UNIVERSIDAD
DE SANTIAGO
DE CHILE

Departamento de Matemática y Ciencia de la Computación

Laboratorio 3

Segundo Semestre 2020

Fundamentos de Ciencia de la Computación 2 22628-2s

Licenciatura en Ciencia de la Computación

Rodrigo Kobayashi Araya
Javier Gomez Gallegos
rodrigo.kobayashi@usach.cl
javier.gomez.g@usach.cl

1 Introducción

La complejidad de tiempo mide el número de pasos que un algoritmo necesita para resolver algún problema el cual tiene un tamaño de entrada definido. Este calculo es vital para el análisis y desarrollo de algoritmos, teniendo en cuenta la eficiencia temporal. A continuación se presentará el enunciado del laboratorio 3 de Fundamentos de Ciencia de la Computación, las actividades por hacer, sus desarrollos y conclusiones.

2 Enunciado

Considere el problema de hallar el n-ésimo número de la secuencia de Fibonacci, esta sucesión se describe como:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

Se puede definir la recurrencia:

$$F(n) = F(n-1) + F(n-2), \text{ para } n > 1$$

Y las condiciones iniciales:

$$F(0) = 0, F(1) = 1$$

Los siguientes algoritmos presentan dos maneras diferentes de hallar el n-ésimo número de la secuencia de Fibonacci, el primero es un algoritmo recursivo y el segundo uno iterativo:

Algorithm FibonacciRecursivo(n)

Input: Un entero n no negativo
Output: El n-ésimo número de la sucesión de Fibonacci

```
1.- if (n < 2) then
2.-     return n
3.- else
4.-     return FibonacciRecursivo(n-1) + FibonacciRecursivo(n-2)
5.- end if
```

Algorithm FibonacciIterativo(n)

Input: Un entero n no negativo
Output: El n-ésimo número de la sucesión de Fibonacci

```
1.- i = 1, j = 0
2.- for (k=1 to n) do
3.-     j = i+j
4.-     i = j-i
5.- end for
6.- return j
```

Existe un tercer algoritmo que se puede obtener al analizar el proceso iterativo siguiente:

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}_n$$

3 Actividades

1.- Calcular la complejidad teórica de los tre algoritmos anteriores. Argumentando su respuesta.
2.- Demostrar que la sucesión de Fibonacci satisface la ecuación anterior
3.- Implementar en lenguaje C los tres algoritmos anteriores y determinar el costo computacional de encontrar el n-ésimo valor de la secuencia de Fibonacci

4 Planteamiento de Hipótesis

Luego de analizar el enunciado y actividades del laboratorio, se concluye que las actividades principales de este es el análisis de complejidad temporal de los algoritmos presentados, por lo tanto se necesita calcular la complejidad temporal teórica y, luego de programar dichos algoritmos, hacer las mediciones de tiempo pertinentes para comparar la complejidad teórica con la empírica.
La hipótesis planteada por los integrantes es que la complejidad teórica coincidirá con la empírica.

5 Complejidad teórica

5.1 FibonacciRecursivo

Tomando el algoritmo:

Algorithm FibonacciRecursivo(n)
Input: Un entero n no negativo
Output: El n-ésimo número de la sucesión de Fibonacci

```
1.- if (n < 2) then
2.-     return n
3.- else
4.-     return FibonacciRecursivo(n-1) + FibonacciRecursivo(n-2)
5.- end if
```

Se puede apreciar que, dentro de la función se hace una comparación, dos substracciones y una adición.

