Matrix Exponentiation

February 2019

1 Introduction

The Fibonacci numbers are the numbers in the following integer sequence.
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation :

$$F_n = F_{n-1} + F_{n-2}$$

with seed values $F_0 = 0$ and $F_1 = 1$
The problem statement is, given a number n, find the n^{th} Fibonacci number.

2 Algorithms

2.1 Recursion

NTH-FIBONACCI(n)

```
1  if n ≤ 1
2        return n
3  return NTH-FIBONACCI(n-1) + NTH-FIBONACCI(n-2)
```

Time complexity of the above code is according to the recurrence T(n) = T(n-1) + T(n-2), which is exponential in n. There is a lot of repetitive calculations that are being done in this method. We can avoid the repetitive work done in the recursive algorithm, by storing the Fibonacci numbers calculated so far.

2.2 Dynamic programming

NTH-FIB-DP(n)

```
1  let F[1..n+2] be a new array
2  F[0] = 0
3  F[1] = 1
4  for i = 2 to n
5        F[i] = F[i-1] + F[i-2]
6  return F[n]
```

Time complexity of the above code is $\mathcal{O}(n)$. But the space complexity is increased to $\mathcal{O}(n)$. We can optimize the space by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

We can make the code faster and bring down the time complexity, to calculate the n^{th} Fibonacci number to $\mathcal{O}(log(n))$ using Matrix Exponentiation.

2.3 Matrix Exponentiation

2.3.1 Matrix multiplication

Consider two matrices:
1. Matrix A having n rows and k columns.
2. Matrix B has k rows and m columns.
Then, the operation matrix multiplication is defined as $C = A * B$,

$$\begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,m} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,m} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,k} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,k} \end{pmatrix} * \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k,1} & b_{k,2} & \cdots & b_{k,m} \end{pmatrix}$$

such that C is a matrix with n rows and m columns, and each element in C should be computed using the following formula:

$$c_{i,j} = \sum_{r=1}^{k} a_{i,r} * b_{r,j}$$

Several things to notice:

1. Matrix C has the same number of rows as A, and the same number of columns as B.

2. Matrix C has $n * m$ elements, each element is computed in k steps with given formula, so the number of steps to obtain C in $\mathcal{O}(n * m * k)$, given A and B.

2.3.2 Matrix Exponentiation

Suppose we have a square matrix of degree n. We can define matrix exponentiation as:
$A^x = A * A * A * ... * A$ (x times)
with a special case of $X = 0 \implies A^0 = I_n$.

NAIVE-MATRIX-POWER(A, x)
1 $R = I_n$
2 **for** $i = 1$ **to** x
3 $\quad R = R * A$
4 return R

The above algorithm runs in $\mathcal{O}(n^3 * x)$, we can make a efficient algorithm by exponentially calculating the even powers of A, such as:

$$A^{2^r} = A^{2*(2^{r-1})} = A^{2^{r-1}+2^{r-1}} = A^{2^{r-1}} * A^{2^{r-1}}$$

thus doing only $\mathcal{O}(r*n^3)$ multiplications, where $r = \mathcal{O}(log_2(x))$ we can calculate the value of A^{2^r}, whereas in the naive algorithm we had to do x multiplications. We implement this idea in the following algorithm.

MATRIX-POWER-OPTIMIZED(A, x)
1 $R = I_n$
2 **while** $x > 0$
3 $\quad$ **if** $x\%2 == 1$
4 $\quad\quad R = R * A$
5 $\quad A = A * A$
6 $\quad x = x/2$
7 return R

We notice that we are able to calculate A^x in $\mathcal{O}(n^3 * log_2(x))$ time. We use this method to find the n^{th} Fibonacci Number in $\mathcal{O}(log_2(n)$ time.

2.3.3 N^{th} Fibonacci Number

The recurrence relation for the n^{th} Fibonacci Number is written as $F(n) = F(n-1) + F(n-2)$. Using matrix exponentiation we can find F_n for values as big as 10^{18}.

We can write the consecutive terms of Fibonacci series in the form of vectors $v_1 = \begin{pmatrix} F_{i-2} & F_{i-1} \end{pmatrix}$ and $v_2 = \begin{pmatrix} F_{i-1} & F_i \end{pmatrix}$ .We have to find a matrix M satisfying the following condition:

$$v_1 * M = v_2$$

In order to find the matrix M we have to find the dimensions of the matrix M and the exact values in M.

Using the properties of Matrix multiplication we calculate the dimension of the matrix M is, $r = 2$ and $c = 2$ where r and c are the number of rows and columns in the matrix respectively. Using this we generalize the matrix M as $M = \begin{pmatrix} x & y \\ u & v \end{pmatrix}$. To find x, y, z, w we multiply v_1 with M giving us,

$$\begin{pmatrix} F_{i-2} & F_{i-1} \end{pmatrix} * \begin{pmatrix} x & y \\ u & v \end{pmatrix} = \begin{pmatrix} F_{i-2} * x + F_{i-1} * z & F_{i-2} * y + F_{i-1} * w \end{pmatrix}$$

on the other hand we know that the result of this multiplication must be v_2.

$$\begin{pmatrix} F_{i-2} * x + F_{i-1} * z & F_{i-2} * y + F_{i-1} * w \end{pmatrix} = \begin{pmatrix} F_{i-1} & F_i \end{pmatrix}$$

Equating the matrices and solving x, y, z, w using the Fibonacci recurrence, we find the values $x = 0$, $z = 1$, $y = 1$, $w = 1$. We know the size and contents of the matrix M.

Initially, we have F_0 and F_1. Arranging them as a vector,

$$\begin{pmatrix} F_0 & F_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \end{pmatrix}$$

Multiplying this vector with the matrix M will get us,

$$\begin{pmatrix} 1 & 1 \end{pmatrix} * M = \begin{pmatrix} 1 & 2 \end{pmatrix}$$

Multiplying the above with M again is calculated as,

$$\begin{pmatrix} 1 & 2 \end{pmatrix} * M = \begin{pmatrix} 2 & 3 \end{pmatrix}$$

We see that we can get the same value by multiplying M two times to $\begin{pmatrix} 1 & 1 \end{pmatrix}$

$$\begin{pmatrix} 1 & 1 \end{pmatrix} * M * M = \begin{pmatrix} 2 & 3 \end{pmatrix}$$

So generalizing for n we can calculate $\begin{pmatrix} F_n & F_{n+1} \end{pmatrix}$ as

$$\begin{pmatrix} 1 & 1 \end{pmatrix} * M^n = \begin{pmatrix} F_n & F_{n+1} \end{pmatrix}$$

NTH-FIB-MATRIX-EXPONENTIATION(n)

```
1  let I = (1, 1)
2  if n ≤ 1
3        return 1
4  let M = ((1, 0), (1, 1))
5  E = MATRIX-POWER-OPTIMIZED(M, n − 1)
6  return (I ∗ E)[1][2]
```

We find that the time complexity of this algorithm is $\mathcal{O}((sizeof M)^3 * log_2(n))$ that is $\mathcal{O}(log_2(n))$. Using matrix exponentiation any form of recurrence expressions linear in terms of n can be calculated very fast.

References

[1] Fibonacci number
https://en.wikipedia.org/wiki/Fibonacci_number

[2] Program for Fibonacci numbers
https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

[3] Matrix Exponentiation
https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

[4] Matrix exponentiation
https://www.geeksforgeeks.org/matrix-exponentiation/

[5] I, ME AND MYSELF !!!
http://zobayer.blogspot.com/2010/11/matrix-exponentiation.html