• Sesión 1: Algoritmos básicos. Samedi, février 8, 2020

① Diseño y análisis:

a) En que consiste el algoritmo de multiplicación a la francesa?

Este método de multiplicación es usado actualmente en algunos países europeos. Y nos ayuda y sirve para multiplicar dos números decimales 'x' y 'y'. Consiste en dividir x entre 2 de forma repetida, al mismo tiempo que multiplicamos y por 2, todo esto se repite hasta que x llegue a 1, despreciando si x es impar y solo tomando en cuenta su parte entera.
Por último, vamos a descartar las filas en las cuales x sea par, y sumando el resto de y's.

b) Explica con un ejemplo, cómo funciona?

~~42~~	~~28~~	
21	56	
~~10~~	~~112~~	+
5	224	
~~2~~	~~448~~	
1	896	
	$\boxed{1176}$	

c) Compáralo con el algoritmo que aprendiste para multiplicar en la primaria:

Multiplicación a la francesa:

```
function multiply(x, y) {
Input: Two n-bit integers x and y, where y ≥ 0
Output: Their product
  if y = 0 : return 0        —— C0, C1
  z = multiply(x, ⌊y/2⌋)     —— C2 · n
  if y is even :             —— C3
     return 2z               —— C4
  else :
     return x + 2z           —— C5
}
```

Comparación:

Analizando ambos métodos, podemos decir que el método francés es recursivo y el otro método es puro convencionalismo, ya que es una forma más de resolver una multiplicación.

Método que aprendí en la primaria:

Consiste en colocar un número bajo otro y multiplicar el multiplicando por cada una de las cifras del multiplicador, comenzando con la de menor peso, colocando los resultados parciales uno bajo otro, pero desplazándolos un lugar a la izquierda respecto al anterior. Finalmente, se suman los resultados parciales.

d) Mide la complejidad algorítmica de cada una, usando la notación O:

- Método Francés:

$$\text{Costo total} = C_0 + C_1 + C_2 \cdot n + C_3 + C_4 + C_5$$
$$= (C_0 + C_1 + C_3 + C_4 + C_5) + C_2 \cdot n$$
$$= a + bn$$

- Método Primaria:

Costo Total = Ya que el problema crece según el número de cifras (n) de cada número entonces se dice que su costo es: No polinomial.

$$T(n) = a + bn$$
$$T(n) = O(?)$$
$$0 \leq a + bn \leq cn \quad \forall n \geq 1$$

$\left\{ \begin{array}{l} c = a + b \\ n_0 = 1 \end{array} \right\}$ Si se cumple la condición $\therefore T(n) = O(n)$

(2) Investiga sobre dos algoritmos para factorizar un
a) número entero:

- Descomposición en factores primos.
- Factores compuestos.

b) Explica con un ejemplo pequeño cómo funciona cada uno de ellos:

- Descomposición en factores primos.
  Por el teorema fundamental de la aritmética, cada entero positivo tiene una única descomposición en números primos (factores primos). La mayor parte de los algoritmos de factorización elementales son de propósito general, es decir, permiten descomponer cualquier número introducido, y solo se diferencian sustancialmente en el tiempo de ejecución.

  • Ejemplo:

$$\left.\begin{array}{r|l} 72 & 2 \\ 36 & 2 \\ 18 & 2 \\ 9 & 3 \\ 3 & 3 \\ 1 & \end{array}\right\} 72 = 2^3 \cdot 3^2$$

- Métodos de factorización por factores compuestos
  Estos métodos consisten en, dado un número natural n, hallar dos números m y k (no necesariamente primos) que cumplan $n = mk$.

De manera general, estos algoritmos siguen los siguientes pasos para la factorización de n:

1.- Hallar dos números naturales m y k que cumplan que $mk = n$.

2.- Factorizar m y k utilizando alguno de los métodos de factorización.

Si se continúa aplicando este mismo tipo de método, se puede apreciar que se sigue un procedimiento de búsqueda de árbol, pues se va simultaneamente hallando los factores primos de n:

$$\begin{tikzcd}
 & & & n \arrow[dll] \arrow[drr] & & & \\
 & m_1 \arrow[dl] \arrow[dr] & & & & k_1 \arrow[dl] \arrow[dr] & \\
m_2 & & k_2 & & m_3 & & k_3
\end{tikzcd}$$

$$\cdots$$

$$p_1 \quad p_2 \quad p_3 \quad p_4 \quad p_5 \quad p_6 \quad p_7 \quad \cdots \quad p_j$$

Ejemplo:

$$6400 = 64 \cdot 100 \begin{array}{l} \longrightarrow 64 = 8^2 = 2^6 \\ \searrow 100 = 10^2 = 2^2 5^2 \end{array} \Bigg\} \; 6400 = 2^8 \cdot 5^2$$

c) ¿Qué se puede decir sobre la complejidad algorítmica de estos algoritmos, cuando se tiene un entero grande?

Si un número grande, de k bits es el producto de dos primos de aproximadamente el mismo tamaño, no existe algoritmo conocido capaz de factorizarlo en tiempo polinómico.

