Análisis de Algoritmos, Sem: 2018-2, 3CV1, Práctica 1, 22 de Agosto de 2018

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

Reyes Valenzuela Alejandro, Gutiérrez Povedano Pablo.

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
*areyesv*11@*gmail.com*, *gpovedanop*@*gmail.com*

Resumen: En este reporte se explicará de manera precisa cómo fueron implementados los algoritmos de la práctica, al igual que la generación de las gráficas de las mismas, y qué nos dicen en cuanto a la complejidad temporal de estos.

Palabras Clave: Tiempo, Algoritmo, Complejidad, Notación.

1 Introducción

Un algoritmo es definido comúnmente como una serie de pasos que son realizados con un propósito en específico. Cada vez que sigamos los pasos de un algoritmo, es importante que lleguemos al mismo resultado, también es importante identificar el principio y el fin de éste, y mayormente, es necesario describir tres partes: Entrada, Proceso y Salida.

Es importante analizar estos algoritmos para poder determinar su complejidad y juzgar cuál es el mejor o el peor para los escenarios que se nos puedan presentar al codificarlos; siendo éste el objetivo de la práctica, codificar algortimos y basar su complejidad con base al número de veces que una línea es ejecutada (esto se tratará más adelante en la Sección 2).

2 Conceptos Básicos

De entrada comenzaremos por definir las notaciones vistas en clase las cuáles nos servirán para identificar el mejor caso, el peor caso, o cuando el peor caso y el mejor caso no son diferentes entre sí.

Notación Θ: Una cota ajustada asintótica es una función que sirve de cota tanto superior como inferior de otra función cuando el argumento tiende a infinito. Usualmente se utiliza la notación Θ(g(n)) para referirse a las funciones acotadas por la función g(n).

Una función es Θ(g(n)) sí y sólo sí existe para toda n las constantes C_1 , C_2 , $n_0 > 0$, tales que $0 \leq C_1 g(n) \leq f(n) \leq C_2 g(n)$ para toda $n \leq n_0$.

En esta notación consideramos que tanto el mejor caso como el peor caso tienen el mismo nivel de dificultad, por lo que acotamos nuestra función por ambos lados para determinar si nuestra función pertenece a Θ, además decimos que g(n) es un ajuste asintático para f(n).

Notación *O*: Una cota superior asintótica es una función que sirve de cota superior de otra función cuando el argumento tiende a infinito.

Una función es $O(g(n))$ sí y sólo sí existe para toda n las constantes C_1 , $n_0 > 0$, tales que $0 \leq f(n) \leq C_1 g(n)$ para toda $n \leq n_0$.

En esta notación solo consideramos el peor caso.

Notación Ω: Una cota inferior asintótica es una función que sirve de cota inferior de otra función cuando el argumento tiende a infinito.

Una función es Ω(g(n)) sí y sólo sí existe para toda n las constantes C_1 , $n_0 > 0$, tales que $0 \leq C_1 g(n) \leq f(n)$ para toda $n \leq n_0$.

En esta notación solo consideramos el mejor caso.

Como podemos observar, dependerá de la notación utilizada la cota que encontraremos, es decir, nos concentraremos en los escenarios que se le presenten al algoritmos; si buscamos la cota inferior, entonces nos concentramos en el mejor escenario, si buscamos la superior, nos concentramos en el peor escenario y si buscamos ambas, el algoritmo no presenta un mejor o peor escenario como tal, es decir todos los casos a tratar mantienen la misma complejidad temporal.

Algoritmos desarrollados

En esta práctica se desarrollaron dos algoritmos:

Suma Binaria: Desarrollar e implementar un algoritmo Suma que sume dos enteros en notación binaria bajo las siguientes consideraciones: Dos arreglos

unidimensionales A de tamaño n y B de tamaño m con k = $\log_2(n)$ y t = $\log_2(m)$ (es decir, que sean potencias de 2) almacenarán los números a sumar. La suma se almacenará en un arreglo C.

Para este algoritmo, se optó por rellenar los arreglos A y B con unos y ceros de manera aleatoria, para posteriormente realizar la 'suma', la cual solamente comparará los valores de los acarreos y de los valores en la posición i para determinar el valor del próximo acarreo de entrada y de la suma actual.

