Instituto Politécnico Nacional

Escuela Superior de Cómputo

Análisis de algoritmos

Ejercicio 2: Complejidad temporal y análisis de casos

Grupo: 3CM3

Alumno:
Ontiveros Salazar Alan Enrique

Profesor:
Franco Martínez Edgardo Adrián

9 de marzo de 2018

Ejercicio 2: Complejidad temporal y análisis de casos

3CM3
ESCOM-IPN

9 de marzo de 2018

1. Resultados útiles

1.1. Ciclo `for` genérico

Consideremos el siguiente ciclo `for`, donde $a, b \in \mathbb{Z}$, $a \leq b$ y $s \in \mathbb{N}$:

```
for(int i = a; i <= b; i += s){
    instruccion();
}
```

Sea x el número de veces que se ejecuta `instruccion()`. Vemos que la sucesión de valores que i va tomando es una progresión aritmética con primer término a y diferencia común s. Entonces:

$$i = a + s(k-1)$$

donde $k \in \mathbb{N}$. Por definición, se tiene que cumplir que $a \leq i \leq b$, es decir:

$$a \leq a + s(k-1) \leq b$$

De esa forma, x es simplemente el número de valores que puede tomar k. Simplificando ambos lados:

$$1 \leq k \leq \frac{b-a}{s} + 1$$

Pero como k es un entero positivo, entonces:

$$1 \leq k \leq \left\lfloor \frac{b-a}{s} + 1 \right\rfloor$$

De esa forma, $x = \left\lfloor \frac{b-a}{s} + 1 \right\rfloor = \left\lfloor \frac{b-a+1}{s} + \frac{s-1}{s} \right\rfloor = \left\lceil \frac{b-a+1}{s} \right\rceil$.

Tenemos que:

- `instruccion()` se ejecuta x veces. Supongamos que el costo de esta instrucción es c, entonces el costo total es cx.
- Hay una asignación inicial a la variable `i`.

- Supongamos que calcular `a` cuesta d.
- El número de comparaciones de `i` con `b` es igual al número de ejecuciones de `instruccion()` más una extra, es decir, $x + 1$.
- Si el costo de calcular `b` es e, entonces el costo total es e veces el número de comparaciones, es decir, $e(x + 1) = ex + e$.
- El número de incrementos a `i` es igual al número de ejecuciones de `instruccion()`, y si cada incremento cuesta 2 (suma y asignación), el costo total es $2x$.
- El número de saltos implícitos es el número de veces que se ejecuta la instrucción más uno, es decir, $x + 1$.

Por lo tanto, la complejidad temporal del `for` anterior es:

$$f_t(a, b, c, d, e, s) = (c + e + 4)x + d + e + 3 = (c + e + 4)\left\lceil \frac{b - a + 1}{s} \right\rceil + d + e + 3$$

Si en la condición del `for` tenemos `i <b`, lo podemos convertir a `i <= b-1`.

Si la declaración del `for` es `for(int i = b; i >= a; i -= s)`, entonces el conteo es equivalente al primero, pues solo estamos contando de reversa.

1.2. Variante geométrica 1

Ahora veamos qué pasa en el siguiente ciclo `for`, donde donde $a, b \in \mathbb{N}$, $a \leq b$ y $r \in \mathbb{N}$:

```
for(int i = a; i <= b; i *= r){
    instruccion();
}
```

Ahora i se comporta como una progresión geométrica con primer término a y razón r, es decir:

$$i = ar^{k-1}$$

donde $k \in \mathbb{N}$. De forma similar que en el primer caso, se tiene que cumplir que $a \leq i \leq b$, es decir, $a \leq ar^{k-1} \leq b$. Simplificamos y obtenemos $1 \leq k \leq \log_r\left(\frac{b}{a}\right) + 1$.

De esa forma, $x = \left\lfloor \log_r\left(\frac{b}{a}\right) + 1 \right\rfloor$ es el número de veces que se ejecutará `instruccion()`.

2. Complejidad temporal y espacial

Para los siguientes 5 algoritmos determine la función de complejidad temporal y espacial en términos de n. Considere las operaciones de: asignación, aritméticas, condicionales y saltos implícitos.

2.1. Algoritmo 1

```
for(i = 1; i < n; i++){
    for(j = 0; j < n - 1; j++){
        temp = A[j];                //1 asignación
        A[j] = A[j + 1];            //1 operación y 1 asignación
        A[j + 1] = temp;            //1 operación y 1 asignación
    }
}
```

Primero veamos el `for` interno, es un `for` genérico con parámetros $a = 0$, $b = n - 2$ y $s = 1$. El costo de calcular `a` es $d = 0$ y el de calcular `b` es $e = 1$ (una resta). Su bloque de instrucciones cuesta $c = 5$ operaciones, por lo tanto, la complejidad de este `for` es:

$$\begin{aligned} g_t(n) &= (5 + 1 + 4)\left\lceil \frac{(n-2) - 0 + 1}{1} \right\rceil + 0 + 1 + 3 \\ &= 10(n-1) + 4 \\ &= 10n - 6 \end{aligned}$$