luego, en el paso iterativo, se generan dos llamadas a funciones nuevas, tomando el tiempo como T(n), se puede expresar de la manera:

$$T(n) = T(n-1) + T(n-2) + 4$$

Si asumimos que el tiempo que toma cada función en particular es el mismo, podemos obtener:

$$T(n) = T(n-1) + T(n-2) + c/ \text{ asumiendo } T(n-1) = T(n-2)$$
$$T(n) = 2T(n-2) + c$$
$$T(n) = 2 * (2T(n-4) + c) + c$$
$$T(n) = 2^2T(n-4) + 3c$$
$$T(n) = 2^3T(n-6) + 7c$$
$$T(n) = 2^4T(n-8) + 15c$$
$$T(n) = 4T(n-4) + 3c$$
$$T(n) = 2^k * T(n-k) + (2^k - 1) * c$$

para F(0), que es donde la recursión termina, k = n, por lo tanto:

$$T(n) = 2^n * T(0) + (2^n - 1) * c$$
$$T(n) = 2^n + c_1$$
$$F_r(n) \in O(2^n)$$

5.2 FibonacciIterativo

Tomando el algoritmo:

Algorithm FibonacciIterativo(n)

Input: Un entero n no negativo
Output: El n-ésimo número de la sucesión de Fibonacci

```
1.- i = 1, j = 0
2.- for (k=1 to n) do
3.-     j = i+j
4.-     i = j-i
5.- end for
6.- return j
```

se puede observar que, dentro del ciclo hay 2 asignaciones y 2 sumas. Fuera de el, hay dos asignaciones y el retorno. Por lo tanto:

$$T(n) = (T(n-1) + 4) + 3$$
$$T(n) = (T(n-2) + 4 + 4) + 3$$
$$T(n) = T(n-k) + 4(k) + 3$$
Para $n - k = 0 \Rightarrow n = k$
$$T(n) = T(0) + 4n + 3$$
$$F_i(n) \in O(n)$$

5.3 FibonacciMatriz

Para obtener F(n) usando el método de la matriz descrito anteriormente, requiere de n multiplicaciones de matrices 2x2, las cuales son de la forma:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ac + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

Luego, multiplicar una matriz requiere de 4 sumas, 8 multiplicaciones y 4 asignaciones (para cada celda de la matriz).Por lo tanto:

$$T(n) = T(n-1) + 16$$
$$T(n) = T(n-2) + 16 + 16$$
$$T(n) = T(n-k) + 16 * k$$
para $n = k$:
$$T(n) = T(0) + 16 * n$$
$$F_m(n) \in O(n)$$

5.4 FibonacciMatriz con exponenciación binaria

La exponenciación binaria es una forma de conseguir potencias la siguiente manera: Ya que:

$$a^{b+c} \text{ y } a^{2b} = a^b * a^b = (a^b)^2$$

Se puede calcular la potencia a partir de la representación binaria del exponente:

$$3^19 = 3^{10011_2} = 3^16 * 3^2 * 3^1$$

De esta manera falta calcular las potencias de los exponentes que son potencias de 2, por ejemplo:

$$3^1 = 3$$
$$3^2 = (3^1)^2 = 3^2 = 9$$
$$3^4 = (3^2)^2 = 9^2 = 81$$
$$3^8 = (3^4)^2 = 81^2 = 6561$$
$$3^16 = (3^8)^2 = 6561^2 = 43046721$$

Finalmente:

$$3^19 = 3^{10011_2} = 3^16 * 3^2 * 3^1 \; 3^19 = 43046721 * 9 * 3 = 1162261467$$

De esta manera, para la potencia a^n, se requieren c pasos para transformar el exponente a binario, calcular $log_2 n$ potencias y multiplicar $log_2 n$ números. Además, multiplicar 2 matrices requieren de 16 pasos, por lo que, finalmente, la complejidad de tiempo de esta forma de calcular el n-ésimo término de la secuencia de Fibonacci usando la exponenciación binaria de una matriz es:

$$T(n) = 16 * log_2 n + log_2(n) + c$$
$$F_{binexp}(n) \in O(log\ n)$$

6 Demostración de la ecuación mostrada en el punto 2

La ecuación mostrada anteriormente es la siguiente:

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

Por inducción, para n=1:

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

Lo cual se cumple. Ahora, supongamos que la ecuación anterior se cumple. Entonces:

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

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

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

Ya que $F(n) = F(n-1) + F(n-2)$, para $n > 1$, entonces:

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

Por lo tanto, queda demostrado que la ecuación de fibonacci satisface la ecuación anterior.