A continuación se muestra el algoritmo en cuestión al igual que una prueba de escritorio con dos bits:

Suma(A,B): (A,B arreglos):

```
for i ← n-1 to ≥ 0 do
    if A[i] == 0 and B[i] == 0 and carry == 0 then
        C[i] ← 0
        carry ← 0
    else if A[i] == 0 and B[i] == 0 and carry == 1 then
        C[i] ← 1
        carry ← 0
    else if A[i] == 1 and B[i] == 0 or A[i] == 0 and B[i] == 1 and
carry == 0 then
        C[i] ← 1
        carry ← 0
    else if A[i] == 1 and B[i] == 0 or A[i] == 0 and B[i] == 1 and
carry == 1 then
        C[i] ← 0
        carry ← 1
    else if A[i] == 1 and B[i] == 1 and carry == 0 then
        C[i] ← 0
        carry ← 1
    else if A[i] == 1 and B[i] == 1 and carry == 1 then
        C[i] ← 1
        carry ← 1
```

Finalmente procedemos a desplegar la suma (almacenada en el arreglo C). Por ejemplo, para A={0,1} y B={0,1} se tiene el siguiente funcionamiento: (tomando en cuenta n=2):

for i $\leftarrow$ 1 to $\geq$ 0 do
....
else if A[i] == 1 and B[i] == 0 or A[i] == 0 and B[i] == 1 and carry == 0 then (esta es la condición que cumple nuestro ejemplo en ambos casos)
C[i] $\leftarrow$ 1
carry $\leftarrow$ 0
....

Debido a esto al ir nuestro for de 0 a 1, el arreglo C tiene dos espacios para almacenar nuestra suma, por lo que en ambos casos se llenará con un 1, por lo que nuestro arreglo C quedará conformado de la siguiente manera C={1,1}.

Algoritmo de Euclides: Implementar el algoritmo de Euclides para encontrar el mcd de dos numeros enteros positivos m y n.

```
Euclides(m,n):
    while n ≠ 0 do
        r ← m mod n
        m ← n
        n ← r
    return m
```

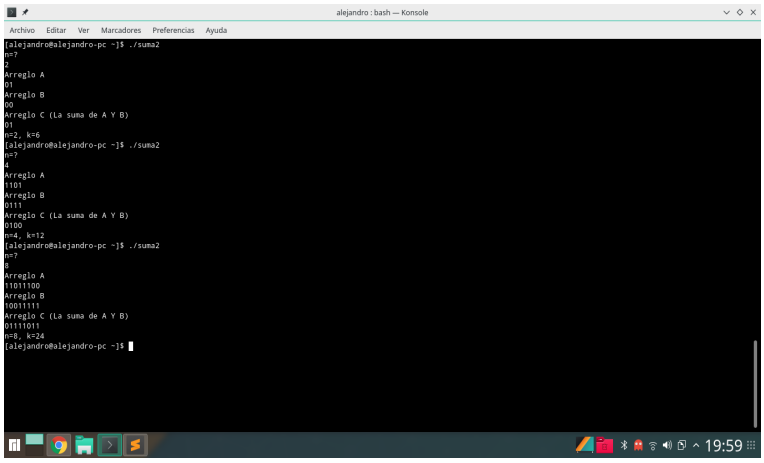
A continuación mostraremos el funcionamiento del algoritmo con m=12 y n=6:

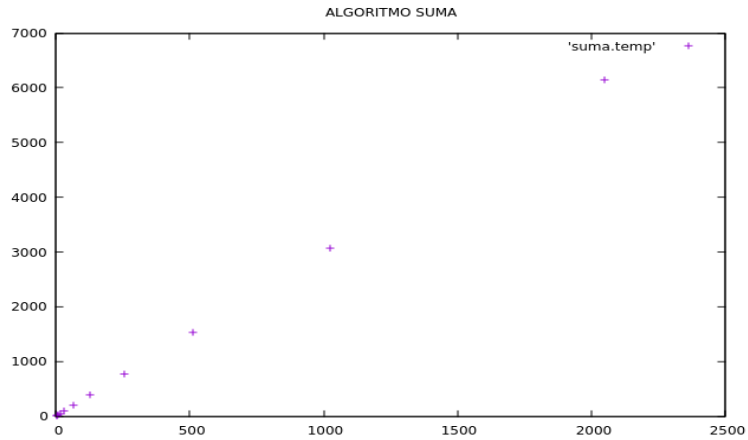
```
Euclides(12,6):
    while 6 ≠ 0 do
        r ← 12 mod 6 ← 0
        m ← n ← 6
        n ← r ← 0
    return m
```