Pero a la vez $g_t(n)$ es el costo de la ejecución del primer `for`, cuyos parámetros son $a = 1$, $b = n-1$ y $s = 1$. Esta vez el costo de calcular `a` y `b` son ambos 0, o sea, $d = e = 0$. Entonces la complejidad temporal es:

$$\begin{aligned} f_t(n) &= (g_t(n) + 0 + 4)\left\lceil \frac{(n-1) - 1 + 1}{1} \right\rceil + 0 + 0 + 3 \\ &= (10n - 6 + 4)(n-1) + 3 \\ &= (10n - 2)(n-1) + 3 \\ &= 10n^2 - 12n + 5 \end{aligned}$$

Pero cuando $n = 0$ solo tenemos 3 ejecuciones en vez de cinco, entonces:

$$f_t(n) = \begin{cases} 3 & \text{si } n = 0 \\ 10n^2 - 12n + 5 & \text{si } n \geq 1 \end{cases} \quad \square$$

La complejidad espacial es $f_e(n) = n + 3$, pues usamos el arreglo A de tamaño n, una variable extra `temp` y los dos iteradores.

2.2. Algoritmo 2

```
polinomio = 0;                                  //1 asignación
for(i = 0; i <= n; i++){
    polinomio = polinomio * z + A[n - i];       //1 asignación y 3 operaciones
}
```

Tenemos una asignación inicial, más un `for` genérico con parámetros $a = 0$, $b = n$, $c = 4$, $d = e = 0$ y $s = 1$. Entonces la complejidad temporal es:

$$\begin{aligned} f_t(n) &= 1 + (4 + 0 + 4)\left\lceil \frac{n - 0 + 1}{1} \right\rceil + 0 + 0 + 3 \\ &= 1 + 8(n + 1) + 3 \\ &= 8n + 12 \quad \square \end{aligned}$$

La complejidad espacial es $f_e(n) = n+3$, pues A es de longitud $n+1$, usamos la variable `polinomio` y un iterador.

2.3. Algoritmo 3

```
for i = 1 to n do
    for j = 1 to n do
        C[i,j] = 0;                                  //1 asignación
        for k = 1 to n do
            C[i,j] = C[i,j] + A[i,k]*B[k,j];     //1 asignación y 2 operaciones
```

Tenemos tres `for` anidados, vayamos del último al primero:

- Para este `for` tenemos como parámetros $a = 1$, $b = n$, $c = 3$, $d = e = 0$ y $s = 1$. Entonces:

$$\begin{aligned} h_t(n) &= (3 + 0 + 4)\left\lceil \frac{n - 1 + 1}{1} \right\rceil + 0 + 0 + 3 \\ &= 7n + 3 \end{aligned}$$

- Para este `for` tenemos como parámetros $a = 1$, $b = n$, $c = 1+h_t(n)$, $d = e = 0$ y $s = 1$. Entonces:

$$\begin{aligned} g_t(n) &= (1 + 7n + 3 + 0 + 4)\left\lceil \frac{n - 1 + 1}{1} \right\rceil + 0 + 0 + 3 \\ &= (7n + 8)(n) + 3 \\ &= 7n^2 + 8n + 3 \end{aligned}$$

- Finalmente, para el `for` externo tenemos como parámetros $a = 1$, $b = n$, $c = g_t(n)$, $d = e = 0$ y $s = 1$. Entonces, la complejidad temporal es:

$$\begin{aligned} f_t(n) &= (7n^2 + 8n + 3 + 0 + 4)\left\lceil \frac{n - 1 + 1}{1} \right\rceil + 0 + 0 + 3 \\ &= (7n^2 + 8n + 7)(n) + 3 \\ &= 7n^3 + 8n^2 + 7n + 3 \quad \square \end{aligned}$$

La complejidad espacial es simplemente $f_e(n) = 3n^2 + 3$, pues las matriz A, B y C son de $n \times n$ y tenemos tres iteradores.

2.4. Algoritmo 4

```
anterior = 1;                       //1 asignación
actual = 1;                           //1 asignación
while (n > 2){
    aux = anterior + actual;      //1 operación y 1 asignación
    anterior = actual;              //1 asignación
    actual = aux;                  //1 asignación
    n = n - 1;
}
```

El ciclo `while` mostrado es equivalente a un `for` genérico con la siguiente declaración: `for(int i = 3; i <= n; i++)`, y removiendo la línea `n = n - 1`, además de que hay que restar el costo $d+1$ (inicialización y costo de calcular `a`). Entonces la fórmula de la complejidad temporal del `while` queda como:

$$g_t(n) = (c+e+4)\left\lceil \frac{b-a+1}{s} \right\rceil + e + 2$$

Pero en este caso $a=3$, $b=n$, $c=4$, $e=0$ y $s=1$. Entonces:

$$\begin{aligned} g_t(n) &= (4+0+4)\left\lceil \frac{n-3+1}{1} \right\rceil + 0 + 2 \\ &= 8(n-2)+2 \\ &= 8n-14 \end{aligned}$$

Como hay dos asignaciones iniciales, la función de complejidad temporal es:

$f_t(n) = 2 + g_t(n) = 8n - 12$, pero como no podemos tener valores negativos:

$$f_t(n) = \begin{cases} 4 & \text{si } n = 0,1,2 \\ 8n-12 & \text{si } n \geq 3 \end{cases} \quad \square$$

La complejidad espacial es $f_e(n) = 3$, pues hay tres variables únicamente.

