

Práctica 1: Determinación experimental de la complejidad temporal de un algoritmo.

Blancas Pérez Bryan Israel

Escuela Superior de Cómputo.
Instituto Politécnico Nacional, México
orionmunecaycanica@gmail.com

Resumen: Realizar los algoritmos convenientes para resolver los problemas planteados, y analizar de manera experimental, el número de operaciones que realizan éstos, para poder graficar los resultados y determinar su complejidad computacional.

Palabras Clave: Complejidad computacional, complejidad del algoritmo, operaciones, tiempo de ejecución.

1. Introducción

Un algoritmo, es un conjunto de instrucciones bien definidas, ordenadas y finitas que permiten llevar a cabo una actividad. Los algoritmos nos ayudan a resolver problemas de todo tipo, de manera que no haya lugar a dudas en el procedimiento.[1]

Analizar un algoritmo es de vital importancia, ya que de ese análisis se obtendrá que tan eficiente es; particularmente en nuestro campo las ciencias de la computación, se desea que el algoritmo use la menor cantidad de recursos del computador (tiempo y memoria).

En esta práctica se desarrollarán algoritmos para resolver los problemas planteados y posteriormente, se analizarán de manera experimental, graficando el tiempo de ejecución dependiendo de la entrada dada.

2. Conceptos Básicos

Notación θ : La notación θ se puede interpretar como el conjunto que acota a $f(n)$ por arriba y por abajo, θ es un ajuste asintótico para $f(n)$. Expresado matemáticamente sería:

Sea $g(n)$ una funcion, entonces

$$\theta(g(n)) = \{f(n) \mid \exists C_0, C_1, n_0 > 0 \text{ talque } 0 \leq C_0 g(n) \leq f(n) \leq C_1 g(n) \forall n \geq n_0\}$$

Notación O : La notación O se puede interpretar como el conjunto que acota a $f(n)$ por arriba. Expresado matemáticamente sería:

Sea $g(n)$ una funcion, entonces

$$O(g(n)) = \{f(n) \mid \exists C, n_0 > 0 \text{ talque } 0 \leq f(n) \leq C g(n) \forall n \geq n_0\}$$

Notación Ω : La notación Ω se puede interpretar como el conjunto que acota a $f(n)$ por abajo. Expresado matemáticamente sería:

Sea $g(n)$ una funcion, entonces

$$\Omega(g(n)) = \{f(n) \mid \exists C, n_0 > 0 \text{ talque } 0 \leq C g(n) \leq f(n) \forall n \geq n_0\}$$

En esta práctica se desarrollarán dos algoritmos. El primero de ellos, es un algoritmo para hacer la suma binaria de dos arreglos y el segundo es el algoritmo de Euclides para encontrar el MCD de un par de números, los cuales son pares de la serie de Fibonacci.

3. Experimentación y Resultados

Ejercicio 1.

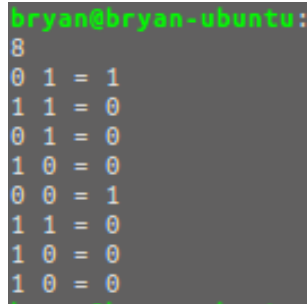
Desarrollar e implementar un algoritmo de *Suma* que sume dos enteros en notación binaria guardados en dos arreglos unidimensionales A y B de tamaño n , considerando $k = \log_2(n)$.

Código fuente del algoritmo de *Suma*:

```
1  int suma(int a[], int b[], int n){
2      int c[n], cont=1, i, acarreo=0;
3      cont++;
4      for (i=n-1; i>=0; i--, cont++){
5          c[i]=a[i]+b[i]+acarreo;  cont++;
6          if (c[i]==2){              cont++;
7              c[i]=0;                cont++;
8              acarreo=1;              cont++;
9          }
10         else if (c[i]==3){          cont++;
11             c[i]=1;                cont++;
12             acarreo=1;              cont++;
13         }
14         else{                        cont++;
15             acarreo=0;              cont++;
16         }
17     }
18     return cont;
19 }
```

La función suma, recibe dos arreglos (A y B) y el tamaño de éstos (n). Luego mediante un *for*, va recorriendo los arreglos de derecha a izquierda, sumando los números correspondientes. Dentro del *for* hay dos condiciones para controlar el acarreo que se pueda generar al momento de hacer la suma. Cada que se encuentra el resultado de la suma, se guarda en la correspondiente posición del arreglo C. El contador *cont*, como lo acordamos en clase, funge como el tiempo de ejecución.