Esto significa que ningún algoritmo conocido puede factorizarlo en tiempo $O(b^k)$, para cualquier constante k. Aunque, existen algoritmos que son más rápidos que $O(a^b)$ para cualquier a mayor que 1. En otras palabras, los mejores algoritmos son súper-polinomiales, pero sub-exponenciales. En particular, el mejor tiempo asintótico de ejecución es el del algoritmo de Criba General del Cuerpo de Números (CGCN), que para un número n es:

$$O\left(\exp\left(\left(\frac{64}{9}b\right)^{\frac{1}{3}}(\log b)^{\frac{2}{3}}\right)\right)$$

(3)

a) Dos algoritmos para determinar si un número es primo o no:

```
inicio
   int i, suma = 0;
         leer (a)
         for (i=1; i<a; i++)
              if (a mod i = 0)
                       suma = suma + 1;
              end if
         if (suma = 1)
              imprimir ("Número primo");
         else
              imprimir ("Número no primo");
```

- Algoritmo para detectar número primo:

Divisor = 1
Escribir ("Ingrese número")
Leer num
Repetir
If num MOD divisor == 0
    divisor = divisor + 1
~~divisor = divisor + 1~~
~~If divisor~~
Hasta que divisor == num + 1
If divisor == 2
    Escribir PRIMO
Else
    Escribir NO PRIMO

b) Explica con un ejemplo:

1.- $a = 5$

$5 \bmod 1 = 0 \rightarrow suma = 1$
$5 \bmod 2 = 1$
$5 \bmod 3 = 2$
$5 \bmod 4 = 1$
Imprimir (PRIMO)

c) ¿Qué se puede decir sobre la complejidad algorítmica de estos algoritmos, cuando se tiene un entero grande?

Tomando el pseudocódigo como modelo, el algoritmo realiza $11(\frac{n}{2}-1)+3$ operaciones para el peor de los casos, por lo que el algoritmo tiene una función de complejidad lineal.

Considerando $n = 71$ digitos, entonces la función de complejidad de nuestro algoritmo $T(n) = 11(\frac{n}{2}-1)+3$ nos da 55×10^{69}.

Suponiendo que cada operación elemental pudiera ser realizada en tan solo un milisegundo, entonces tardaría 55×10^{69} milisegundos para determinar si es primo o no. Y es equivalente a 1.8945×10^{60} años

4) Calcular el mcd$(\cancel{210}, \cancel{588})$ 588 210

$$\begin{array}{r|l} 210 & 2 \\ 105 & 3 \\ 35 & 5 \\ 7 & 7 \\ 1 & \end{array} \qquad \begin{array}{r|l} 588 & 2 \\ 294 & 2 \\ 147 & 3 \\ 49 & 7 \\ 7 & 7 \\ 1 & \end{array}$$

$210 = 2 \cdot 3 \cdot 5 \cdot 7$

$588 = 2^2 \cdot 3 \cdot 7^2$

Método: Factorización en primos.

$mcd(588, 210) = 588/210 = 2_{168}$

$\hookrightarrow 588 = 210 \cdot 2 + 168$

$mcd(210, 168) = 210/168 = 1_{42}$

$\hookrightarrow 210 = 168 + 42$

$mcd(168, 42) = 168/42 = 4_0$

$\hookrightarrow 168 = 42 \cdot 4 + 0$

$mcd(42, 0) = \underline{42}$

Metodo: Algoritmo Euclides.

c) ¿Qué se puede decir sobre la complejidad algorítmica de estos algoritmos, cuando se tiene un entero grande?

Tomando el pseudocódigo como modelo, el algoritmo realiza $11(\frac{n}{2} - 1) + 3$ operaciones para el peor de los casos, por lo que el algoritmo tiene una función de complejidad lineal.

Considerando $n = 71$ dígitos, entonces la función de complejidad de nuestro algoritmo $T(n) = 11(\frac{n}{2} - 1) + 3$ nos da 55×10^{69}.

Suponiendo que cada operación elemental pudiera ser realizada en tan solo un milisegundo, entonces tardaría 55×10^{69} milisegundos para determinar si es primo o no. Y es equivalente a 1.8945×10^{60} años

4) Calcular el mcd$(\cancel{210}, \cancel{588})$ 588 210

$$
\begin{array}{r|l}
210 & 2 \\
105 & 3 \\
35 & 5 \\
7 & 7 \\
1 &
\end{array}
\qquad
\begin{array}{r|l}
588 & 2 \\
294 & 2 \\
147 & 3 \\
49 & 7 \\
7 & 7 \\
1 &
\end{array}
$$

$210 = 2 \cdot 3 \cdot 5 \cdot 7$

$588 = 2^2 \cdot 3 \cdot 7^2$

Método: Factorización en primos.

$\text{mcd}(588, 210) = 588/210 = 2_{168}$

$\hookrightarrow 588 = 210 \cdot 2 + 168$

$\text{mcd}(210, 168) = 210/168 = 1_{42}$

$\hookrightarrow 210 = 168 + 42$

$\text{mcd}(168, 42) = 168/42 = 4_0$

$\hookrightarrow 168 = \cancel{168}\,42 \cdot 4 + 0$

$\text{mcd}(42, 0) = 42$

Metodo: Algoritmo Euclides.