AYED II
Lautaro Gastón Peralta

Práctico 1.3

1. Calculá el orden de complejidad de los siguientes algoritmos:

(a)
```
proc f1(in n : nat)
    if n ≤ 1 then skip
    else
        for i := 1 to 8 do f1(n div 2) od
        for i := 1 to n^3 do t := 1 od
```

(b)
```
proc f2(in n : nat)
    for i := 1 to n do
        for j := 1 to i do t := 1 od
    od
    if n > 0 then
        for i := 1 to 4 do f2(n div 2) od
```

2.1 Principales órdenes de complejidad

Orden	Nombre
O(1)	constante
O(log n)	logarítmica
O(n)	lineal
O(n log n)	casi lineal
O(n²)	cuadrática
O(n³)	cúbica
O(a^n)	exponencial

4(a)

Vamos a desglosar el algoritmo y calcular el **orden de complejidad** del procedimiento $\mathrm{f1}(n)$.

Descripción del algoritmo:

- **Base del caso:** Si $n \leq 1$, el algoritmo no hace nada y termina. Esto es la condición de salida.
- **Recursión:** Si $n > 1$, el algoritmo:
  1. Ejecuta un bucle de $i := 1$ hasta 8, donde en cada iteración llama recursivamente a $\mathrm{f1}(n \div 2)$.
  2. Luego, ejecuta otro bucle de $i := 1$ hasta n^3, donde asigna el valor $t := 1$.

Paso 1: Análisis de la recursión

El primer bucle llama a $\mathrm{f1}(n \div 2)$ 8 veces. La llamada recursiva trabaja sobre un problema de tamaño $n \div 2$, lo que implica que la recurrencia del algoritmo se puede escribir como:

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + \text{coste del segundo bucle}$$

Paso 2: Análisis del segundo bucle

El segundo bucle se ejecuta n^3 veces, y en cada iteración se realiza una asignación constante $t := 1$, lo cual tiene un coste de $O(1)$ por iteración.

Por lo tanto, el coste total del segundo bucle es $O(n^3)$.

Paso 3: Establecer la recurrencia

La recurrencia completa es:

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^3)$$

Paso 2: Análisis del segundo bucle

El segundo bucle se ejecuta n^3 veces, y en cada iteración se realiza una asignación constante $t := 1$, lo cual tiene un coste de $O(1)$ por iteración.

Por lo tanto, el coste total del segundo bucle es $O(n^3)$.

Paso 3: Establecer la recurrencia

La recurrencia completa es:

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + O(n^3)$$

Este es un caso típico de una relación de recurrencia que podemos resolver usando el **Teorema Maestro**.

Paso 4: Aplicar el Teorema Maestro

La recurrencia $T(n) = 8T(n/2) + O(n^3)$ tiene la forma:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

Donde:

- $a = 8$,
- $b = 2$,
- $d = 3$.

Ahora, calculemos el valor de $\log_b(a)$:

$$\log_b(a) = \log_2(8) = 3$$

Comparando el valor de d con $\log_b(a)$:

- Si $d = \log_b(a)$, entonces el tiempo de ejecución del algoritmo es $O(n^d \log n)$.

En este caso, $d = 3$ y $\log_b(a) = 3$, por lo que estamos en el **caso 2** del Teorema Maestro.

Conclusión:

La complejidad del algoritmo es $O(n^3 \log n)$.

4(b)

Vamos a desglosar el procedimiento **f2(n)** y calcular su **complejidad temporal**.

Descripción del algoritmo:

1. **Bucle doble**:
   - El primer bucle recorre $i := 1$ hasta n.
   - Dentro de ese bucle, hay otro que recorre $j := 1$ hasta i, con una asignación constante $t := 1$ en cada iteración.
2. **Recursión**:
   - Después de los bucles, si $n > 0$, el algoritmo llama recursivamente a $\mathrm{f2}(n \div 2)$ **4 veces**.

Paso 1: Análisis del bucle doble

El primer bucle externo recorre i desde 1 hasta n, y el segundo bucle recorre j desde 1 hasta i.
Entonces, el número total de iteraciones de los dos bucles es:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$$

Por lo tanto, el costo del bucle doble es $O(n^2)$.

Paso 2: Análisis de la recursión

Después de los bucles, si $n > 0$, el algoritmo llama 4 veces a $\mathrm{f2}(n \div 2)$. Esto introduce la siguiente relación de recurrencia:

$$T(n) = O(n^2) + 4 \cdot T\left(\frac{n}{2}\right)$$

Paso 3: Establecer la recurrencia

La relación de recurrencia es de la forma:

$$T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n^2)$$

Esta es una recurrencia que podemos resolver usando el **Teorema Maestro**.

Paso 4: Aplicar el Teorema Maestro

La recurrencia tiene la forma:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

Donde:

- $a = 4$,
- $b = 2$,
- $d = 2$.