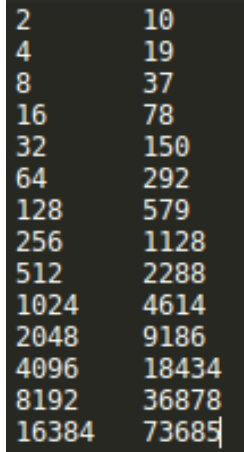
Programa en ejecución



```
bryan@bryan-ubuntu:
8
0 1 = 1
1 1 = 0
0 1 = 0
1 0 = 0
0 0 = 1
1 1 = 0
1 0 = 0
1 0 = 0
```

Figura 1: Ejecución del programa 1.

En la figura 1, se aprecia que el programa recibe como entrada el número, potencia de dos, que se interpretará como el tamaño de los arreglos. Como salida en la terminal, se muestra las sumas echas por el programa, empezando por los bit menos significativos de ambos arreglos.



2	10
4	19
8	37
16	78
32	150
64	292
128	579
256	1128
512	2288
1024	4614
2048	9186
4096	18434
8192	36878
16384	73685

Figura 2: output en archivo del programa 1.

En la figura 2, se muestra la salida del programa 1 en el archivo. La primer columna representa el tamaño de los arreglos sumados y la segunda salida representa el tiempo de ejecución (cont) que se requirió para terminar exitosamente el programa.

Gráfica de resultado.

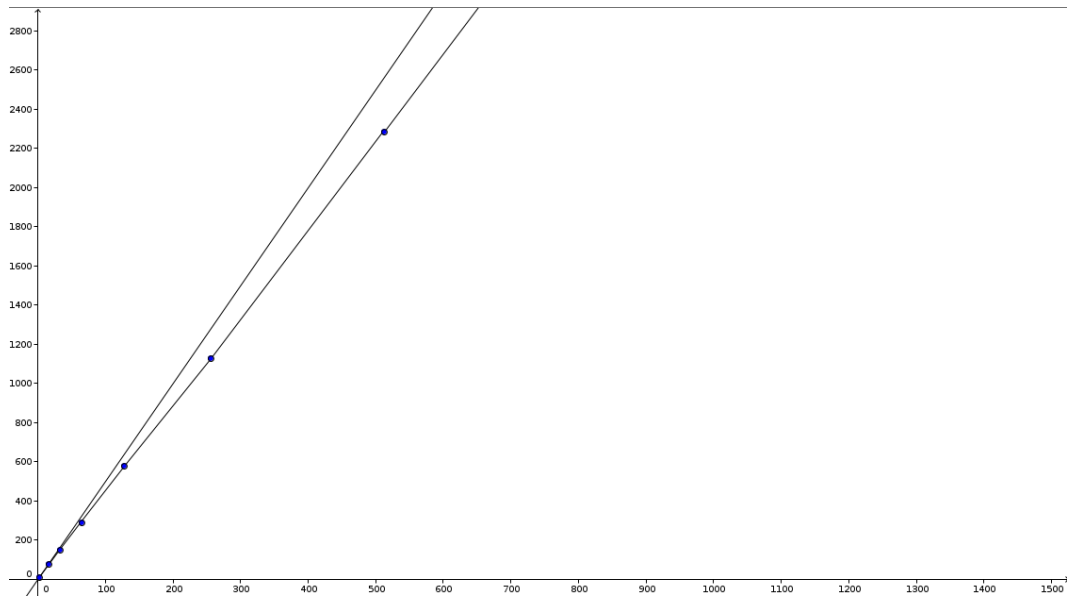


Figura 3: Gráfica del ejercicio 1.

En la figura 3, se muestra la gráfica $T(n)$ vs n , en la que los puntos marcados corresponden a los datos obtenidos en el archivo de la figura 2. La práctica pide proponer una función que acote por arriba a $T(n)$, en este caso la función propuesta es $y = 5x$, la cual se puede apreciar siempre por encima de $T(n)$ en la figura 3.

Conclusiones del programa.

Este es un algoritmo fácil de desarrollar, lo interesante desde mi punto de vista, fue ver el comportamiento que tiene dependiendo de la entrada, es decir, como se aprecia en la gráfica de la figura 3, este algoritmo es de orden lineal.

Ejercicio 2.

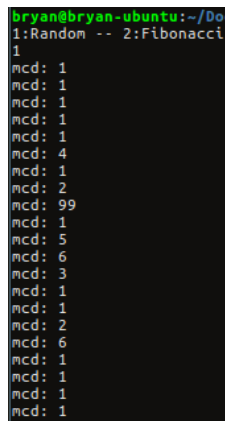
Implementar el algoritmo de *Euclides* para encontrar el MCD de dos números enteros positivos m y n . En la ejecución existen dos modos, el primero es elegir dos números aleatorios entre 0 y 1000, y la segunda es encontrar el MCD de los números de la serie de Fibonacci.