2.5. Algoritmo 5

```
for (i = n - 1, j = 0; i >= 0; i--, j++)
    s2[j] = s[i];
for (i = 0, i < n; i++)
    s[i] = s2[i];
```

El bloque anterior es equivalente a:

```
j = 0;                                  //1 asignación
for (i = n - 1; i >= 0; i--)
    s2[j] = s[i];                      //1 asignación
    j++;                              //1 operación y 1 asignación
for (i = 0; i < n; i++)
    s[i] = s2[i];                      //1 asignación
```

El primer `for` tiene como parámetros $a = 0$, $b = n - 1$, $c = 3$, $d = 1$ (una resta), $e = 0$ y $s = 1$. Entonces:

$$\begin{aligned} g_t(n) &= (3 + 0 + 4) \left\lceil \frac{(n-1) - 0 + 1}{1} \right\rceil + 1 + 0 + 3 \\ &= 7n + 4 \end{aligned}$$

El segundo `for` tiene como parámetros $a = 0$, $b = n - 1$, $c = 1$, $d = e = 0$ y $s = 1$. Entonces:

$$\begin{aligned} h_t(n) &= (1 + 0 + 4) \left\lceil \frac{(n-1) - 0 + 1}{1} \right\rceil + 0 + 0 + 3 \\ &= 5n + 3 \end{aligned}$$

La complejidad temporal total es la de una asignación inicial más la de los dos `for`, por lo tanto:

$$\begin{aligned} f_t(n) &= 1 + g_t(n) + h_t(n) = 1 + 7n + 4 + 5n + 3 \\ &= 12n + 8 \quad \square \end{aligned}$$

La complejidad espacial es $f_e(n) = 2n + 2$, asumiendo que los arreglos $s2$ y s tienen tamaño n, más los dos iteradores.

3. Aproximación de una función que cuente el número de "prints"

3.1. Algoritmo 6

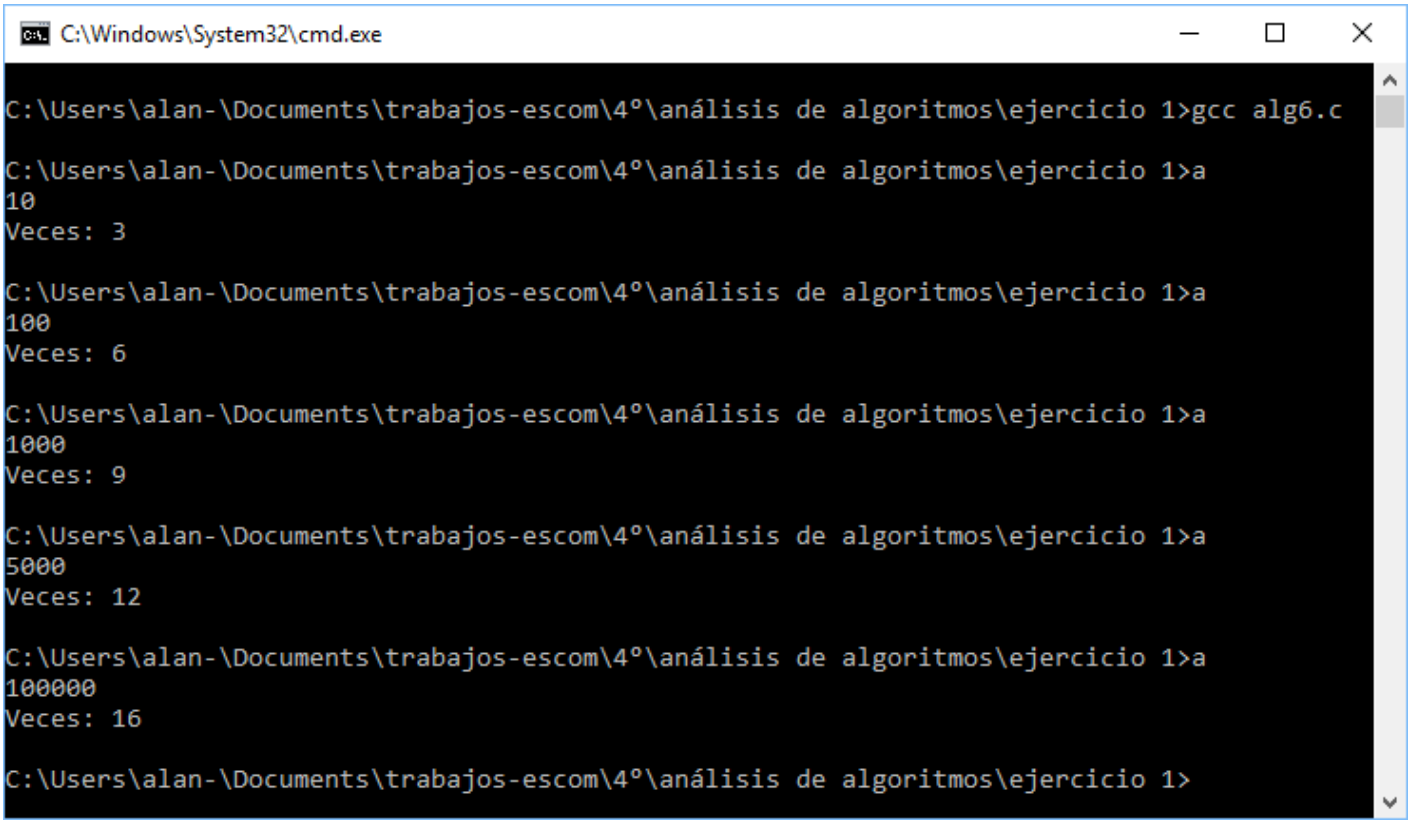
```c
#include <stdio.h>

int main(){
    int n;
    scanf("%d", &n);
    int count = 0;
    for(int i = 10; i < n * 5; i *= 2){
        //printf("\"Algoritmos\"\n");
        count++;
    }
    printf("Veces: %d\n", count);
    return 0;
}
```

