Ordenación elemental

Práctico 1.1

Ejercicio 1

Escribí algoritmos para resolver cada uno de los siguientes problemas sobre un arreglo a de posiciones 1 a n, utilizando **do.** Elegí en cada caso entre estos dos encabezados el que sea más adecuado:

```
proc nombre (in/out a:array[1..n] of nat)
      …
end proc

proc nombre (out: a:array[1..n] of nat)
      …
end proc
```

a) Inicializar cada componente del arreglo con el valor 0.

```
proc init_0 (out: a:array[1..n] of nat)
    var counter: int
    counter := 1
    while (counter ≤ n) do
        a[counter] := 0
        counter := counter + 1
    od
end proc
```

b) Inicializar el arreglo con los primeros números naturales positivos.

```
proc init_nat (out: a:array[1..n] of nat)
    var counter: int
    counter := 1
    while (counter ≤ n) do
        a[counter] := counter
        counter := counter + 1
    od
end proc
```

c) Inicializar el arreglo con los primeros n números naturales impares.

```
proc init_imp (out: a:array[1..n] of nat)
    var counter: int
    counter := 1
    while (counter ≤ n) do
        a[counter] := 2 * counter - 1
        counter := counter + 1
    od
end proc
```

d) Incrementar las posiciones impares del arreglo y dejar intactas las posiciones pares.

```
proc inc_imp (out: a:array[1..n] of nat)
    var counter: int
    counter := 1
    while (counter ≤ n) do
        a[counter] := a[counter] + 1
        counter := counter + 2
    od
end proc
```

De a) a c) es solo out ya que el algoritmo no necesita leer los valores iniciales del arreglo; en cambio d) es in/out ya que debo leer los valores de las posiciones impares para incrementarlos en 1.

Ejercicio 2

Transforma cada uno de los algoritmos anteriores en uno equivalente que utilice **for** … **to**.

a) Inicializar cada componente del arreglo con el valor 0.

```
proc init_0 (out: a:array[1..n] of nat)
    for i := 1 to n do
        a[i] := 0
    od
end proc
```

b) Inicializar el arreglo con los primeros números naturales positivos.

```
proc init_nat (out: a:array[1..n] of nat)
    for i := 1 to n do
        a[i] := i
    od
end proc
```

c) Inicializar el arreglo con los primeros n números naturales impares.

```
proc init_imp (out: a:array[1..n] of nat)
    for i := 1 to n do
        a[i] := 2 * i - 1
    od
end proc
```

d) Incrementar las posiciones impares del arreglo y dejar intactas las posiciones pares.

```
proc inc_imp (out: a:array[1..n] of nat)
    for i := 1 to n do
        if (i mod 2 ≠ 0) then
```

```
            a[i] := a[i] + 1
        fi
    od
end proc
```

Ejercicio 3

Escribí un algoritmo que reciba un arreglo a de posiciones 1 a n y determine si el arreglo recibido está ordenado o no. Explica en palabras **que** hace el algoritmo. Explica en palabras **como** lo hace.

```
proc esta_ordenado(in a: array[1..n] of nat, out res: bool)
    res := true
    var counter: int
    counter := 1
    while (counter < n ∧ 1 < n) do {- 1 < n para evitar que el programa
se rompa en los casos de arreglos vacíos y de un elemento -}
        if (a[i + 1] < a[i] ∧ res = true) then
            res := false
        if
        counter := counter + 1
    od
end proc
```

El algoritmo recorre un arreglo y verifica si está ordenado de menor a mayor; lo hace comparando de a pares, de izquierda a derecha, en caso de encontrar un par desordenado, devuelve false; en caso contrario devuelve true.

Ejercicio 4

Ordena los siguientes arreglos, utilizando el algoritmo de ordenación por selección visto en clase. Mostrar en cada paso de iteración cual es el elemento seleccionado y como queda el arreglo después de cada intercambio.

a) [7, 1, 10, 3, 4, 9, 5]
{ busco el mínimo }
[7, **1**, 10, 3, 4, 9, 5]
{ swap con la primera posición }
[**1**, 7, 10, 3, 4, 9, 5]
{ busco el mínimo del arreglo restante }
[1, 7, 10, **3**, 4, 9, 5]
{ swap con la segunda posición }
[1, **3**, 10, 7, 4, 9, 5]
{ busco el mínimo del arreglo restante }
[1, 3, 10, 7, **4**, 9, 5]
{ swap con la tercera posición }
[1, 3, **4**, 7, 10, 9, 5]
{ busco el mínimo del arreglo restante }
[1, 3, 4, 7, 10, 9, **5**]
{ swap con la cuarta posición }
[1, 3, 4, **5**, 10, 9, 7]
{ busco el mínimo del arreglo restante }