Código fuente del algoritmo de *Euclides*:

```
1 int euclides(int m, int n){
2     int r, cont=1;
3     while (n!=0){
4         cont++;
5         r=m%n;   cont++;
6         m=n;     cont++;
7         n=r;     cont++;
8     }
9     printf("mcd: _%d\n",m); cont++;
10    return cont;
11 }
```

La función `euclides`, recibe como parámetros dos números enteros positivos, a los cuales les calcula el MCD. EL contador (`cont`) utilizado funge como tiempo de ejecución. La función retorna el tiempo que se tardó en ejecutarse para que posteriormente ese dato pueda ser guardado en un archivo.

Programa en ejecución



```
bryan@bryan-ubuntu:~/Doc
1:Random -- 2:Fibonacci
1
mcd: 1
mcd: 1
mcd: 1
mcd: 1
mcd: 1
mcd: 4
mcd: 1
mcd: 2
mcd: 99
mcd: 1
mcd: 5
mcd: 6
mcd: 3
mcd: 1
mcd: 1
mcd: 2
mcd: 6
mcd: 1
mcd: 1
mcd: 1
mcd: 1
```

Figura 4: Ejecución del programa 2.

En la figura 4, se muestra que el programa recibe como entrada el número correspondiente a la elección del usuario. Como salida en la terminal, se muestra el MCD de los dos números analizados.

```
0 1 6
1 1 6
1 2 10
2 3 14
3 5 18
5 8 22
8 13 26
13 21 30
21 34 34
34 55 38
55 89 42
89 144 46
144 233 50
233 377 54
377 610 58
610 987 62
987 1597 66
```

Figura 5: Output en archivo del programa 2.

En la figura 4, se muestra la salida del programa 2 en el archivo. Las primeras dos columnas son los números a los que se les busco el MCD y la tercer columna representa el tiempo de ejecución (cont) que se requirió para calcular el MCD.

Gráfica de resultado.

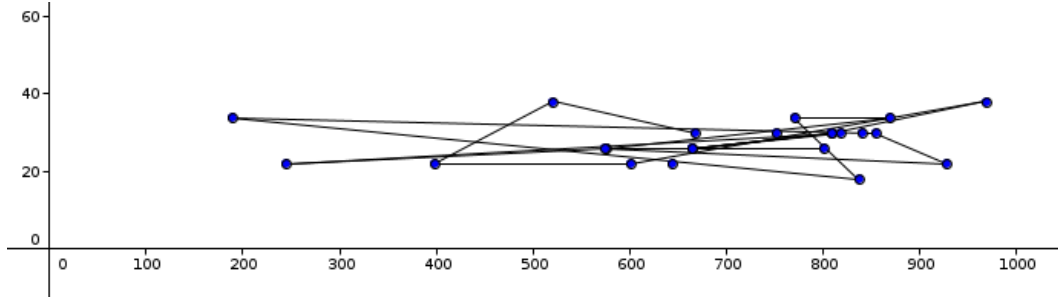


Figura 6: Gráfica 1 del ejercicio 2.

En la figura 6, se muestra la gráfica $T(n) vs n$, en la que los puntos marcados corresponden a los datos obtenidos en el archivo de la ejecución del programa 2 en modo random, tomando la coordenada sobre n (eje horizontal) como el número más grande de ambos.

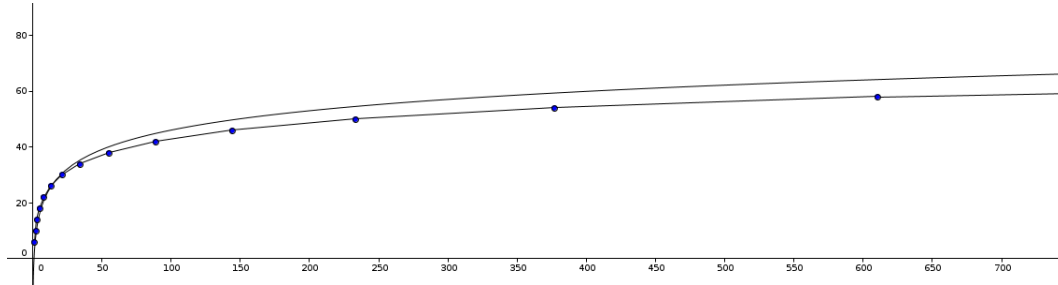


Figura 7: Gráfica 2 del ejercicio 2.