7 Imlpementación

Se presenta el código implementado en C para el problema presentado en este laboratorio:

Algorithm FibonaxiRecursivo(n)

Input: Un int n no negativo
Output: El n-ésimo número de la sucesión de Fibonacci

```
1  unsigned long long int FibonaxiRecursivo(int n)
2  {
3     if (n < 2)
4     {
5     return 1;
6     }
7     return FibonaxiRecursivo(n - 1) + FibonaxiRecursivo(n - 2);
8  }
```

Algorithm FibonaxiIterativo(n)

Input: Un int n no negativo
Output: El n-ésimo número de la sucesión de Fibonacci

```
1  unsigned long long int FibonaxiIterativo(int n)
2  {
3   unsigned long long int d = 0, e = 0, f = 1;
4   for (d = 0; d < n; d++)
5   {
6   e = f + e;
7   f = e - f;
8   }
9   return e;
10 }
```

Algorithm linearFibonaxi(n)

Input: Un int n no negativo
Output: El n-ésimo número de la sucesión de Fibonacci

```
1  unsigned long long int linearFibonaxi(int n)
2  {
3   unsigned long long int **mt, **mresult, **zero;
4   if (n < 2)
5   { //Caso base
6   return 1;
7   }
8   else
9   { //Caso general

/*Inicializacion de matrices*/

10  mt = initMatrix(mt, 2);
11  mresult = initMatrix(mresult, 2);
12  zero = initMatrix(zero, 2);
13  mt = setMatrix(mt, 2);
14  mresult = setMatrix(mresult, 2);
15  zero = setMatrix(zero, 2);
16  mt[0][1] = 1;
17  mt[1][0] = 1;
18  mt[1][1] = 1;
19  mresult = exponentation(n, mt, mresult, zero); //Computo de power matrix
20  return mresult[1][0];
21   }
22 }
```

Tomando en cuenta que exponentation() es la función de multiplicación de una matriz por fuerza bruta.

Algorithm binaryFibonaxi(n)

Input: Un int n no negativo
Output: El n-ésimo número de la sucesión de Fibonacci

```
1  unsigned long long int binaryFibonaxi(int n)
2  {
3   unsigned long long int **mt, **mresult, **zero;
4   if (n < 2)
5   { //Caso base
6   return 1;
7   }
8   else
9   { //Caso general

/*

Inicializacion de matrices

*/

10  mt = initMatrix(mt, 2);
11  mresult = initMatrix(mresult, 2);
12  zero = initMatrix(zero, 2);
13  mt = setMatrix(mt, 2);
14  mresult = setMatrix(mresult, 2);
15  zero = setMatrix(zero, 2);
16  mt[0][1] = 1;
17  mt[1][0] = 1;
18  mt[1][1] = 1;

//Se define mresult como identidad 2x2

19  mresult[0][0] = 1;
20  mresult[1][1] = 1;
21
22  mresult = binaryExponentation(n, mt, mresult, zero);
        //Computo de power matrix

23  return mresult[1][0];
24  }
25 }
```

con la funcion binaryExponentation:

