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
end proc
```

$$fl(x) = \begin{cases} 0 \\ 8 * fl(\frac{x}{2}) + n^3 \end{cases}$$

$$\begin{array}{ccc} a & = & 8 \\ b & = & 2 \\ k & = & 3 \\ & \downarrow & \\ a & = & b^k \\ 8 & = & 2^3 \\ & \downarrow & \\ fl(x) & = & n^3 \log n \end{array}$$

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
```

n	k	s	a
1	0	1 + 4*0	1
2	1	3 + 4*1	7
3	1	6 + 4*1	10
4	7	10 + 4*7	38
5	7	15 + 4*7	43
6	10	21 + 4*10	61
7	10	28 + 4*10	68

$$
\begin{aligned}
ops(f2) &= ops(for\ i := 1\ to\ n\ do; if\ n > 0) \\
&= ops(for\ i := 1\ to\ n\ do) + ops(if\ n > 0) \\
&= \sum_{i=1}^{n} ops(for\ j := 1\ to\ i\ do) + ops(for\ i := 1\ to\ 4\ do) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{i} ops(t := 1) + \sum_{i=1}^{4} ops(f2(ndiv2)) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{i} 1 + 4 * ops(f2(ndiv2)) \\
&= \sum_{i=1}^{n} i + 4 * ops(f2(ndiv2)) \\
&= \frac{n(n+1)}{2} + 4 * ops(f2(ndiv2)) \\
&\approx n^2
\end{aligned}
$$

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***

```
fun hasCima(a : array[1..n] of nat) ret r: bool
    var i: nat
    r := true

    i := 1
    while (i < n && a[i] < a[i + 1]) do
        i := i + 1
    od

    while (i < n && r) do
        if (a[i] < a[i + 1]) then
            r := false
        fi

        i := i + 1
    od

end fun
```

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.

```
fun findCima(a : array[1..n] of nat) ret r: nat

    while (r < n && a[i] < a[i + 1]) do
        r := r + 1
    od

end fun
```

c

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

```
fun findCimaBin(a : array[1..n] of nat) ret r: nat
    var kB, k, kN: nat

    kB := 0
```

```
    k := (n + 1) / 2
    kN := n
    r := 0

    while (r = 0) do
        if (k = n || a[k] > a[k + 1] && a[k] > a[k - 1]) then
            r := k
        else if (a[k] < a[k + 1]) then
            kB := k + 1
        else
            kN := k - 1
        fi

        k := (kB + kN) / 2
    od

end fun
```

d

Calculá y compará el orden de complejidad de ambos algoritmos

$$
\begin{aligned}
ops(findCima) &= ops(while(r < n \quad a[i] < a[i+1])do) \\
&= \textstyle\sum_{i=1}^{n} ops(r := r + 1) \\
&= \textstyle\sum_{i=1}^{n} 1 \\
&= n
\end{aligned}
$$

$$
\begin{aligned}
ops(findCimaBin) &= ops(kB := 0; k := (n+1)/2; kN := n; r := 0; while(r = 0)do) \\
&= 4 + ops(while(r = 0)do) \\
&= 4 + ops(while(r = 0)do) \\
&= \textstyle\sum_{i=1}^{n} ops(r := r + 1) \\
&= \textstyle\sum_{i=1}^{n} 1 \\
&= n
\end{aligned}
$$

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 $minimo(a, 1, n)$.

```
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
```

$$minimo(x) = \begin{cases} 1 \\ 2 * minimo\left(\frac{x}{2}\right) + 0 \end{cases}$$

$$
\begin{array}{ccc}
a & = & 2 \\
b & = & 2 \\
k & = & 0 \\
 & \downarrow & \\
a & > & b^k \\
2 & > & 2^0 \\
 & \downarrow & \\
fl(x) & = & n^{\log_b a} \\
 & = & n^{\log_2 2} \\
 & = & n
\end{array}
$$