Al terminar el algoritmo (antes de que n valiera cero), nos devolverá el MCD de 12 y 6, en el que en este caso es 6.

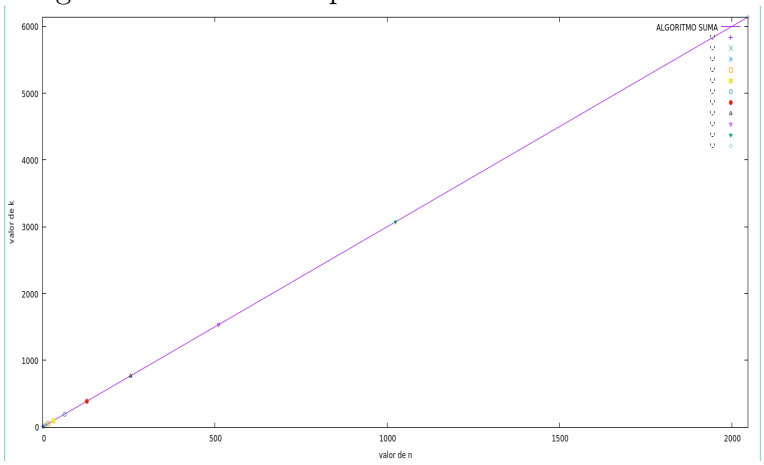
3 Experimentación y Resultados

Suma Binaria: Se adjunta foto del funcionamiento tras compilación del algoritmo y la gráfica generada con valores desde 2^1 hasta 2^{11}.



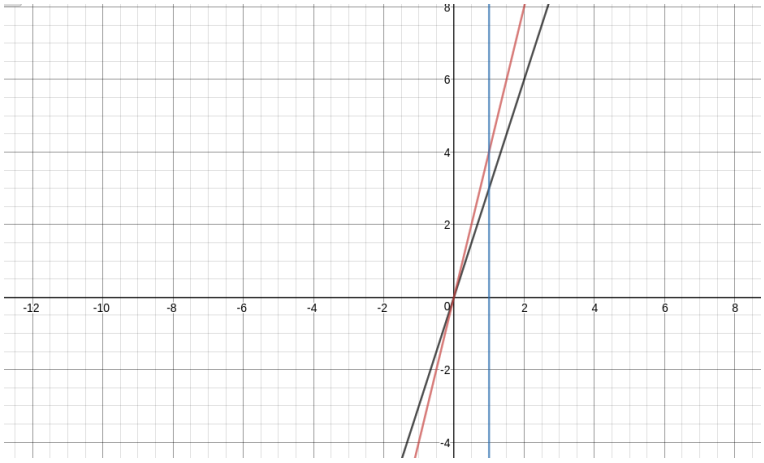


Al asignar una constante k que se incrementara cada vez que una línea de código era ejecutada o por cada iteración nos dimos cuenta que estos podían ser unidos por una recta, por lo que nos damos cuenta de que los puntos presentados en la gráfica anterior son parte de la misma función.



Ahora se propone la función g(n)=4n para demostrar que Suma ϵ O(g(n)) y g(n) sea mínima, en el sentido de que si suma ϵ O(h(n)), entonces $g(n)$ ϵ $(h(n))$.

Esto se puede observar de manera gráfica, siendo la gráfica roja la función suma, 4n la negra y n_0=1 (el primer punto donde se empieza a acotar la función) la recta azul.