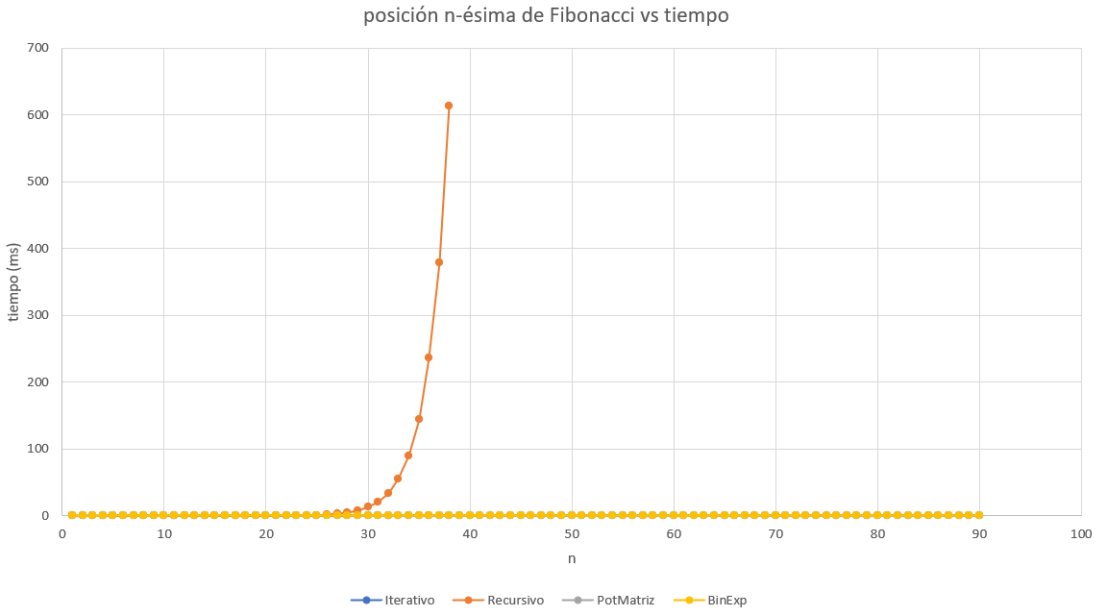
Algorithm BinaryExponentation(n)

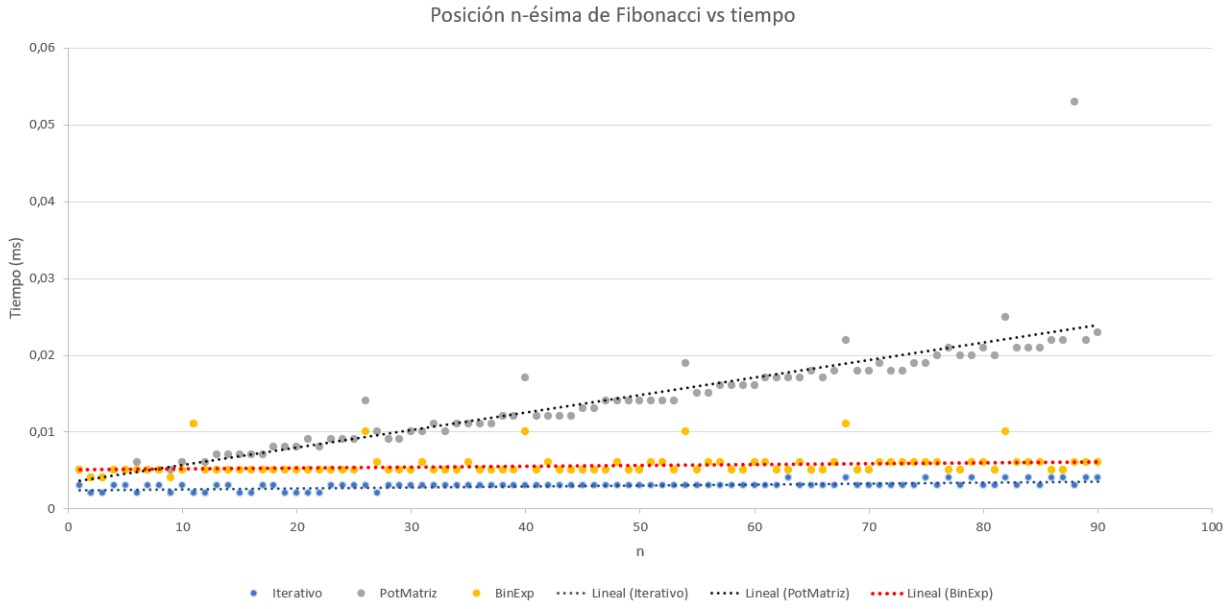
Input: n, int; mt, unsigned long long int**;
mresult, unsigned long long int**; zero, , unsigned long long int**
Output: matriz mt elevada a n

```
1  unsigned long long int **binaryExponentation(int n, unsigned long long int
  **mt, unsigned long long int **mresult, unsigned long long int **zero)
2  {
3   char flag = '0';
4   while (n > 0)
5   {
6   if (n % 2 == 1)
7   {
//Si es la primera vez que se entra en este condicional
//mt se copia en mresult, pues mresult es identidad

8   zero = BTMultiplication(mresult, mt, zero);
9       mresult[0][0] = zero[0][0];
10   mresult[0][1] = zero[0][1];
11   mresult[1][0] = zero[1][0];
12   mresult[1][1] = zero[1][1];
13   zero = setMatrix(zero, 2);
14   flag = '1';
15   }
16   zero = BTMultiplication(mt, mt, zero);
17   mt[0][0] = zero[0][0];
18   mt[0][1] = zero[0][1];
19   mt[1][0] = zero[1][0];
20   mt[1][1] = zero[1][1];
21   zero = setMatrix(zero, 2);
22   n = n / 2;
23   }
//Los numeros que son iguales 2^k nunca entran en el condicional
//Por lo que es necesario marcar cuando se entra y cuando no

24   if (flag == '1')
25   {
26   return mresult;
27   }
28   return mt;
29  }
```

