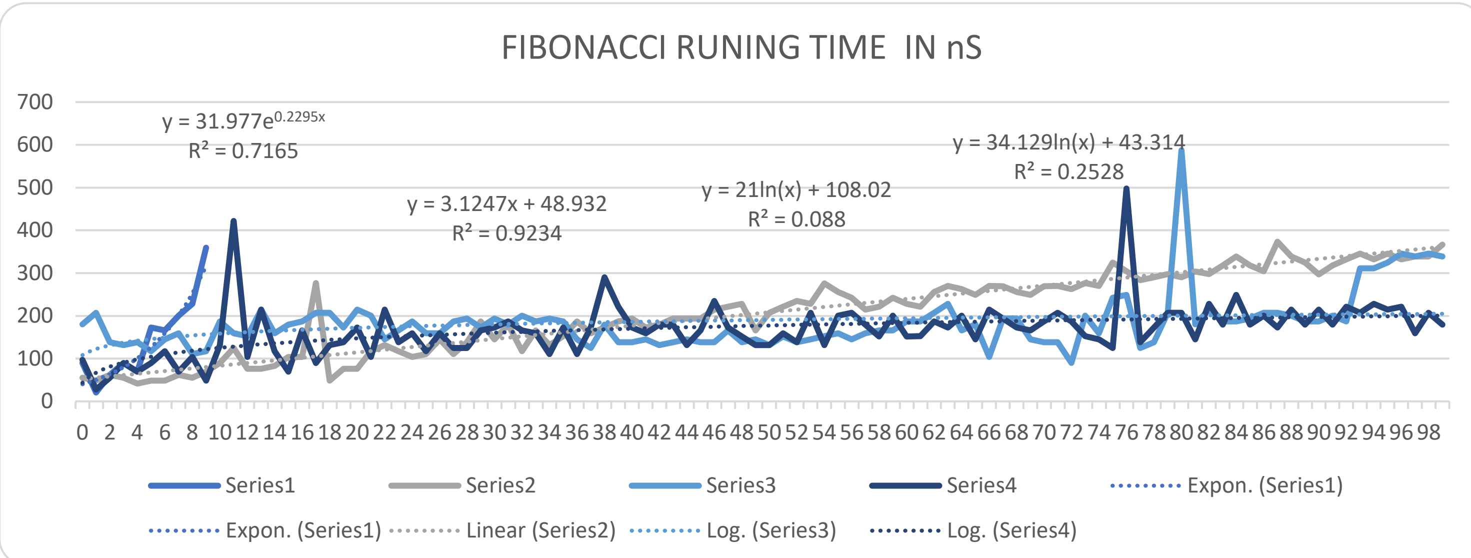
```
//function multiplies two matrixes of 2x2
void multiplicacion(unsigned long long int matr[2][2], unsigned long long int auxmat[2][2])
{
  unsigned long long int primero =  matr[0][0]*auxmat[0][0] + matr[0][1]*auxmat[1][0];
  unsigned long long int segundo =  matr[0][0]*auxmat[0][1] + matr[0][1]*auxmat[1][1];
  unsigned long long int tercero =  matr[1][0]*auxmat[0][0] + matr[1][1]*auxmat[1][0];
  unsigned long long int cuarto =  matr[1][0]*auxmat[0][1] + matr[1][1]*auxmat[1][1];

  matr[0][0] = primero;
  matr[0][1] = segundo;
  matr[1][0] = tercero;
  matr[1][1] = cuarto;
}
// gives you the power n of the matrix
void power(unsigned long long int matr[2][2], unsigned long long int n)
{
  if( n == 0 || n == 1)
      return;
  unsigned long long int auxmat[2][2] = {{1,1},{1,0}};

  power(matr, n/2);
  multiplicacion(matr, matr);

  if (n%2 != 0)
     multiplicacion(matr, auxmat);
}
// applieds the algorithm
unsigned long long int fibonacci(unsigned long long int n)
{
  unsigned long long int matr[2][2] = {{1,1},{1,0}};
  if (n == 0)
      return 0;
  power(matr, n-1);

  return matr[0][0];
}
```