Tenemos la variante geométrica 1 del `for` con parámetros $a = 10$, $b = 5n - 1$ y $r = 2$. Por lo tanto, la función que indica el número de impresiones de la palabra "Algoritmos" es:

$$f(n) = \left\lfloor \log_2 \left(\frac{5n-1}{10} \right) + 1 \right\rfloor \quad \square$$

Comprobación:



n	Número de impresiones reales	Número de impresiones según $f(n)$
10	3	$\left\lfloor \log_2\left(\frac{5(10)-1}{10}\right)+1 \right\rfloor = \lfloor \log_2(4.9)+1 \rfloor = \lfloor 3.29 \rfloor = 3$
100	6	$\left\lfloor \log_2\left(\frac{5(100)-1}{10}\right)+1 \right\rfloor = \lfloor \log_2(49.9)+1 \rfloor = \lfloor 6.64 \rfloor = 6$
1000	9	$\left\lfloor \log_2\left(\frac{5(1000)-1}{10}\right)+1 \right\rfloor = \lfloor \log_2(499.9)+1 \rfloor = \lfloor 9.96 \rfloor = 9$
5000	12	$\left\lfloor \log_2\left(\frac{5(5000)-1}{10}\right)+1 \right\rfloor = \lfloor \log_2(2499.9)+1 \rfloor = \lfloor 12.28 \rfloor = 12$
100000	16	$\left\lfloor \log_2\left(\frac{5(100000)-1}{10}\right)+1 \right\rfloor = \lfloor \log_2(49999.9)+1 \rfloor = \lfloor 16.60 \rfloor = 16$

3.2. Algoritmo 7

```c
#include <stdio.h>

int main(){
    int n;
    scanf("%d", &n);
    int count = 0;
    for(int j = n; j > 1; j /= 2){
        if(j < n / 2){
            for(int i = 0; i < n; i += 2){
                //printf("\"Algoritmos\"\n");
                count++;
            }
        }
    }
    printf("Veces: %d\n", count);
    return 0;
}
```

Vemos que podemos simplificar la parte principal del código a:

```c
int count = 0;
for(int j = n / 2; j > 1; j /= 2){
    for(int i = 0; i < n; i += 2){
        //printf("\"Algoritmos\"\n");
        count++;
    }
}
```

El `for` interno por sí solo se ejecutará $\left\lceil \frac{n}{2} \right\rceil$ veces.

Podríamos pensar que el externo es lo mismo que un `for` geométrico pero de reversa, sin embargo no lo es del todo, porque al dividir solo nos quedamos con la parte entera, aunque es muy buena