En la figura 7, se muestra la gráfica $T(n) vs n$, en la que los puntos marcados corresponden a los datos obtenidos en el archivo de la figura 5, tomando la coordenada sobre n (eje horizontal) como el número más grande de ambos. La práctica pide proponer una función que acote por arriba a $T(n)$, en este caso la función propuesta es $y = 10\log(x)$, la cual se puede apreciar siempre por encima de $T(n)$ en la figura 7.

Conclusiones del programa.

Es interesante notar del comportamiento de un algoritmo en el peor de los casos. En el caso del algoritmo de Euclides, el peor de los casos se presenta cuando se analizan los números de la serie de Fibonacci. En ese caso el algoritmo se comporta de manera logarítmica, sólo hace ver la gráfica para poder percatarse de ello.

4. Conclusiones

Al momento de desarrollar los algoritmos no tenía ni idea del comportamiento que tendrían, pero al obtener los datos y graficarlos me di cuenta de que analizar un algoritmo nos brinda una forma de expresar el comportamiento de este mediante funciones matemáticas, y así, poder determinar cual algoritmo conviene más para la solución de un determinado problema. En lo general, no tuve problemas al desarrollar la práctica, ya que los problemas planteados eran fáciles de resolver. Al final los resultados fueron los esperados, una función lineal para el ejercicio 1 y una función logarítmica para el ejercicio 2, y eso me indica que los algoritmos fueron desarrollados e implementados de forma correcta.

5. Anexo

Resolver el siguiente problema.

El siguiente algoritmo, es un algoritmo de ordenamiento llamado por selección (Select-Sort). Calcula el orden de complejidad en el peor de los casos.

```
1 Select-Sort(A[0], ..., n-1)
2   for j <- 0 to j <= n-2 do
3     k <- j
4     for i <- j+1 to i <= n-1 do
5       if A[i] < A[k] then
6         k <- i
7     intercambia(A[j], A[k])
```

Análisis del Algoritmo		
Línea de Código	Tiempo Ejecución	Número de Ejecuciones
2	C1	n
3	C2	$n - 1$
4	C3	$\sum_0^{n-1} ti$
5	C4	$\sum_0^{n-1} (ti - 1)$
6	C5	$\sum_0^{n-1} (ti - 1)$
7	C6	$n - 1$

Cuadro 1: Análisis del algoritmo

Encontrando el orden de complejidad.

$$T(n) = C1n + C2(n-1) + C3 \sum_0^{n-1} ti + C4 \sum_0^{n-1} (ti-1) + C5 \sum_0^{n-1} (ti-1) + C6(n-1).$$

Pero.

Encontrando ti		
j	i	ti
0	$1 \leq i \leq n - 1$	n
1	$2 \leq i \leq n - 1$	$n - 1$
2	$3 \leq i \leq n - 1$	$n - 2$
...
j	.	$n - j$

Cuadro 2: Encontrando ti

Encontrar.

$$T(n) = C1n + C2(n-1) + C3 \sum_0^{n-1} (n - j) + C4 \sum_0^{n-1} (n - j - 1) + C5 \sum_0^{n-1} (n - j - 1) + C6(n-1).$$

$$T(n) = C1n + C2(n-1) + C3 \sum_0^{n-1} (n) - C3 \sum_0^{n-1} (j) + C4 \sum_0^{n-1} (n) - C4 \sum_0^{n-1} (j) - C4 \sum_0^{n-1} (1) + C5 \sum_0^{n-1} (n) - C5 \sum_0^{n-1} (j) - C5 \sum_0^{n-1} (1) + C6(n-1).$$

$$T(n) = C_1n + C_2(n-1) + C_3(n^2 - (n)(n+1)/(2)) + (C_4+C_5) (n^2 - (n)(n+1)/2 - n) + C_6(n-1).$$

$$T(n) = C_1n + C_2(n-1) + C_3(n^2 - (n^2 + n)/(2)) + (C_4+C_5) (n^2 - (n^2 + n)/2 - n) + C_6(n-1).$$

Como se aprecia, el máximo exponente de n es 2, es decir, n^2 . Por lo tanto la ecuación queda de la siguiente manera.

$$T(n) = An^2 + Bn + C.$$

$$\text{por lo tanto } T(n) \in \theta(n^2)$$

6. Bibliografía

[1] es.wikipedia.org/wiki/Algoritmo