N	Naïve recursive	Bottom up	Closed form	Matrix representation
0	89.84314	55.33333	179.8431	96.80392
1	20.7451	48.37255	207.4118	27.66667
2	62.17647	62.23529	138.2941	55.31373
3	89.86275	55.31373	131.3529	89.86275
4	69.13725	41.4902	138.3137	69.11765
5	172.902	48.41176	117.5294	89.88235
6	165.9412	48.39216	145.1961	117.5294
7	200.5098	62.19608	158.9608	69.11765
8	228.1569	55.33333	110.6275	103.7255
9	359.4314	69.17647	117.549	48.39216
10	553.098	89.90196	186.6667	131.3725
11	1783.882	124.451	159	421.7451
12	2025.765	76.03922	152.1373	103.7255
13	4964.235	76.05882	214.3529	214.3725
14	3208	82.96078	159.0392	117.4902
15	6561.451	103.7059	179.7843	69.11765
16	9154.235	103.7255	186.6863	165.9804
17	13841.84	276.5686	207.4314	89.88235
18	21717	48.37255	207.4118	131.3725
19	26743.65	76.03922	172.8431	138.2157
20	53922.8	76.05882	214.3725	172.8824
21	76932.76	117.5686	200.5098	103.6667
22	177082.8	131.3725	145.1961	214.3725
23	188615.6	117.5098	165.9608	138.2745
24	414864.1	103.6667	186.6078	159.0196
25	571336.4	110.6078	159.0196	117.5098
26	912275.5	145.1961	159.0196	159.0392
27	1457608	110.6471	186.7059	124.4902
28	2149429	138.3137	193.5882	124.4706
29	3530305	186.6275	172.8431	165.9608
30	5723825	145.1961	193.5882	172.8627
31	9013416	179.7451	179.7255	186.7059
32	15419430	117.5294	200.4706	165.9608
33	23762653	165.9608	186.6667	159
34	38923706	131.3725	193.5882	110.5882
35	62848577	152.1176	186.6863	172.8824
36	1.03E+08	186.6863	145.1961	110.6275
37	1.69E+08	159.0196	124.451	172.8039
38	3.09E+08	172.8431	179.7451	290.3529
39	4.43E+08	186.6471	138.3137	221.3137
40	8.76E+08	193.5686	138.2745	172.8235
41	1.25E+09	172.8431	145.1765	159

1.3 For the same value of n

Something really interesting is that not for all values of n is the same Fibonacci number because in some algorithms the values of the numbers start getting really big that you can not store it in a data type so they overflow and the numbers start changing, in the closed form of the algorithm as n grows there are some floating-point problems so it start changing.

1.4 Plot



On the graph (series 1,2,3,4 are in order of appearance) we can see that the recursive method is cut because the time grows so fast that the other methods won't be visible, for the bottom up approach it has a really closed linear growth for the closed form there are some peaks on the data it might be affected by the multiplication of big numbers or some floating point issues, but the time is usually faster than the bottom up algorithm, the 4th approach the matrix representation is a little bit faster when n is small but than it is mostly constant

2.DIVIDE AND CONQUER

2.1 brute force

For this algorithm we have to integer A and B with n bits each lets take this as doing it by hand.

AXB is equal to multiply each term by each other and then doing bit shifting as it advances for example

```
1011   (this is 11 in binary)
 x 1110   (this is 14 in binary)
  ======
      0000   (this is 1011 x 0)
     1011    (this is 1011 x 1, shifted one position to the left)
    1011     (this is 1011 x 1, shifted two positions to the left)
 + 1011      (this is 1011 x 1, shifted three positions to the left)
  =========
  10011010   (this is 154 in binary)
```

So how this will work? Well we have integer A with n elements and B has n elements and every element of B multiplies A , n times so we have n multiplications then we shift one position, we have n-1 shifts we realize the shift and the multiplication at the same time and then we have to add those numbers there are n sums

So we have $N*(N-1)+N= N^2$ so the brute force procedure is $O(n^2)$

2.2 Divide and Conquer

For making this way simpler we know that A and B have different numbers so:

$A=a_n a_{n-1} \dots a_0$

$B= b_n\ b_{n-1} \dots b_0.$

We can see A and B as

$A = A_{left}2^{(n/2)} + A_{right}$ [A_{left} and A_{right} contain leftmost and rightmost n/2 bits of A]

$B = B_{left}2^{(n/2)} + B_{right}$ [B_{left} and B_{right} contain leftmost and rightmost n/2 bits of B]