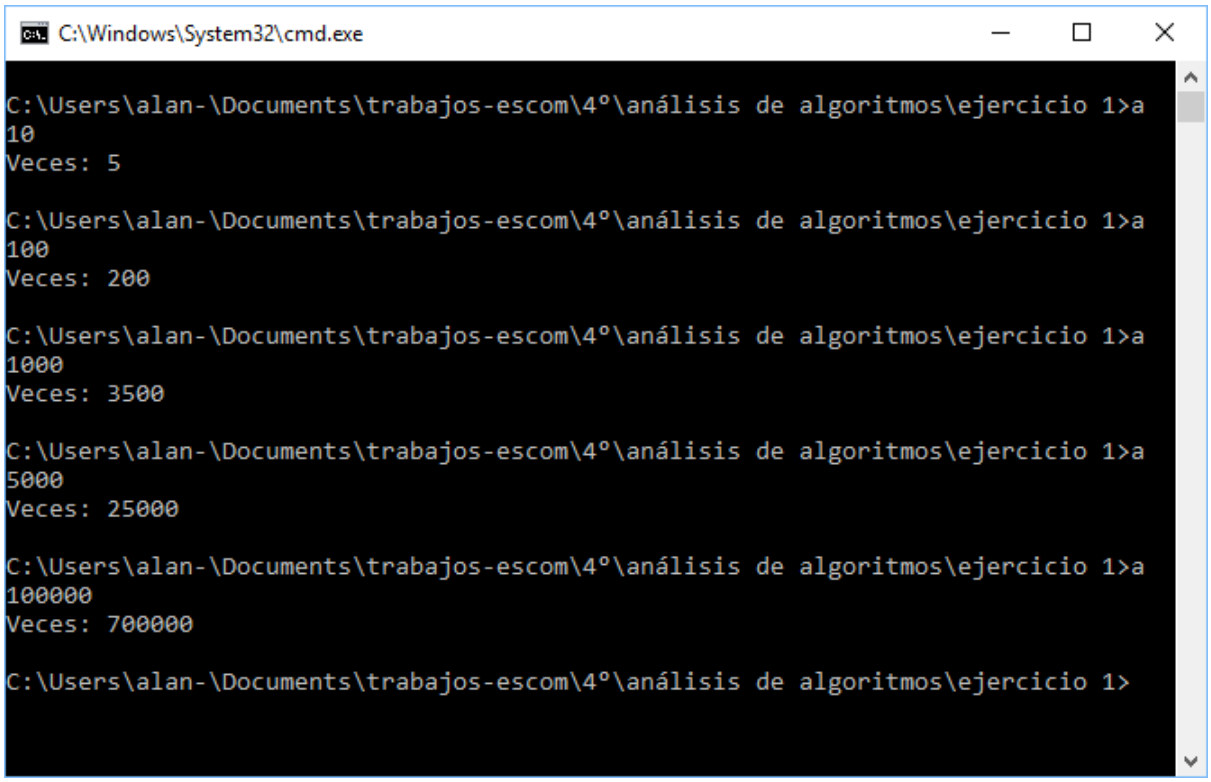
aproximación. De esa forma tenemos que sus parámetros son $a = 2$, $b = \left\lfloor \frac{n}{2} \right\rfloor$ y $r = 2$. Entonces:

$$f(n) \approx \left\lceil \frac{n}{2} \right\rceil \left\lfloor \log_2 \left(\frac{\left\lfloor \frac{n}{2} \right\rfloor}{2} \right) + 1 \right\rfloor$$

Pero parece que nos pasamos por lo de los redondeos, entonces una mejor aproximación sería de forma empírica:

$$f(n) \approx \left\lceil \frac{n}{2} \right\rceil \left\lfloor \log_2 \left(\frac{\left\lfloor \frac{n}{2} \right\rfloor}{2} \right) \right\rfloor \quad \square$$

Comprobación:



n	Número de impresiones reales	Número de impresiones según $f(n)$
10	5	$\left\lceil \frac{10}{2} \right\rceil \left\lfloor \log_2 \left(\frac{\left\lfloor \frac{10}{2} \right\rfloor}{2} \right) \right\rfloor = 5\lfloor \log_2(2.5) \rfloor = 5\lfloor 1.32 \rfloor = 5$
100	200	$\left\lceil \frac{100}{2} \right\rceil \left\lfloor \log_2 \left(\frac{\left\lfloor \frac{100}{2} \right\rfloor}{2} \right) \right\rfloor = 50\lfloor \log_2(25) \rfloor = 50\lfloor 4.64 \rfloor = 200$
1000	3500	$\left\lceil \frac{1000}{2} \right\rceil \left\lfloor \log_2 \left(\frac{\left\lfloor \frac{1000}{2} \right\rfloor}{2} \right) \right\rfloor = 500\lfloor \log_2(250) \rfloor = 500\lfloor 7.96 \rfloor = 3500$
5000	25000	$\left\lceil \frac{5000}{2} \right\rceil \left\lfloor \log_2 \left(\frac{\left\lfloor \frac{5000}{2} \right\rfloor}{2} \right) \right\rfloor = 2500\lfloor \log_2(1250) \rfloor = 2500\lfloor 10.28 \rfloor = 25000$
100000	700000	$\left\lceil \frac{100000}{2} \right\rceil \left\lfloor \log_2 \left(\frac{\left\lfloor \frac{100000}{2} \right\rfloor}{2} \right) \right\rfloor = 50000\lfloor \log_2(25000) \rfloor = 700000$