[1, 3, 4, 5, 10, 9, **7**]
{ swap con la quinta posición }
[1, 3, 4, 5, **7**, 9, 10]
{ busco el mínimo del arreglo restante }
[1, 3, 4, 5, 7, **9**, 10]
{ swap con la sexta posición }
[1, 3, 4, 5, 7, 9, 10]
{ el arreglo está ordenado }

b) [5, 4, 3, 2, 1]
{ busco el mínimo }
[5, 4, 3, 2, **1**]
{ swap con la primera posición }
[**1**, 4, 3, 2, 5]
{ busco el mínimo del arreglo restante }
[1, 4, 3, **2**, 5]
{ swap con la segunda posición }
[1, **2**, 3, 4, 5]
{ busco el mínimo del arreglo restante }
[1, 2, **3**, 4, 5]
{ swap con la tercera posición }
[1, 2, **3**, 4, 5]
{ busco el mínimo del arreglo restante }
[1, 2, 3, **4**, 5]
{ swap con la cuarta posición }
[1, 2, 3, 4, 5]
{ el arreglo está ordenado }

c) [1, 2, 3, 4, 5]
{ busco el mínimo }
[**1**, 2, 3, 4, 5]
{ swap con la primera posición }
[**1**, 2, 3, 4, 5]
{ busco el mínimo del arreglo restante }
[1, **2**, 3, 4, 5]
{ swap con la segunda posición }
[1, **2**, 3, 4, 5]
{ busco el mínimo del arreglo restante }
[1, 2, **3**, 4, 5]
{ swap con la tercera posición }
[1, 2, **3**, 4, 5]
{ busco el mínimo del arreglo restante }
[1, 2, 3, **4**, 5]
{ swap con la cuarta posición }
[1, 2, 3, 4, 5]
{ el arreglo está ordenado }

Ejercicio 5

Calcula de la manera más exacta y simple posible el número de asignaciones a la variable t de los siguientes algoritmos.

a) t := 0
for i := 1 **to** n **do** {- n asignaciones -}

```
        for j := 1 to n^2 do {- n^2 asignaciones -}
            for k := 1 to n^3 do {- n^3 asignaciones -}
                t := t + 1 {- 1 asignación -}
            od
        od
    od
```

```
ops (t := 0) + ops(for i := 1 to n do (for j := 1 to n^2 do (for k := 1 to n^3 do (t := t + 1))))
= 1 + ops(for i := 1 to n do (for j := 1 to n^2 do (for k := 1 to n^3 do (1))))
= 1 + ops(for i := 1 to n do (for j := 1 to n^2 do (Σ(1 to n^3) (1))))
= 1 + ops(for i := 1 to n do (Σ(1 to n^2) (Σ(1 to n^3) 1)))
= 1 + ops(Σ(1 to n) (Σ(1 to n^2) (Σ(1 to n^3) 1)))
= 1 + ops(Σ(1 to n) (Σ(1 to n^2) (Σ(1 to n^3) 1)))
= 1 + ops(Σ(1 to n) (Σ(1 to n^2) (n^3 * 1))
= 1 + ops(Σ(1 to n) (Σ(1 to n^2) n^3)
= 1 + ops(Σ(1 to n) n^2 * n^3)
= 1 + n * n^2 * n^3
= 1 + n^6
```

```
b) t := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to j + 3 do
                t := t + 1
            od
        od
    od
```

```
ops(t := 0) + ops(for i := 1 to n do(for j := 1 to i do(for k := j to j + 3 do(t := t + 1))))
= 1 + ops(for i := 1 to n do(for j := 1 to i do(Σ(j to j+3) (1))))
= 1 + ops(for i := 1 to n do(Σ(1 to i) (Σ(j to j+3) (1))))
= 1 + ops(Σ(1 to n) (Σ(1 to i) (Σ(j to j+3) 1)))
= 1 + ops(Σ(1 to n) (Σ(1 to i) 4))
= 1 + ops(Σ(1 to n) 4*i)
= 1 + 4 * ops(Σ(1 to n) i)
= 1 + 4 * (n*(n+1) / 2)
= 1 + 2n*(n + 1)
```

Ejercicio 6

Descifra que hacen los siguientes algoritmos, explicar como lo hacen y reescribirlos asignando nombres adecuados a todos los identificadores:

```
proc p (in/out a: array[1..n] of T)
    var x: nat
    for i := n downto 2 do
        x := f(a,i)
        swap(a,i,x)
    od
end proc

fun f (a: array[1..n] of T, i: nat) ret x: nat
```

```
    x := 1
    for j := 2 to i do
        if a[j] > a[x] then
            x := j
        fi
    od
end fun
```

