GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: SUCESIÓN DE FIBONACCI

1. Introducción

La conocida sucesión de Fibonacci es aquella sucesión que comienza con los términos 0 y 1, y continua obteniendo cada término sumando sus dos anteriores. Mas formalmente:

$$Fib(n) = \begin{cases} 0 & \text{si} \quad n = 0 \\ 1 & \text{si} \quad n = 1 \\ Fib(n-1) + Fib(n-2) & \text{si} \quad n > 1 \end{cases}$$

Esta sucesión o alguna se sus variantes esta presente en varios problemas de concursos por lo cual es vital que un concursante sepa cuales son las maneras de calcular cualquier elemento de sucesión conociendo el orden de este dentro de la sucesión, así como las propiedades que presenta la sucesión.

2. Conocimientos previos

2.1. Sucesión númerica

Una **sucesión númerica** es un conjunto ordenado de números, que se llaman **términos** de la sucesión. Cada término se representa por una letra y un subíndice que indica el lugar que ocupa dentro de ella. De acuerdo a la cantidad de términos la msma se puede clasificar como **sucesión finita** cuando la cantidad de términos es finito mientras cuando ocurre lo contrario se dice que estamos en presencia de una **sucesión infinita**. El **término general** ó **término n-ésimo**, a_n, de una sucesión es una fórmula que nos permite calcular cualquier término de la sucesión en función del lugar que ocupa.

2.2. Exponenciación Binaria

La exponenciaión binaria es una técnica que permite generar cualquier cantidad de potencia n^{th} para multiplicaciones O ($\log N$) (en lugar de n multiplicaciones en el método habitual).

2.3. Multiplicación de matrices

En matemática, la multiplicación o producto de matrices es la operación de composición efectuada entre dos matrices, o bien la multiplicación entre una matriz y un escalar según unas determinadas reglas.

Vamos a considerar dos matrices:

- La matriz A con n filas y k columnas.
- La matriz B con k filas y m columnas.

Para poder efectuar una multiplicacion entre dos matrices la cantidad de columnas de una debe coincidir con la cantidad de filas de la otra matriz. Si se cumple eso la operación se puede definir

como $C = A * B$

$$\begin{bmatrix} a_{11} & a_{12} & ... & a_{1k} \\ a_{21} & a_{22} & ... & a_{2k} \\ ... & ... & ... & ... \\ a_{n1} & ... & ... & a_{nk} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & ... & b_{1m} \\ b_{21} & b_{22} & ... & b_{2m} \\ ... & ... & ... & ... \\ b_{k1} & ... & ... & b_{km} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & ... & c_{1m} \\ c_{21} & c_{22} & ... & c_{2m} \\ ... & ... & ... & ... \\ c_{n1} & ... & ... & c_{nm} \end{bmatrix}$$

Tal que C es una matriz con n filas y m columnas y cada elemento de C se puede calcular con la siguiente fórmula:

$$C_{ij} = \sum_{r=1}^{k} A_{ir} * B_{rj}.$$

3. Desarrollo

En este orden de ideas, es fácil descubrir los primeros términos de la sucesión:
$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$

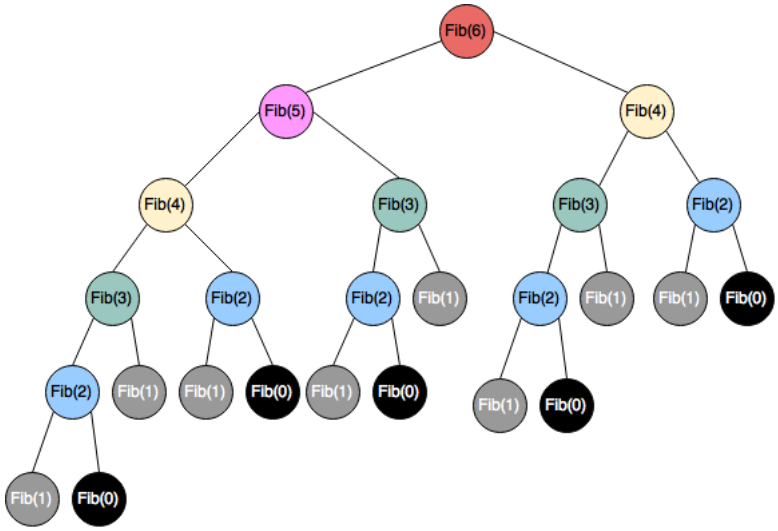
Del mismo modo podemos expresar cada término en base a su función:
Fib(0)=0, Fib(1)=1, Fib(2)=1, Fib(3)=2, Fib(4)=3, Fib(5)=5, ...

