Recurrencias y jerarquía de funciones

Práctico 1.3

Ejercicio 1

Calcular 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 n³ do
               t:= 1
           od
       fi
   end proc
```

primero, verifico si es divide y vencerás:
- hay caso base ($n \leq 1$) y hace algo constante (skip)
- en el caso complejo, se llama recursivamente con un tamaño de n div 2
- se llama recursivamente 8 veces
- además de llamar recursivamente al algoritmo, hace algo constante (n^3 veces)

ahora analizo la complejidad:
- a = 8
- b = 2
- k = 3 (n^3)

t(n) = cantidad de operaciones que realiza el algoritmo f1(n)

$$t(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ 8 * (n \text{ div } 2) + g(n) & \text{si } n > 1 \end{cases}$$

$a = b^k$ puesto que $8 = 2^3$

por lo tanto, el algoritmo es del orden de $n^3 \log n$

```
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
        fi
    end proc
```

divido el problema, primero analizo el primer ciclo for

```
for i:= 1 to n do
    for j:= 1 to i do
        t := 1
    od
od
```

no es divide y vencerás, por lo que calculo el orden de la forma tradicional:

ops(for i:= 1 to n do (for j:= 1 to i do (t:= 1) od) od)
$= \Sigma_{1 \text{ to } n}$ (for j:= 1 to i do (t:= 1) od)
$= \Sigma_{i=1 \text{ to } n} (\Sigma_{j=1 \text{ to } i} (t:= 1))$
$= \Sigma_{i=1 \text{ to } n} (\Sigma_{j=1 \text{ to } i} 1)$
$= \Sigma_{i=1 \text{ to } n} (i)$
$= (n*(n+1))/2$
$= n^2/2 + n/2$
$\approx n^2$

verifico si es divide y vencerás:

- hay caso base ($n \leq 0$) y hace algo constante (skip)
- en el caso complejo, se llama recursivamente con un tamaño de n div 2
- se llama recursivamente 4 veces
- además de llamar recursivamente al algoritmo, hace algo constante (n^2 veces)

ahora analizo la complejidad:

- a = 4
- b = 2
- k = 2

t(n) = cantidad de operaciones que realiza el algoritmo f1(n)

```
t(n) = 0                      si n ≤ 0
       4 * (n div 2) + g(n)   si n > 0
```

$a = b^k$ puesto que $4 = 2^2$

por lo tanto, el algoritmo es del orden de $n^2 \log n$

Ejercicio 2

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

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

```
fun tiene_cima (a: array[1..n] of nat) ret res: bool
    var parcial_res: bool
    var cima: nat
    cima:= 0

    if n ≤ 0 then {- caso arreglo vacío -}
        parcial_res:= false
    else if n = 1 then {- caso arreglo de un elemento -}
        parcial_res:= true
    else if n = 2 then
        if a[1] = a[2] then
            parcial_res:= false
        else
            parcial_res:= true
        fi
    else {- caso promedio -}
        if maximo(a, n) = 1 then {- la cima es el primer elemento -}
            parcial_res:= true
            for i:= 2 to n-1 do
                if a[i] < a[i+1] then
                    parical_res:= parcial_res ∧ false
                fi
        else if minimo(a, n) = a[n] then {- cima es el último elemento-}
            parcial_res:= true
            if ¬esta_ordenado(a, n) then
                parcial_res:= false
            fi
        else
            for i:= 1 to n-2 do
                if a[i] < a[i+1] ∧ a[i+2] < a[i+1] then
                    cima:= i+1
                    parcial_res:= true
                fi
            od
            for j:= cima + 1 to n do
                if a[cima] < a[j]
                    parcial_res:= false
                fi
            od
            res:= parcial_res
        fi
    fi
end fun
```

b) Escribir 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.

```
fun cima (a: array[1..n] of nat) ret cima: nat
    for i:= 2 to n do
        if a[i-1] < a[i]
            cima:= a[i]
        else
            skip
        fi
```

```
end fun
```

Ejercicio 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. Analizar la eficiencia de minimo(1, n).