3.3. Algoritmo 8

```c
#include <stdio.h>

int main(){
    int n;
    scanf("%d", &n);
    int count = 0;
    int i = n;
    while(i >= 0){
        for(int j = n; i < j; i -= 2, j /= 2){
            //printf("\"Algoritmos\"\n");
            count++;
        }
    }
    printf("Veces: %d\n", count);
    return 0;
}
```

Es fácil ver que el algoritmo se cicla para toda $n \geq 0$:

Hacemos la asignación $i \leftarrow n$, por lo tanto entraremos al `while`, pues $i = n \geq 0$. Luego asignamos $j \leftarrow n$, y la condición del `for` es que $i < j$, pero $i = n$ y $j = n$, entonces nunca entramos al `for` pues $i \not< j$. Así, regresamos al `while` que se seguirá cumpliendo, por lo tanto entramos en un ciclo infinito. □

De esta forma, nunca alcanzamos la instrucción de `printf()`, por lo que el número de impresiones es $f(n) = 0$, pero por obvias razones no se puede comprobar experimentalmente.

4. Análisis de casos

4.1. Algoritmo 9

```
1  func Producto2Mayores(A,n)
2      if(A[1] > A[2])                    //3 instrucciones si escogemos este if
3          mayor1 = A[1];
4          mayor2 = A[2];
5      else                              //3 instrucciones si nos vamos por el else
6          mayor1 = A[2];
7          mayor2 = A[1];
8      i = 3;
9      while(i <= n)                      //n-2 repeticiones del ciclo while
10         if(A[i] > mayor1)              //3 operaciones si nos vamos por aquí
11             mayor2 = mayor1;
12             mayor1 = A[i];
13         else if (A[i] > mayor2)         //3 operaciones si nos vamos por aquí
14             mayor2 = A[i];
15         i = i + 1;                     //2 operaciones si no entramos a ningún if
16     return = mayor1 * mayor2;
17 fin
```

Usaremos como operaciones básicas: comparaciones entre elementos del arreglo A y asignaciones a `mayor1` y `mayor2`.

4.1.1. Mejor caso

Se da cuando el elemento mayor de A es $A[1]$ y el siguiente mayor es $A[2]$. De esta forma tenemos 3 instrucciones iniciales (línea 1 a 8), más $2(n-2)$ operaciones dentro del `while` (nunca se cumple que los siguientes elementos sean mayores). Por lo tanto, la complejidad temporal del mejor caso es $f_n(t) = 3 + 2(n-2) = 2n - 1 \quad \square$.

4.1.2. Peor caso

Se da cuando el arreglo A está ordenado de menor a mayor. Aquí tenemos 3 operaciones iniciales (línea 1 a 8) y $3(n-2)$ operaciones dentro del `while`, pues siempre se cumplirá que $A[i] > mayor1$. Por lo tanto, la complejidad temporal del peor caso es $f_n(t) = 3 + 3(n-2) = 3n - 3 \quad \square$.

4.1.3. Caso medio

De las líneas 1 a 8 siempre habrá 3 ejecuciones independientemente de cómo estén distribuidos los elementos de A. Dentro del `while` hay tres caminos posibles con costos igual a:

- Nos vamos por el primer `if`: $3 + 3(n-2) = 3n - 3$
- Nos vamos por el segundo `if`: $3 + 3(n-2) = 3n - 3$
- No entramos al `if`: $3 + 2(n-2) = 2n - 1$

Asumiendo que los tres caminos son igual de probables, la complejidad temporal del caso medio es $f_t(n) = \frac{1}{3}(3n-3) + \frac{1}{3}(3n-3) + \frac{1}{3}(2n-1) = \frac{8n-7}{3}$ $\square$.

4.2. Algoritmo 10

```
1 func OrdenamientoIntercambio(a, n)
2     for(i = 0; i < n - 1; i++)                  //Este for se ejecuta n-1 veces
3         for(int j = i + 1; j < n; j++)          //Este for se ejecuta n-i-1 veces
4             if(a[j] < a[i]){                   //4 instrucciones si a[j] < a[i]
5                 temp = a[i];
6                 a[i] = a[j];
7                 a[j] = temp;
8             }                                   //1 instruccion si a[j] >= a[i]
9 fin
```

Usaremos como operaciones básicas las comparaciones entre elementos del arreglo A, asignaciones a `temp` y al arreglo A.

4.2.1. Mejor caso

Se da cuando el arreglo está ordenado ascendentemente, por lo que las líneas 5 a 7 nunca se ejecutan. Así, la complejidad temporal del mejor caso es:

$$f_t(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i)$$
$$= (n-1)(n-1) - \frac{(n-2)(n-1)}{2} = \frac{n^2-n}{2} \quad \square$$

4.2.2. Peor caso

Se da cuando el arreglo está ordenado descendentemente, por lo que las líneas 5 a 7 siempre se ejecutan. Así, la complejidad temporal del peor caso es 4 veces la del mejor caso: $f_t(n) = 2n^2 - 2n$ $\square$.