4

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

a

$$
\begin{array}{cccc}
n \log 2^n & 2^n \log n & n! \log n & 2^n \\
\downarrow & & & \\
n * n & & & \\
\downarrow & & & \\
n^2 & & &
\end{array}
$$

$$
n^2 \quad \sqsubset \quad 2^n \quad \sqsubset \quad 2^n \log n \quad \sqsubset \quad n! \log n
$$

b

$$
\begin{array}{ccccc}
n^4 + 2\log n & \log\left(n^{n^4}\right) & 2^{4\log n} & 4^n & n^3 \log n \\
 & \downarrow & \downarrow & \downarrow & \\
 & n^4 \log n & (2^{\log n})^4 & 2^{2n} & \\
 & & \downarrow & & \\
 & & n^4 & &
\end{array}
$$

$$
n^3 \log n \quad \sqsubset \quad 2^{4\log n} \quad \approx \quad n^4 + 2\log n \quad \sqsubset \quad \log\left(n^{n^4}\right) \quad \sqsubset \quad 4^n
$$

c

$$
\begin{array}{ccc}
\log n! & n \log n & \log n^n \\
\downarrow & \downarrow & \downarrow \\
\log n + \log{(n-1)!} & n^4 \log n & n \log n
\end{array}
$$

$$
\log n! \quad \sqsubset \quad n \log n \quad \approx \quad \log n^n
$$

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
            operacion_de_O(1)
        od
    fi
end proc
```

$$f(x) = \begin{cases} 0 \\ K * f\left(\frac{x}{L}\right) + n^4 \end{cases}$$

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

a

$$n^4 \log n$$

$$\begin{array}{rcl} fl(x) & = & n^4 \log n \\ & \downarrow & \\ a & = & b^k \\ K & = & L^4 \\ & \downarrow & \\ K & = & 16 \\ L & = & 2 \end{array}$$

b

$$n^4$$

$$\begin{array}{rcl} fl(x) & = & n^4 \\ & \downarrow & \\ a & < & b^k \\ K & < & L^4 \\ & \downarrow & \\ K & = & 2 \\ L & = & 2 \end{array}$$

c

$$n^5$$

$$\begin{array}{rcl} fl(x) & = & n^5 \\ & = & n^{\log_L K} \\ & \downarrow & \\ a & > & b^k \\ K & > & L^4 \\ L^5 & = & K \\ & \downarrow & \\ K & = & 32 \\ L & = & 2 \end{array}$$

6

Escribí algoritmos cuyas complejidades sean (asumiendo que el lenguaje no tiene multiplicaciones ni logaritmos, o sea que no podés escribir `for i := 1 to n^2 + 2 log n do . . . od)`:

a

$$n^2 + 2\log n$$

```
proc f(in n : nat)
    var q, i, j: nat
    q := 0

    for i := 1 to n do
        for j := 1 to n do
            q := q + 1
        od
    od

    i := 1

    while (i < n) do
        i := i * 2
        q := q + 1
    od

end proc
```

b

$$n^2 \log n$$

$$\begin{array}{ccc} fl(x) & = & n^3 \log n \\ & \downarrow & \\ 4 & = & 2^2 \\ a & = & b^k \\ & \downarrow & \\ a & = & 4 \\ b & = & 2 \\ k & = & 2 \end{array}$$

```
proc f(in n : nat)
    vat t: nat

    if (n = 0) then t := 1
    else
        for i := 1 to 4 do
            t := f(n/2)
        od
    fi

end proc
```

c

$$n^3$$

```
proc f(in n : nat)

    if n = 0 then skip
    else
        for i := 1 to 3 do
            f(n - 1)
        od
    fi

end proc
```

7

Una secuencia de valores $x_1, \ldots, x_n$ se dice que tiene orden cíclico si existe un i con $1 \leq i \leq n$ tal que $x_i < x_{i+1} < \ldots < x_n < x_1 < \ldots < x_{i-1}$. Por ejemplo, la secuencia $5, 6, 7, 8, 9, 1, 2, 3, 4$ tiene orden cíclico (tomando $i = 6$).

a

Escribí un algoritmo que determine si un arreglo almacena una secuencia de valores que tiene orden cíclico o no.