```
fun minimo(a: array[1..n] of nat, i, k: nat) ret m: nat
    var j: 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
```

primero, verifico si es divide y vencerás:

- hay caso base (i = k) y hace algo constante (asignación a m)
- en el caso complejo, se llama recursivamente con un tamaño de i+k div 2
- se llama recursivamente 2 veces
- además de llamar recursivamente al algoritmo, hace algo constante (2 asignaciones)

ahora analizo la complejidad:

- a = 2
- b = 2
- k = 0

t(n) = cantidad de operaciones que realiza el algoritmo minimo(a,i,k)

```
t(n) = 1                        si m = 1
       2 * t(m div 2) + 1       si n > 1
```

$a > b^k$ puesto que $2 > (2^0 = 1)$

por lo tanto, el algoritmo es del orden de $n^{log\_2\ 2} = n$; es decir, es de orden lineal

Ejercicio 4

Ordenar utilizando $\sqsubset$ e $\approx$ los órdenes de las siguientes funciones. No calcular límites, utilizar las propiedades algebraicas.

a) $n \log 2^n$ - $2^n \log n$ - $n! \log n$ - 2^n
Rta: $n \log 2^n \sqsubset 2^n \sqsubset 2^n \log n \sqsubset n! \log n$

b) $n^4 + 2\log n$ - $\log((n^n)^4)$ - $2^{4 \log n}$ - 4^n - $n^3 \log n$
Rta: $\log((n^n)^4) \sqsubset n^3 \log n \sqsubset n^4 + 2\log n \sqsubset 2^{4 \log n} \sqsubset 4^n$

c) $\log n!$ - $n\log n$ - $\log(n^n)$
Rta: $\log n! \sqsubset \log(n^n) \sqsubset n\log n$

Ejercicio 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
            operacion_de_ω(1)
        od
    fi
end proc
```

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

primero me fijo si es divide y vencerás:

- hay caso base (n ≤ 1) y hace algo constante (skip)
- en el caso complejo, se llama recursivamente con un tamaño de n div L
- se llama recursivamente K veces
- además de llamar recursivamente al algoritmo, hace algo constante (n^4 veces)

ahora analizo la complejidad:

- a = K
- b = L
- k = n^4

t(n) = cantidad de operaciones que realiza el algoritmo f(n)

```
t(n) = 0                      si n ≤ 1
       K * (n div L) + g(n)   si n > 1
```

a) $n^4 \log n$

K = 16
L = 2
$2^4 = 16$, entonces t(n) = $n^4 \log n$

b) n^4

K = 2
L = 4
$2 < 4^4$, entonces t(n) = n^4

c) n^5

K = 32
L = 2
$32 > 2^4$, entonces t(n) = $n^{\log 32} = n^5$

Ejercicio 6

Escribir algoritmos cuyas complejidades sean (asumiendo que el lenguaje no tiene multiplicaciones ni logaritmos, o sea que no se puede escribir **for** i:= 1 **to** n^2 + 2 log n **do** … **od**):

a) $n^2 + 2 \log n$

Divido el problema en dos partes:
en la primera, escribo un algoritmo de la complejidad de n^2:

```
var n: nat
n:= 0
for i:= 1 to n do
    for j to n do
        n:= n + 1
    od
od
```

luego, escribo un algoritmo de la complejidad de 2 log n:

```
var k: nat
for i:= 1 to 2 do
    k:= n
    while 0 < k do
        n div 2
    od
od
```

al finalizar, un algoritmo de la complejidad de n^2 + 2 log n seria el siguiente:

```
proc complejidad_a(in a: array[1..n] of T)
    var l: nat
    l:= 0
    for i:= 1 to n do
        for j to n do
            l:= l + 1
        od
    od

    var m: nat
    for i:= 1 to 2 do
        m:= n
        while 0 < m do
            m:= n div 2
        od
    od
end proc
```

b) $n^2 \log n$

igual que antes divido por partes:
primero busco un algoritmo del orden de n^2:

```
var k: nat
k:= 0
for i:= 1 to n² do
    k:= k + 1
od
```

luego busco un algoritmo del orden de log n:

```
var m: nat
m:= n
while 0 < m do
    m:= n div 2
od
```

ahora lo junto en un mismo algoritmo:

```
proc complejidad_b(in a: array[1..n] of T)
    var k: nat
    k:= 0
    for i:= 1 to n² do
        k:= k + 1
    od

    var m: nat
    m:= n
    while 0 < m do
        m:= n div 2
    od
end proc
```