El procedimiento p ordena un arreglo de menor a mayor, lo hace buscando el máximo del arreglo y colocándolo en la última posición, luego buscando el segundo máximo y colocándolo en la ante ultima posicion y asi sucesivamente, es una especia de selection sort que recorre el arreglo de derecha a izquierda; por su parte la función f se encarga de encontrar el máximo de un fragmento del arreglo, lo hace comparando de a pares, en este caso recorriendo el arreglo de izquierda a derecha.

```
proc insertion_sort_downto (in/out a: array[1..n] of T)
    var maximo: nat
    for pos := n downto 2 do
        maximo := max(a,pos)
        swap(a,pos,maximo)
    od
end proc

fun max (a: array[1..n] of T, i: nat) ret x: nat
    x := 1
    for j := 2 to i do
        if a[j] > a[x] then
            x := j
        fi
    od
end fun
```

Ejercicio 7

Ordenar los arreglos del ejercicio 4 utilizando el algoritmo de ordenación por inserción. Mostrar en cada paso de iteración las comparaciones e intercambios realizados hasta ubicar el elemento en su posición.

a) [7, 1, 10, 3, 4, 9, 5]
{ comparo 7,1 }
[**7**, **1**, 10, 3, 4, 9, 5]
{ ordeno }
[**1**, **7**, 10, 3, 4, 9, 5]
{ comparo 7,10 }
[1, **7**, **10**, 3, 4, 9, 5]
{ comparo 3,10 }
[1, 7, **10**, **3**, 4, 9, 5]
{ ordeno }
[1, 7, **3**, **10**, 4, 9, 5]
{ comparo 7,3 }
[1, **7**, **3**, 10, 4, 9, 5]
{ ordeno }
[1, **3**, **7**, 10, 4, 9, 5]

{ comparo 1,3 }
[**1**, **3**, 7, 10, 4, 9, 5]
{ comparo 10,4 }
[1, 3, 7, **10**, **4**, 9, 5]
{ ordeno }
[1, 3, 7, **4**, **10**, 9, 5]
{ comparo 7,4 }
[1, 3, **7**, **4**, 10, 9, 5]
{ ordeno }
[1, 3, **4**, **7**, 10, 9, 5]
{ comparo 3,4 }
[1, **3**, **4**, 7, 10, 9, 5]
{ comparo 10,9 }
[1, 3, 4, 7, **10**, **9**, 5]
{ ordeno }
[1, 3, 4, 7, **9**, **10**, 5]
{ comparo 7,9 }
[1, 3, 4, **7**, **9**, 10, 5]
{ comparo 10,5 }
[1, 3, 4, 7, 9, **10**, **5**]
{ ordeno }
[1, 3, 4, 7, 9, **5**, **10**]
{ comparo 9,5 }
[1, 3, 4, 7, **9**, **5**, 10]
{ ordeno }
[1, 3, 4, 7, **5**, **9**, 10]
{ comparo 7,5 }
[1, 3, 4, **7**, **5**, 9, 10]
{ ordeno }
[1, 3, 4, **5**, **7**, 9, 10]
{ comparo 4,5 }
[1, 3, **4**, **5**, 7, 9, 10]
{ el arreglo está ordenado }

b) [5, 4, 3, 2, 1]
{ comparo 5,4 }
[**5**, **4**, 3, 2, 1]
{ ordeno }
[**4**, **5**, 3, 2, 1]
{ comparo 5,3 }
[4, **5**, **3**, 2, 1]
{ ordeno }
[4, **3**, **5**, 2, 1]
{ comparo 4,3 }
[**4**, **3**, 5, 2, 1]
{ ordeno }
[**3**, **4**, 5, 2, 1]
{ comparo 5,2 }
[3, 4, **5**, **2**, 1]
{ ordeno }
[3, 4, **2**, **5**, 1]
{ comparo 4,2 }
[3, **4**, **2**, 5, 1]
{ ordeno }
[3, **2**, **4**, 5, 1]

{ comparo 3,2 }
[**3**, **2**, 4, 5, 1]
{ ordeno }
[**2**, **3**, 4, 5, 1]
{ comparo 5,1 }
[2, 3, 4, **5**, **1**]
{ ordeno }
[2, 3, 4, **1**, **5**]
{ comparo 4,1 }
[2, 3, **4**, **1**, 5]
{ ordeno }
[2, 3, **1**, **4**, 5]
{ comparo 3,1 }
[2, **3**, **1**, 4, 5]
{ ordeno }
[2, **1**, **3**, 4, 5]
{ comparo 2,1 }
[**2**, **1**, 3, 4, 5]
{ ordeno }
[**1**, **2**, 3, 4, 5]
{ el arreglo está ordenado }