8 Costo computacional

Luego de crear el programa en c para calcular las n-ésimas posiciones bajo los 4 algoritmos distintos, se crean los siguientes gráficos:





De estos gráficos se puede concluir que el algoritmo recursivo es, en efecto, exponencial. Por esto se decide generar el segundo gráfico, que muestra el comportamiento de los algoritmos sin tomar en cuenta el recursivo.

En cuanto al segundo gráfico, se puede concluir que, para n's pequeños los tres algoritmos se comportan como funciones lineales. Sin embargo, para valores de n muy grandes, se comprueba a través de pruebas que el algoritmo de exponenciación binaria se comporta de mejor manera que el iterativo y el de potencia de matriz por fuerza bruta.

Pese a estos datos, no se logra encontrar el valor en que el algoritmo lineal supera al de exponenciación binaria en tiempo, ya que, para valores de n pequeños los tiempos son muy cortos, lo cual genera errores en perspectiva del tiempo. La diferencia entre estos dos algoritmos se nota sólamente cuando el algoritmo lineal sobrepasa por mucho al de exponenciación binaria en el tiempo de ejecución, por ejemplo, para n = 1000000, el tiempo del algoritmo iterativo es de 15625 [ms], y para el de exponenciación binaria, el tiempo que toma es tan corto que no se logra captar por el cálculo (0,0 [ms]).

9 Conclusiones

Luego de demostrar que la sucesión de fibonacci satisface la ecuación presentada, calcular las complejidades teóricas de los algoritmos y comprobarlas mediante mediciones de tiempo al correr el programa hecho, se concluye que el cálculo de la complejidad teórica coincide con la empírica.
Además, se conluye que la elección de algoritmos para solucionar problemas dependen totalmente del contexto de aplicación, por ejemplo, si se necesita aplicar el algoritmo para bajas cantidades de datos, no hay mucha diferencia entre un algoritmo que resuelva el problema en tiempo lineal y otro en tiempo logarítmico. Pero para grandes valores o cantidades, se puede apreciar la diferencia entre estos dos algoritmos por un margen enorme de tiempo.

Finalmente, se concluye que el cálculo teórico de complejidad de tiempo es de suma utilidad para crear algoritmos, ya que este cálculo no se aleja mucho de la realidad.