Ahora, calculemos $\log_b(a)$:

$$\log_b(a) = \log_2(4) = 2$$

Comparando el valor de d con $\log_b(a)$:

- Si $d = \log_b(a)$, entonces el tiempo de ejecución del algoritmo es $O(n^d \log n)$.

En este caso, $d = 2$ y $\log_b(a) = 2$, por lo que estamos en el **caso 2** del Teorema Maestro.

Conclusión:

La complejidad del algoritmo es **O(n^2 \log n)**.

correccion -> la complejidad es : n^2 * log n

2. Dado un arreglo a : **array**$[1..n]$ **of nat** se define una *cima* de a como un valor k en el intervalo $1, \ldots, n$ tal que $a[1..k]$ está ordenado crecientemente y $a[k..n]$ está ordenado decrecientemente.

   (a) Escribí un algoritmo que determine si un arreglo dado tiene cima.

   (b) Escribí un algoritmo que encuentre la cima de un arreglo dado (asumiendo que efectivamente tiene una cima) utilizando una búsqueda secuencial, desde el comienzo del arreglo hacia el final.

   (c) Escribí un algoritmo que resuelva el mismo problema del inciso anterior utilizando la idea de *búsqueda binaria*.

   (d) Calculá y compará el orden de complejidad de ambos algoritmos.

(a)

```
fun es_cima (a : array [1..N] of nat) ret cima : bool
        {-una cima necesita al menos 3 elementos-}
        cima := true
        if n < 3 then
                cima := false
        else
                var i : nat
                i := 1
                {-ahora deberemos buscar si existe la creciente-}
                while i < n ^ a[i] < a[i+1] do
                        i:=i+1
                od
                if i = 1 v i = n then
                        cima := false
                fi
                {-ahora deberemos ver desde el punto encontrado, si en adelante es
                        decreciente-}
                while i < n ^ a[i] > a[i+1] do
                        i:=i+1
                od
                {-ahora tenemos dos condiciones, si llegamos al final del arreglo
                        es una cima, sino, no hay cima-}
                if i = n then
                        res:= true
                else
```

```
                    res:= false
                fi
        fi
end fun

prueba=> a = [1,2,3,2,1]
        cima = es_cima(a[1..5])
                    n = 5
                    else ->         i:=1
                                    primer while i(1) < 5 ^ a[1]<a[2]
                                                i:=2
                                            while i(2)<5 ^ a[2]<a[3]
                                                i:=3
                                            while i(3)<5 ^ a[3]<a[4] {termina ciclo}
                                    if i=1 v i=n then       cima:=false {NO PASA}
                                    segundo while i(3)<5 ^ a[3]>a[4]
                                                i:=4
                                              while i(4)<5 ^ a[4]>a[5]
                                                i:=5
                                              while i(5)<5 ^ a[5]>a[5] {termina ciclo}
                                    if i=n then     cima :=true {PASA ESTE CASO}
                                    else no se ejecuta
        devuelve cima := true
prueba 2 => [1,1,1,1]
        cima = es_cima(a[1..4])
                    n = 4
                    else ->         i:=1
                                    primer while i(1) < 4 ^ a[1]<a[2]  {termina ciclo}
                                    if i=1 v i=n then       cima:=false {PASA}
                                    segundo while i(1)<5 ^ a[3]>a[4] {termina ciclo}
                                    if i=n then     cima :=true {NO PASA ESTE CASO}
                                    else then       cima :=false {PASA ESTE CASO}
        devuelve cima := false
```

OBSERVACION: se podria mejorar haciendo que despues del primer while, el uso del if sea decisivo, para que no se tenga que ejecutar obligatoriamente el segundo while.

(b)

```
fun encontrar_cima (a : array [1..N] of nat) ret cima :nat
        {-una cima necesita al menos 3 elementos-}
        var i : nat
        i := 1
        while i < n ^ a[i] < a[i+1] do
                i:=i+1
        od
        cima := i
end fun
```

(c)

(d)

3. El siguiente algoritmo calcula el mínimo elemento de un arreglo a : **array**$[1..n]$ **of nat** mediante la técnica de programación *divide y vencerás*. Analizá la eficiencia de $minimo(1, n)$.

```
fun minimo(a : array[1..n] of nat, i, k : nat)  ret m : nat
    if i = k then m := a[i]
    else
        j := (i + k) div 2
        m := min(minimo(a, i, j), minimo(a,j+1,k))
    fi
end fun
```

Este algoritmo utiliza la técnica de **divide y vencerás** para encontrar el mínimo en un arreglo $a[1..n]$. Vamos a analizar su **eficiencia** en términos de la **complejidad temporal**.