4.2.3. Caso medio

Dentro del `if` podemos decidir si ejecutar las líneas 5 a 7 o no. Suponiendo que ambas opciones son igual de probables, en este caso la complejidad temporal del caso medio sí sería el promedio de la del mejor y peor caso: $f_t(n) = \frac{5n^2-5n}{2}$ $\square$.

4.3. Algoritmo 11

```
1 func MaximoComunDivisor(m, n){
2     a = max(n, m);
3     b = min(n, m);
4     residuo = 1;
```

```
5       mientras (b > 0){
6           residuo = a mod b;
7           a = b;
8           b = residuo;
9       }
10      MaximoComunDivisor = a;
11      return MaximoComunDivisor;
12  }
```

Usaremos como operación básica el módulo de `a` y `b`. Sin pérdida de generalidad asumamos que $m \geq n$, pues si $m < n$, se intercambian en las líneas 2 y 3.

4.3.1. Mejor caso

Se da cuando $n = 0$, así nunca entramos al `while`, por lo que la complejidad del mejor caso es $f_t(n) = 0$ □.

4.3.2. Peor caso

Se da cuando m y n son dos números consecutivos de la serie de Fibonacci, es decir, $m = F_{k+1}$ y $n = F_k$ para alguna $k \in \mathbb{N}$.

Por definición sabemos que $F_{k+1} = F_k + F_{k-1}$ y $0 \leq F_{k-1} < F_k$, de esa forma el residuo de dividir F_{k+1} entre F_k será F_{k-1} y el cociente será 1 en **cada** iteración. Entonces, estamos transformando $(F_{k+1}, F_k) \to (F_k, F_{k-1})$, los cuales siguen siendo números de Fibonacci consecutivos.

Como el cociente es al menos 1 para cualquier entrada y con esta entrada siempre obtenemos 1, estamos reduciendo al mínimo los dos números en cada iteración, obteniendo el peor caso. Así, nos tardaremos k divisiones llegar a que $F_k = 0$, y como $F_k \approx \phi^k$, la complejidad temporal aproximada del peor caso será $f_t(n) \approx \log_\phi(n)$, donde $\phi = \frac{1+\sqrt{5}}{2}$. □

4.3.3. Caso medio

Si m y n no son números consecutivos de Fibonacci, al menos uno de los cocientes obtenidos será mayor a uno. Informalmente, podemos aumentar la base del logaritmo, de ϕ a 2, y decir que la complejidad temporal del caso medio es $f_t(n) \approx \log_2(n)$. □

4.4. Algoritmo 12

```
1   func BurbujaOptimizada(A, n)
2       cambios = "Si"
3       i = 0
4       Mientras i < n - 1 && cambios != "No" hacer //Este while se ejecutará a lo más
        ↪  n-1 veces
5           cambios = "No"
6           Para j = 0 hasta (n - 2) - i hacer //Este for se ejecutará n-i-1 veces
7               Si A[j + 1] < A[j] hacer    //4 instrucciones si A[j+1] < A[j]
8                   aux = A[j]
```

```
9                   A[j] = A[j + 1]
10                  A[j + 1] = aux
11                  cambios = "Si"
12              Fin Si                  //1 instrucción si A[j+1] >= A[j]
13          Fin Para
14          i = i + 1
15      Fin Mientras
16  Fin func
```

Usaremos como operaciones básicas las comparaciones entre elementos del arreglo A, asignaciones a `aux` y al arreglo A.

4.4.1. Mejor caso

Se da cuando A está ordenado ascendentemente, pues nunca entraremos al `if` de la línea 7 a la 12, y al `while` de la 4 a la 15 solo una vez, pues la variable `cambios` se queda en "No". Por lo tanto, la complejidad temporal del mejor caso es $f_t(n) = n - 1 \quad \square$.

4.4.2. Peor caso

Se da cuando A está ordenado descendentemente. En cada `while` habrá un cambio, por lo tanto, la complejidad temporal del peor caso es:

$$\begin{aligned} f_t(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 4 = 4 \sum_{i=0}^{n-2} (n-1-i) \\ &= 4(n-1)(n-1) - \frac{4(n-2)(n-1)}{2} = 2n^2 - 2n \quad \square \end{aligned}$$

4.4.3. Caso medio

De forma similar como en el algoritmo 10, solo tenemos dos opciones en el `if` de la línea 7, entonces, la complejidad temporal del caso medio es $f_t(n) = \dfrac{5n^2 - 5n}{2} \quad \square$.

4.5. Algoritmo 13

```
1   func BurbujaSimple(A, n)
2       Para i = 0 hasta n - 2 hacer            //este for se ejecutará n-1 veces
3           Para j = 0 hasta (n - 2) - i hacer    //este for se ejecutará n-i-1 veces
4               Si A[j + 1] < A[j] hacer        //4 instrucciones si A[j+1] < A[j]
5                   aux = A[j]
6                   A[j] = A[j + 1]
7                   A[j + 1] = aux
8               Fin Si                            //1 instrucción si A[j+1] >= A[j]
9           Fin Para
10      Fin Para
11  Fin func
```

Usaremos como operaciones básicas las comparaciones entre elementos del arreglo A, asignaciones a `aux` y al arreglo A.