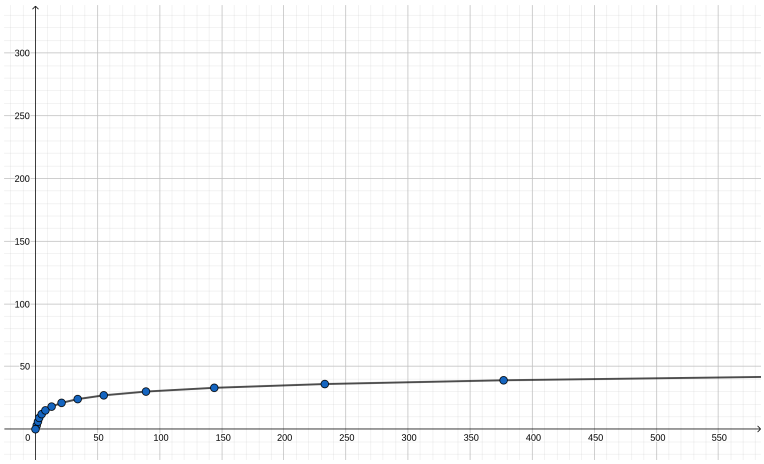
Conclusión Reyes Valenzuela: Podemos observar que la complejidad de este algorimto crece con base al número de bits que tenga el arreglo, por este motivo decimos que poseé una complejidad lineal.

Conclusión Gutiérrez Povedano: Ya que el tamaño de los arreglos varía en potencias de 2 y que tanto en el mejor caso como en el peor se deben recorre los arreglos por completo el orden de complejidad para este algoritmo es $\theta(n)$.

Euclides: Se adjunta imágen correspondiente al funcionamiento del algoritmo en código.

```
[alejandro@alejandro-pc ~]$ ./euc
m=?
16
n=?
4
k=3
el MCD DE m:16 y n:4 es:4
[alejandro@alejandro-pc ~]$ ./euc
m=?
37
n=?
12
k=6
el MCD DE m:37 y n:12 es:1
[alejandro@alejandro-pc ~]$ ./euc
m=?
89
n=?
55
k=27
el MCD DE m:89 y n:55 es:1
[alejandro@alejandro-pc ~]$ ./euc
m=?
2584
n=?
610
k=15
el MCD DE m:2584 y n:610 es:2
[alejandro@alejandro-pc ~]$ ./euc
m=?
21
n=?
13
k=18
el MCD DE m:21 y n:13 es:1
[alejandro@alejandro-pc ~]$
```

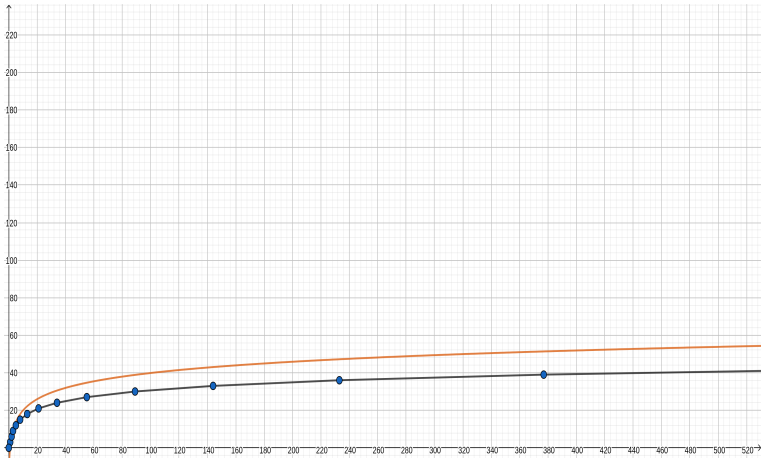
Para graficar consideramos el peor de los casos, el cu'al es que los números ingreados a la función correspondan a la serie de Fibonacci, en los cuales no existen un MCD, por lo que la función siempre nos retornará un 1, (k vs n, siendo n el mayor nmero ingresado a la función, los puntos representan los puntos encontrados en la relación n/tiempo, donde se utilizaron los primeros 16 términos (0-987)).



Podemos observar que nuestros valores de constante k aumentan de manera logarítmica.

Ahora se propone la función $6\log_2(n)$para demostrar que Euclides ϵ O(g(n)) y g(n) sea mínima, en el sentido de que si Euclides ϵ O(h(n)), entonces $g(n)$ ϵ $O(h(n))$.