```
fun isCiclic(a: array[1..n] of T) ret r: bool
    vat c: nat

    r := (a[n] = a[1])
    c := 0

    if (a[n] > a[1]) then
        c := 1
    fi

    for i := 1 to (n - 1) do
        if (a[i] = a[i + 1]) then
            r := false
        else if (a[i] > a[i + 1]) then
            c := c + 1
        fi
    od

    r := r && (c = 0 || c = 1)

end fun
```

b

Escribí un algoritmo que dado un arreglo $a : array[1..n]\ of\ nat$ que almacena una secuencia de valores que tiene orden cíclico, realice una búsqueda secuencial en el mismo para encontrar la posición del menor elemento de la secuencia (es decir, la posición i).

```
fun findMinInCiclic(a: array[1..n] of nat) ret r: nat
    vat i: nat

    r := 0
    i := 1

    if (a[n] > a[1]) then
        r := 1
    fi
```

```
    while (r = 0) do
        if (a[i] > a[i + 1]) then
            r := i + 1
        fi

        i := i + 1
    od

end fun
```

c

Escribí un algoritmo que resuelva el problema del inciso anterior utilizando la idea de búsqueda binaria.

```
fun findMinInCiclicBin(a: array[1..n] of nat) ret r: nat
    vat kB, k, kN: nat

    r := 0

    if (a[n] > a[1]) then
        r := 1
    fi

    kB := 1
    kN := n

    while (r = 0) do
        k := (kB + kN) / 2

        if (a[k] > a[k + 1]) then
            r := k + 1
        else (a[k] < a[k - 1]) then
            r := k
        else if (a[k] < a[kN]) then
            kN := k
        else if (a[kB] < a[k]) then
            kB := k + 1
        fi
    od

end fun
```

d

Calculá y compará el orden de complejidad de ambos algoritmos

$$
\begin{aligned}
ops(findMinInCiclic) &= 2 + ops(while\ (r = 0)\ do) \\
&= 2 + \textstyle\sum_{i=1}^{n} ops(i = i + 1) + 1 \\
&= 3 + \textstyle\sum_{i=1}^{n} ops(1) \\
&= 3 + n
\end{aligned}
$$

$$
\begin{aligned}
ops(findMinInCiclicBin) &= 3 + ops(while\ (r = 0)\ do) \\
&= 3 + ops(t(n/2) + 1) \\
&= 3 + \log n
\end{aligned}
$$

8

Calculá el orden de complejidad del siguiente algoritmo:

```
proc f3(n : nat)
    for j := 1 to 6 do
        if n <= 1 then skip
        else
            for i := 1 to 3 do
                f3(n div 4)
            od
            for i := 1 to n^4 do
                t := 1
            od
        fi
    od
end proc
```

$$
\begin{aligned}
ops(f3) &= ops(for\ j := 1\ to\ 6\ do) \\
&= \textstyle\sum_{j=1}^{6} ops(for\ i := 1\ to\ 3\ do; for\ i := 1\ to\ n^4\ do) \\
&= \textstyle\sum_{j=1}^{6} (\sum_{i=1}^{3} ops(f3(n\ div\ 4)) + \sum_{i=1}^{n^4} ops(t := 1)) \\
&= \textstyle\sum_{j=1}^{6} 3 * ops(f3(n\ div\ 4)) + \sum_{j=1}^{6} \sum_{i=1}^{n^4} 1 \\
&= 6 * 3 * ops(f3(n\ div\ 4)) + \textstyle\sum_{j=1}^{n} n^4 \\
&= 18 * ops(f3(n\ div\ 4)) + 6n^4 \\
&= 18 * (18 * ops(f3(n\ div\ 16)) + \tfrac{3}{128} n^4) + 6n^4 \\
&\approx n^4
\end{aligned}
$$