4.5.1. Mejor caso

Se da cuando A está ordenado ascendentemente. Si ocurre eso, el `if` de la línea 4 no se cumplirá, dando como complejidad temporal del mejor caso la siguiente función:

$$f_t(n) = \sum_{i=0}^{n-2} \sum_{i=0}^{n-2-i} 1 = \frac{n^2 - n}{2} \quad \square$$

4.5.2. Peor caso

Ocurre cuando A está ordenado descendentemente. Esto implica que el `if` de la línea 4 siempre se cumplirá, causando que se ejecuten cuatro instrucciones en vez de una como en el caso pasado. De esta forma, solo multiplicamos por cuatro la función anterior para obtener la complejidad del peor caso: $f_t(n) = 2n^2 - 2$. $\square$

4.5.3. Caso medio

De nuevo, será el promedio de los dos anteriores: $f_t(n) = \frac{5}{2}n^2 - \frac{5}{2}n$. $\square$

4.6. Algoritmo 14

```
1  func Ordena(a, b, c)
2      if(a > b)
3          if(a > c)
4              if(b > c)
5                  salida(a, b, c);        //3 pasos: a > b, a > c, b > c
6              else
7                  salida(a, c, b);        //3 pasos: a > b, a > c, b <= c
8          else
9              salida(c, a, b);            //2 pasos: a > b, a <= c
10     else
11         if(b > c)
12             if(a > c)
13                 salida(b, a, c);        //3 pasos: a <= b, b > c, a > c
14             else
15                 salida(b, c, a);        //3 pasos: a <= b, b > c, a <= c
16         else
17             salida(c, b, a);            //2 pasos: a <= b, b <= c
18 Fin func
```

Usaremos como operaciones básicas las comparaciones entre a, b y c.

4.6.1. Mejor caso

Se da cuando:

- $a > b$ y $a \leq c$
- $a \leq b$ y $b \leq c$

Dando como complejidad $f_t(n) = 2$. □

4.6.2. Peor caso

Se da cuando:

- $a > b$, $a > c$ y $b > c$
- $a > b$, $a > c$ y $b \leq c$
- $a \leq b$, $b > c$ y $a > c$
- $a \leq b$, $b > c$ y $a \leq c$

Dando como complejidad $f_t(n) = 3$. □

4.6.3. Caso medio

Como todas las salidas de la función tienen la misma probabilidad, $\frac{1}{6}$, la complejidad del caso medio es: $f_t(n) = \dfrac{3 \times 4 + 2 \times 2}{6} = \dfrac{8}{3}$. □

4.7. Algoritmo 15

```
1  func Seleccion(A, n)
2      Para k = 0 hasta n - 2 hacer            //for: n-1 veces
3          p = k
4          Para i = k + 1 hasta n - 1 hacer    //for: n-k-1 veces
5              Si A[i] < A[p] entonces            //1 operaciones si A[i]<A[p]
6                  p = i
7              Fin Si                           //1 operación si A[i]>=A[p]
8          Fin Para
9          temp = A[p]
10         A[p] = A[k]
11         A[k] = temp                         //3 operaciones más en cualquier caso
12     Fin Para
13 Fin func
```

Usaremos como operaciones básicas las comparaciones entre elementos del arreglo A, asignaciones a `temp` y al arreglo A.

4.7.1. Mejor caso

Ocurre cuando A está ordenado ascendentemente. Esto implica que la condición de la línea 5 no se cumple nunca, teniendo 4 operaciones en el `for` interno. Por lo tanto, la complejidad temporal del mejor caso es:

$$\begin{aligned} f_t(n) &= \sum_{k=0}^{n-2} \left(3 + \sum_{i=k+1}^{n-1} 1 \right) = \sum_{k=0}^{n-2} (3 + n - 1 - k) \\ &= \sum_{k=0}^{n-2} (n + 2 - k) = (n+2)(n-1) - \frac{(n-2)(n-1)}{2} \\ &= \frac{2n^2 + 2n - 4 - n^2 + 3n - 2}{2} = \frac{n^2 + 5n - 6}{2} \quad \square \end{aligned}$$

4.7.2. Peor caso

Ocurre cuando A está ordenado descendentemente. Esto implica que la condición de la línea 5 siempre se cumple, pero como no estamos contando la asignación de la línea 6 como operación básica, tenemos exactamente la misma complejidad que el mejor caso: $f_t(n) = \frac{n^2 + 5n - 6}{2} \quad \square$.

4.7.3. Caso medio

Como la complejidad del peor caso es la misma que la del mejor caso, la del caso medio también tiene que ser la misma: $f_t(n) = \frac{n^2 + 5n - 6}{2} \quad \square$.