Dada esta definición formal, no es nada difícil crear una primera solución recursiva para obtener un término n de la sucesión de Fibonacci. Pero analicemos a fondo lo que hace esta función. Recursivamente, estará llamando a sus dos anteriores. Cada una de estas llamadas, si es mayor que 1, de nuevo estará llamando a sus dos anteriores. Analicemos el árbol de llamadas para un fibonacci de orden 6:

Pero es eficiente? Volvamos al árbol. Para calcular 1 vez Fib(6), es necesario calcular 1 vez Fib(5), 2 veces Fib(4), 3 veces Fib(3), 5 veces Fib(2), 8 veces Fib(1) y cinco veces Fib(0) (Como el algoritmo considera constante Fib(1), solo tenemos 5 llamadas a Fib(0). Si Fib(1) se calculara con su antecesor, tendríamos 13 llamadas). ¡Estamos repitiendo operaciones! Y lo peor, estas operaciones se repiten justamente siguiendo la secuencia de Fibonacci como se puede ver en los números en negrilla (1,1,2,3,5,8,13).

Si eliminamos los cálculos repetidos, fácilmente podemos tener un algoritmo de iterativo. Para hacer esto, utilizaremos un array que guarde los valores calculados de forma que en la posición i estará la suma de los valores presentes en $i-1$ y $i-2$ partitiendo que para i igual 0 y 1 esos serán sus valores respectivamente.

Ahora el algoritmo es mucho mas eficiente, reduciendo enormemente el número de operaciones innecesarias para valores altos de n. Pero surge un nuevo problema. La complejidad espacial



es de $O(n)$ también. Hemos sacrificado espacio por tiempo, ahora tenemos un array de n posiciones cuando en realidad solo necesitábamos la posición n y las demás podemos desecharlas. Como el algoritmo únicamente necesita de los dos valores anteriores para obtener el siguiente podemos remplazar el array por un grupo de tres variables. Ahora la complejidad temporal sigue siendo de orden $O(n)$ pero la complejidad espacial se reduce a un orden $O(1)$.

Volviendo al inicio, sabemos que:

$Fib(n) = Fib(n-1) + Fib(n-2)$

Partiendo de aquí, podemos fácilmente desarrollar un sistema de ecuaciones:

$$0Fib(n-2) + 1Fib(n-1) = 1Fib(n-1)$$
$$1Fib(n-2) + 1Fib(n-1) = 1Fib(n-1)$$

Y por supuesto, lo reescribiremos en notación matricial:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f(n-2) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} f(n-1) \\ f(n) \end{bmatrix}$$

Verificando estos valores para n = 2

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} f(1) \\ f(2) \end{bmatrix}$$

Conociendo que Fib(0)=0 y Fib(1)=1:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Y efectivamente, Fib(2)=1. Ahora bien, generalizando para diferentes valores de n.

$$\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{n-1}\begin{bmatrix}0\\ 1\end{bmatrix}=\begin{bmatrix}f(n-1)\\ f(n)\end{bmatrix}$$

$$\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{n-1}=\begin{bmatrix}f(n-2) & f(n-1)\\ f(n-1) & f(n)\end{bmatrix}$$

Ahora recordemos 3 propiedades de la potenciación:

1. $A^1 = A$
2. $A^{m^n} = A^{mn}$
3. $A^n = A^i A^j$ si $i + j = n$

Los cuales nos permitirán hacer un "divide y vencerás" para resolver la potencia de una manera eficiente.