Para ello graficamos la función (expresada en naranja) contra los puntos generados por la función de Euclides:



Como podemos observar, la grfica naranja acota a la funcin de Euclides por arriba, por lo que podemos decir con certeza que entonces $g(n)$ ϵ $O(h(n))$.

Conclusión Reyes Valenzuela: En este algoritmo tenemos que nuestros puntos crecen de manera logarítmica, por lo que podemos decir que nuestra función tiene un nivel de complejidad menor que el anterior, y esto se puede determinar gracias a las formas de gráficas generadas.

Conclusión Gutiérrez Povedano: Al usarse dos números consecutivos de la sucesión Fibonacci en el algoritmo de Euclides se obtiene el peor caso del algoritmo y ya que después de 2 iteraciones del programa el algoritmo pasa de euc(Fi,Fi-1) a euc(Fi-1,Fi-2) que es dividir a la mitad el tamao del problema, por lo que la complejidad del algoritmo disminuye de manera considerable.

4 Conclusiones

Conclusión General: Esta práctuca nos sirvió para comparar dos algoritmos para calcular su complejidad, los cuáles fueron muy distintos de graficar, ya que la primera quedó de manera lineal (como recta) mientras que en la otra se nos generó una que se asemeja a una función logarítmica. El hecho de que hayamos podido determinar esto mediante la experimentación nos indica que

hemos entendido los conceptos vistos en clase de manera correcta.

Conclusión Reyes Valenzuela: En mi caso tuve complicaciones con el primer programa, ya que la suma binaria funciona de manera distinta a la suma normal, por lo que debíamos de controlar el valor de la suma y el acarreo que se pudiese generar, esto no sucedió con el segundo programa, ya que de alguna manera este algorítmo ya está definido.

Conclusión Gutiérrez Povedano: Al termino de la practica se puede concluir que el orden de complejidad de un algoritmo puede ser el mismo para el peor o mejor de los casos como en el ejercicio 1 o distinto pero nunca mayor que el peor de los casos como en el ejercicio 2, ademas de que la complejidad del algoritmo puede estimarse facilmente analizando la cantidad y las condiciones de los ciclos que se implementan en dicho algoritmo.

5 Anexo

El siguiente algoritmo, es un algoritmo de ordenamiento llamado por selección (SelectSort(A)).
i) Muestre mediante un ejemplo, el funcionamiento del algoritmo.
ii) Calcular el orden de complejidad en el peor de los casos mediante el cálculo temporal línea por línea.

Select-Sort(A[0, . . . , n − 1])

```
for j ← 0 to j ≤ n − 2 do
    k ← j
        for i ← j + 1 to i ≤ n − 1 do
            if A[i] < A[k] then
                k ← i
    Intercambia (A[j],A[k])
```

Para mostrar el funcionamiento usaremos un arreglo de tamaño 4, siendo $A\{12, 31, 5, 100\}$ el arreglo que usaremos para la prueba de escritorio (Sobreentendiéndose el hecho de que n=4).

```
for j ← 0 to j ≤ 4 − 2 do
    k ← j
        for i ← j + 1 to i ≤ 4 − 1 do
            if A[i] < A[k] then
```

```
                    k ← i
Intercambia (A[j],A[k])
```

Tras hacer la suma tenemos que:

```
for j ← 0 to j ≤ 2 do
    k ← j
        for i ← j + 1 to i ≤ 3 do
            if A[i] < A[k] then
                k ← i
    Intercambia (A[j],A[k])
```

Iteración 1:

```
for j ← 0 to j ≤ 2 do
    k ← 0
        for i ← 0 + 1 to i ≤ 3 do
          if A[1] < A[0] then (el if no se cumple, ya que 12 < 31)
                k ← i (no se hace)
    Intercambia (A[j](12),A[k](12)) (se queda igual)
```

Iteración 2:

```
for j ← 1 to j ≤ 2 do
    k ← 1
        for i ← 1 + 1 to i ≤ 3 do
          if A[2] < A[1] then (el if se cumple, ya que 5 < 31)
                k ← 2 (se hace)
    Intercambia (A[j](31),A[k](5)) (cambia de posición)
```