Descripción del algoritmo:

El algoritmo sigue los siguientes pasos:

1. **Caso base:** Si $i = k$, el subarreglo tiene un solo elemento, y el mínimo es simplemente $a[i]$.
2. **Caso recursivo:** Si $i \neq k$, divide el intervalo $[i, k]$ a la mitad en dos subintervalos:
   - Uno va desde i hasta $j = \frac{i+k}{2}$.
   - El otro va desde $j + 1$ hasta k. El algoritmo llama recursivamente a la función para ambos subintervalos, y luego toma el mínimo entre los resultados de las dos llamadas.

Análisis de la eficiencia:

Paso 1: Establecer la recurrencia

La función divide el problema en dos subproblemas de aproximadamente la mitad del tamaño del original. El número de operaciones extra realizadas fuera de las llamadas recursivas es simplemente calcular el índice medio j y encontrar el mínimo entre los dos resultados, lo cual toma un tiempo constante $O(1)$.

Podemos escribir la siguiente recurrencia para describir el tiempo de ejecución $T(n)$ del algoritmo:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

Paso 2: Resolver la recurrencia

Esta es una recurrencia clásica de **divide y vencerás**, que podemos resolver usando el **Teorema Maestro**.

La recurrencia tiene la forma general:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

Donde:

- $a = 2$ (ya que el problema se divide en 2 subproblemas),
- $b = 2$ (ya que cada subproblema tiene tamaño $\frac{n}{2}$),
- $d = 0$ (ya que el término adicional es constante, $O(1)$).

Para aplicar el Teorema Maestro, calculamos $\log_b(a)$:

$$\log_b(a) = \log_2(2) = 1$$

Paso 3: Comparar los casos del Teorema Maestro

- Si $d < \log_b(a)$, el tiempo de ejecución es $O(n^{\log_b(a)})$.
- Si $d = \log_b(a)$, el tiempo de ejecución es $O(n^d \log n)$.
- Si $d > \log_b(a)$, el tiempo de ejecución es $O(n^d)$.

En este caso, $d = 0$ y $\log_b(a) = 1$, lo que significa que estamos en el **caso 1** del Teorema Maestro, donde el tiempo de ejecución es:

$$T(n) = O(n^{\log_b(a)}) = O(n^1) = O(n)$$

Conclusión:

El algoritmo de **mínimo** tiene una complejidad temporal de **O(n)**. Esto tiene sentido porque el algoritmo recorre todo el arreglo al descomponerlo repetidamente en mitades, pero en última instancia examina todos los elementos una vez.

4. Ordená utilizando $\sqsubset$ e $\approx$ los órdenes de las siguientes funciones. No calculés límites, utilizá las propiedades algebraicas.

   (a) $n \log 2^n$ $\quad 2^n \log n$ $\quad n! \log n$ $\quad 2^n$

   (b) $n^4 + 2 \log n$ $\quad \log(n^{n^4})$ $\quad 2^{4 \log n}$ $\quad 4^n$ $\quad n^3 \log n$

   (c) $\log n!$ $\quad n \log n$ $\quad \log(n^n)$

(a) n log 2^n < 2^n log n < 2^n < n! log n
(b) n^3 log n < n^4 + 2 log n ≈ 2^4 log n < log(n^n^4) < 4 ^ n
anotaciones de b :
n^4+2logn ≈ n^4 && 2^4logn ≈ n^4 (prop: a^log b = b^log a)
log(n^n^4) ≈ n^4 log n
(c) log n! ≈ n log n ≈ log (n^n)

5. Sean K y L constantes, y f el siguiente procedimiento:

```
proc f(in n : nat)
    if n ≤ 1 then skip
    else
        for i := 1 to K do f(n div L) od
        for i := 1 to n^4 do operación_de_O(1) od
```

Determiná posibles valores de K y L de manera que el procedimiento tenga orden:

(a) $n^4 \log n$ (b) n^4 (c) n^5

El primer if es irrelevante, por ende pasaremos al else.
El primer for contiene la llamada recursiva en pedazos mas chicos, por lo que podemos decir que es un divide y venceras.
Segundo for son operaciones BigO de valor 1 n^4
Por lo que podemos definir la funcion:

T(n) = K T(n div L) + n^4

- Si

$$t(n) = \begin{cases} c & \text{si la entrada es pequeña o simple} \\ a * t(n/b) + g(n) & \text{en caso contrario} \end{cases}$$

si $t(n)$ es no decreciente, y $g(n)$ es del orden de n^k, entonces

- $$t(n) \text{ es del orden de } \begin{cases} n^{\log_b a} & \text{si } a > b^k \\ n^k \log n & \text{si } a = b^k \\ n^k & \text{si } a < b^k \end{cases}$$

a) **n^4 log n => K = L^4 => K=16 ^ L=2**
b) **n^4 => K < L^4 => K=8 ^ L=2**
c) **n^5 => no es posible?**