$$A^n = \begin{cases} A & si\ n = 1 \\ (A^{\frac{n}{2}})^2 & si\ n\%2 == 0 \\ A * A^{n-1} & si\ n\%2! = 0 \end{cases}$$

¿Que demonios he hecho? Básicamente, partir el problema. Para explicarlo, miremos paso a paso el proceso para hallar el Fib(11):

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix}=\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{10}$$
$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix}=\left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{5}\right)^{2}$$
$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix}=\left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{4}\right)^{2}$$
$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix}=\left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}\left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{2}\right)^{2}\right)^{2}$$
$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix}=\left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & 2\end{bmatrix}^{2}\right)^{2}$$

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix}=\left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}2 & 3\\ 3 & 5\end{bmatrix}\right)^{2}$$

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix}=\begin{bmatrix}3 & 5\\ 5 & 8\end{bmatrix}^{2}$$

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix}=\begin{bmatrix}34 & 55\\ 55 & 89\end{bmatrix}$$

Y así llegamos a la respuesta de manera mas rápida.

Esto no es una última mejora, en realidad es un cambio total. Hasta ahora hemos manejado cada término de la sucesión en base a sus términos anteriores. Pero existe también una formula explicita creada por Lucas para hallar directamente un término sin necesidad de iterar.

$$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

3.1. Propiedades de la sucesión de Fibonacci

1. Tan sólo un término de cada tres es par, uno de cada cuatro es múltiplo de 3, uno de cada cinco es múltiplo de 5, etc. Esto se puede generalizar, de forma que la sucesión de Fibonacci es periódica en las congruencias módulo m, para cualquier m.
2. Cualquier número natural se puede escribir mediante la suma de un número limitado de términos de la sucesión de Fibonacci, cada uno de ellos distinto a los demás. Por ejemplo, 17=13+3+1, 65=55+8+2.
3. Cada número de Fibonacci es el promedio del término que se encuentra dos posiciones antes y el término que se encuentra una posición después. Es decir:

   $F_n = \frac{F_{n-2}+F_{n+1}}{2}$
4. Lo anterior también puede expresarse así: calcular el siguiente número a uno dado es 2 veces éste número menos el número 2 posiciones más atrás.

   $F_{n+1} = F_n * 2F_{n-2}$
5. El último dígito de cada número se repite periódicamente cada 60 números. Los dos últimos, cada 300; a partir de ahí, se repiten cada $15 \times 10^{n-1}$ números.
6. La suma de los n primeros números es igual al número que ocupa la posición $n+2$ menos uno. Es decir

   $F_{n+2} - 1 = F_n ... + F_2 + F_1$
7. La suma de diez números Fibonacci consecutivos es siempre 11 veces superior al séptimo número de la serie.
8. El máximo común divisor de dos números de Fibonacci es otro número de Fibonacci. Más específicamente

   $mcd(F_n, F_m) = F_{mcd(n,m)}$
9. Otras identidades interesantes incluyen las siguientes:

   $f_0 - f_1 + f_2 - \cdots + (-1)^n f_n = (-1)^n f_{n-1} - 1$

   $f_1 + f_3 + f_5 + \cdots + f_{2n-1} = f_{2n}$

   $f_0 + f_2 + f_4 + \cdots + f_{2n} = f_{2n+1} - 1$

   $f_0^2 + f_1^2 + f_2^2 + \cdots + f_n^2 = f_n f_{n+1}$

   $f_1 f_2 + f_2 f_3 + f_3 f_4 + \cdots + f_{2n-1} f_{2n} = f_{2n}^2$

   $f_{n+2}^2 - f_n^2 = f_{2n+2}$

   $f_{n+2}^3 + f_{n+1}^3 - f_n^3 = f_{3n+3}$

4. Implementación

4.1. C++

4.1.1. Variante recursiva

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

4.1.2. Variante iterativa

```
int fibonacci[1000];

// Complejidad temporal y espacial O(n)
void buildFibonacci(){
   fibonacci[0]=0; fibonacci[1]=1;
   for(int i=2;i<1000;i++) fibonacci[i]=fibonacci[i-1]+fibonacci[i-2];
}

// Complejidad temporal y espacial O(n) y O(1) respectivamente
int fibonacci(int n){
   int a=0, b=1,c=n;
   for(int i=2;i<=n;i++){ c=a+b; a=b; b=c; }
   return c;
}
```