Iteración 3:

```
for j ← 2 to j ≤ 2 do
    k ← 2
        for i ← 2 + 1 to i ≤ 3 do
          if A[3] < A[2] then (el if no se cumple, ya que 5 < 100)
                k ← i (no se hace)
    Intercambia (A[j](5),A[k](5)) (se queda igual)
```

Por lo que nuestro arreglo queda acomodado de la siguiente manera A$\{5, 12, 31, 100\}$

A continuacón calcularemos el nivel de complejidad de manera analítica, para ello asignamos cosntantes y determinamos el número de veces que las líneas de código son ejecutadas:

for j $\leftarrow$ 0 to j $\leq$ n $-$ 2 do	C_1	n
k $\leftarrow$ j	C_2	$n-1$
for i $\leftarrow$ j + 1 to i $\leq$ n $-$ 1 do	C_3	$\sum_{i=0}^{n-1} t_i$
if A[i] $<$ A[k] then	C_4	$\sum_{i=0}^{n-1}(t_i - 1)$
k $\leftarrow$ i	C_5	$\sum_{i=0}^{n-1}(t_i - 1)$
Intercambia (A[j],A[k])	C_6	$n-1$

Calcularemos el valor de las sumatorias:

i	t_i
1	n-1
2	n-2
3	n-3
4	n-4
5	n-5

Por lo que tenemos que $t_i = n - i$

Ahora, el algoritmo está definido por $T(n)$, quién a su vez está conformado por:

$$T(n) = C_1{*}n + C_2{*}(n-1) + C_3{*}\sum_{i=0}^{n-1} t_i + C_4{*}\sum_{i=0}^{n-1}(t_i - 1) + C_5{*}\sum_{i=0}^{n-1}(t_i - 1) + C_6{*}(n-1)$$

Sustituyendo:

$$T(n) = C_1{*}n + C_2{*}(n-1) + C_3{*}\sum_{i=0}^{n-1}(n-i) + C_4{*}\sum_{i=0}^{n-1}(n-i-1) + C_5{*}\sum_{i=0}^{n-1}(n-i-1) + C_6{*}(n-1)$$

Como $\sum_{i=0}^{n-1}(n-i-1) = \frac{1}{2}(n-1)n$ y $\sum_{i=0}^{n-1}(n-i) = \frac{1}{2}(n+1)n$, entonces $T(n)$ queda de la siguiente forma:

$$T(n) = C_1{*}n + C_2{*}(n-1) + C_3{*}\tfrac{1}{2}(n+1)n + C_4{*}\tfrac{1}{2}(n-1)n + C_5{*}\tfrac{1}{2}(n-1)n + C_6{*}(n-1)$$

Desarrollando y agrupando términos semejantes (las constantes al multiplicarse dan como resultado otra constante, por lo que podemos representarlas en una sola):

$$T(n) = C_1{*}n + C_2{*}n - C_2 + C_3{*}\tfrac{n^2}{2} + C_3{*}\tfrac{n}{2} + C_4{*}\tfrac{n^2}{2} - C_4{*}\tfrac{n}{2} + C_5{*}\tfrac{n^2}{2} - C_5{*}\tfrac{n}{2} + C_6{*}n - C_6$$

$$T(n) = n^2(C_3 + C_4 + C_5) + n(C_1 + C_2 + C_3 + C_4 + C_5 + C_6) + (-C_2 - C_6)$$

Observamos que nuestro algoritmo en el peor de los casos se torna cuadrático, por lo que decimos que $SelectionSort \in O(n^2)$.

6 Bibliografía

Es.wikipedia.org. (2018). Cota ajustada asintótica. [online] Available at: https://es.wikipedia.org/wiki/Cota_ajustada_asintótica [Accessed 27 Aug. 2018].

Es.wikipedia.org. (2018). Cota inferior asintótica. [online] Available at: https://es.wikipedia.org/wiki/Cota_inferior_asintótica [Accessed 27 Aug. 2018].

Es.wikipedia.org. (2018). Cota superior asintt́ica. [online] Available at: https://es.wikipedia.org/wiki/Cota_superior_asintótica [Accessed 27 Aug. 2018].