We can write the product of A and B as the following

$AB = (A_{left}2^{(n/2)} + A_{right})(B_{left}2^{(n/2)} + B_{right})$

$= C_2 2^n + C_1 2^{(n/2)} + C_0$

$C_2 = A_{left}B_{left}$

$C_1 = A_{left}B_{right} + A_{left}B_{right}$

$C_0 = A_{right}B_{right\backslash}$

If we look at the above formula, there are four size n/2 multiplications, so we divided the size n problem into four size n/2 sub - problems. But that doesn't help because the problem still have too many multiplications but we can write C_1 as the following

$(A_{left} + A_{right})(B_{left}+B_{right})-A_{left}B_{left}- A_{right}B_{right}$

This is the same as saying $(A_{left} + A_{right})(B_{left}+B_{right})-C_1- C_0$ so we save 1 multiplication with 2 sums which in terms of big elements is a lot

Finally we have that

$AB = 2^n A_{left}B_{left} + 2^{n/2} * [(A_{left} + A_{right})(B_{left} + B_{right}) - A_{left}B_{left} – A_{left}B_{right}] + A_{right}B_{right}$

2.3 Divide and Conquer complexity

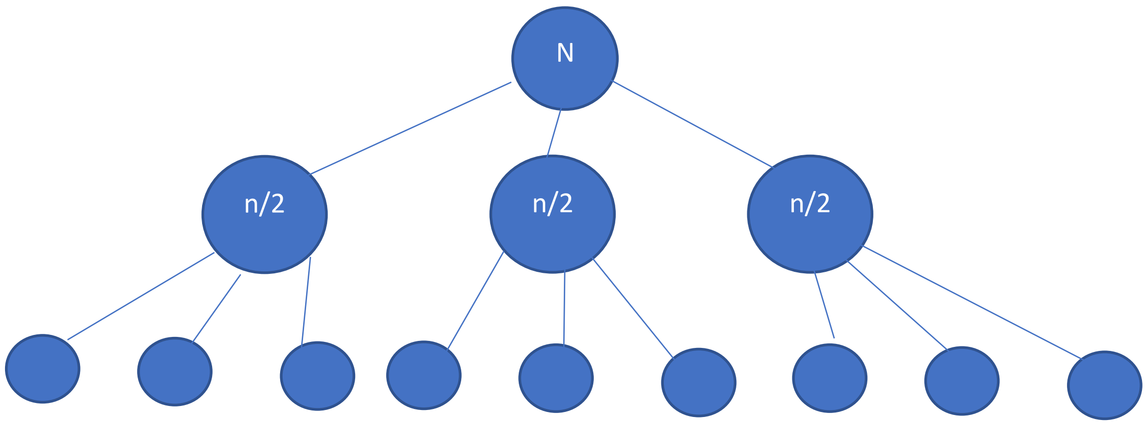
So we have our algorithm and will have the following

We divide our number in 3 parts that will have n/2 elements and we add the las number that is not a power of 2 is a linear expression addition

So T(n) = 3T(n/2)+O(n)

2.4 Recursion Tree

So T(n) = 3T(n/2)+O(n)



Sum

N

3(n/2)

9(n/4)

.

.

.

So ill be adding the following

$$Total = n\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \left(\frac{3}{2}\right)^3 + \left(\frac{3}{2}\right)^4 \ldots\right)$$

Because is a geometric series we got that 3/2 > 1 so

$$Total = n * \frac{1 - \frac{3^n}{2}}{1 - \frac{3}{2}}$$

$$Total = \frac{n - n(\frac{3}{2})^n}{-\frac{1}{2}} = \frac{-2n + 2n\frac{3^n}{2^n}}{1} = 2n\frac{3^n}{2^n} - 2n = 2n\left(\frac{3}{2}\right)^n - 2n = 2n * n^{\log_2 \frac{3}{2}} - 2n$$
$$= 2n^{1+\log_2 3 - \log_2 2} - 2n$$

We have a T(n)= ɵ ($n^{\log_2 3}$)

2.5 Master Theorem

So $T(n) = 3T(n/2)+O(n)$

$a=3$ $b=2$ $f(n)=n$

$$\log_2 3 \approx 1.584962$$

$n^{\log_2 3} > n$

so there is a $\varepsilon > 0$ $f(n) = O(n^{\log_2 3-\varepsilon})$

Then

$T(n)=\theta(n^{\log_2 3})$