4.1.3. Variante matricial

```
int fibonacci(int n){
   int h,i,j,k,aux; h=i=1; j=k=0;
   while(n>0){
      if(n%2!=0){
         aux=h*j; j=h*i+j*k+aux; i=i*k+aux;
      }
      aux=h*h; h=2*h*k+aux; k=k*k+aux; n=n/2;
    }
    return j;
}
```

4.1.4. Función

```
double fibonacci(int n){
```

```
   double root=sqrt(5);
   double fib=((1/root)*(pow((1+root)/2,n))- (1/root)*(pow((1-root)/2,n)));
   return fib;
}
```

4.2. Java

4.2.1. Variante recursiva

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

4.2.2. Variante iterativa

```
int [] fibonacci = new int[1000];

// Complejidad temporal y espacial O(n)
public void buildFibonacci(){
   fibonacci[0]=0; fibonacci[1]=1;
   for(int i=2;i<1000;i++) fibonacci[i]=fibonacci[i-1]+fibonacci[i-2];
}

// Complejidad temporal y espacial O(n) y O(1) respectivamente
public int fibonacci(int n){
   int a=0, b=1,c=n;
   for(int i=2;i<n;i++){ c=a+b; a=b; b=c; }
   return c;
}
```

4.2.3. Variante matricial

```
public int fibonacci(int n){
   int h,i,j,k,aux; h=i=1; j=k=0;
   while(n>0){
      if(n%2!=0){
         aux=h*j; j=h*i+j*k+aux;i=i*k+aux;
      }
      aux=h*h; h=2*h*k+aux; k=k*k+aux; n=n/2;
   }
   return j;
}
```

4.2.4. Función

```
public double fibonacci(int n){
   double root=Math.sqrt(5);
   double fib=((1/root)*(Math.pow((1+root)/2,n))- (1/root)*(Math.pow((1-root)
      /2,n)));
   return fib;
}
```

5. Complejidad

La solución recursiva es un algoritmo de orden exponencial, mas exactamente ϕ^n. Esto significa que es terriblemente lento, pues tendrá que hacer excesivas operaciones para valores altos de n.

La solución iterativa tiene una complejidad tanto espacial como temporal de $O(n)$ siendo n el término enésimo de fibonacci a calcular.

La solución matricial su complejidad es $O(log_2 n)$. Para hacernos una idea, para calcular el Fib(200) con el algoritmo exponencial, ocuparíamos $6{,}2737x10^{41}$ cálculos (interminable). Con el algoritmo de orden n ocuparíamos 200 cálculos, y con este algoritmo, 8 operaciones (Por supuesto estos datos son aproximados, pero muestran claramente las diferencias abismales).

En cuanto a la función es muy eficiente, pero su orden depende de la manera de implementar la potenciación. Una potenciación por cuadrados como en el caso anterior, arroja un costo de $O(log_2 n)$. Pero se debe tener mucho cuidado en su implementación, pues cada lenguaje tiene diferentes maneras de implementar las funciones matemáticas, y muy posiblemente sea necesario redondear el número para dar la respuesta exacta.

6. Aplicaciones

Dentro de las sucesiones conocidas la de Fibonacci es de las más utilizadas en problemas de concursos siempre con alguna que otra variación o con la aplicación de algunas de las propiedades que ella presente es por ello la importancia de concerla y las formas de hallar determinado término de la sucesión.

7. Ejercicios propuestos

El siguiente listado de problemas para su solución debemos apoyarnos en la suseción de Fibonacci.

- DMOJ - Salto de rana
- DMOJ - Fibonacci 2D
- DMOJ - Secuencia numerada de lapices

- DMOJ - Fibonacci Calculation