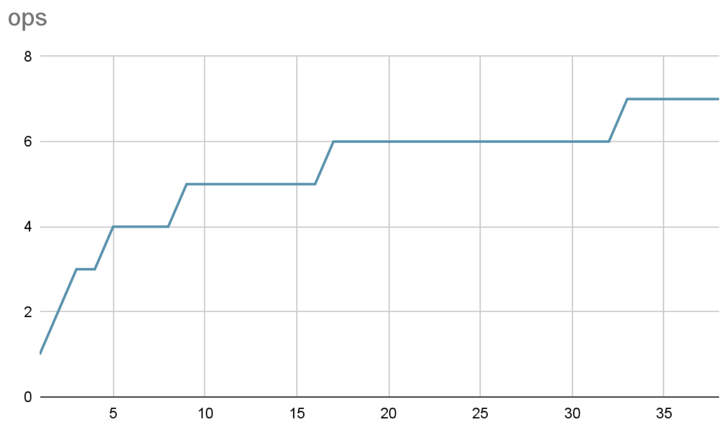
c) [1, 2, 3, 4, 5]
{ comparo 1,2 }
[**1**, **2**, 3, 4, 5]
{ comparo 2,3 }
[1, **2**, **3**, 4, 5]
{ comparo 3,4 }
[1, 2, **3**, **4**, 5]
{ comparo 4,5 }
[1, 2, 3, **4**, **5**]
{ el arreglo está ordenado }

Ejercicio 8

Calcular el orden del número de asignaciones a la variable t de los siguientes algoritmos:

a) t := 1
while t < n **do**
t := t * 2
od

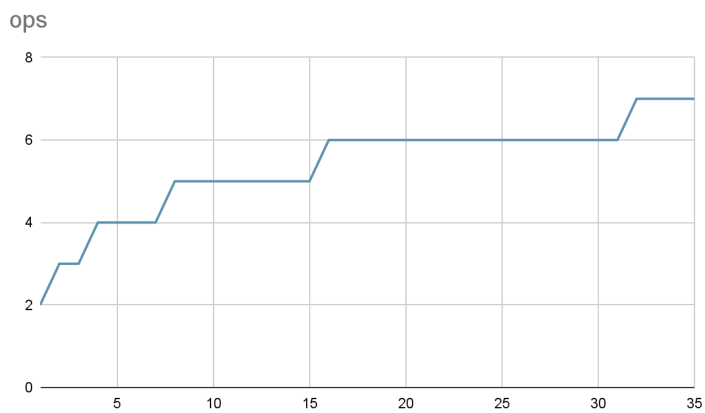
n	1	2	3	4	5	6	7	8	9	10	17	33
ops	1	2	3	3	4	4	4	4	5	5	6	7



luego, ops(n) ~ $\log_2(n)$

```
b) t := n
   do t > 0
       t := t div 2
   od
```

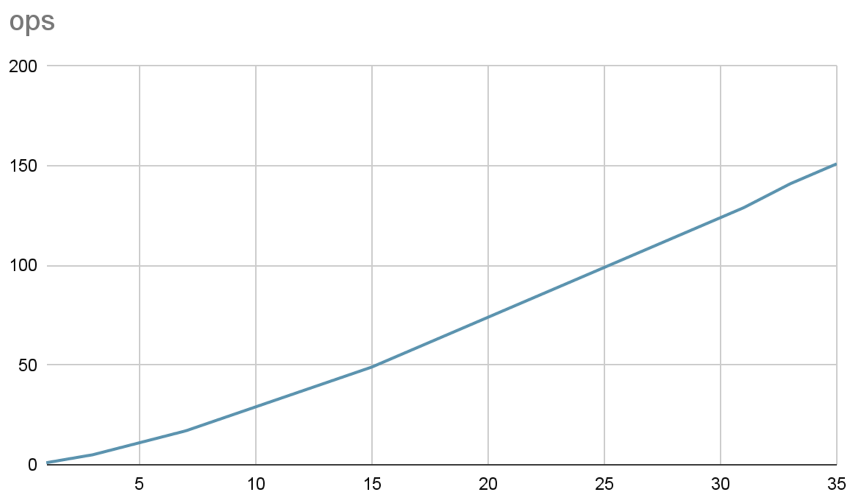
n	1	2	3	4	5	6	7	8	9	10	17	33
ops	2	3	3	4	4	4	4	5	5	5	6	7



luego, ops(n) ~ $\log_2(n)$

```
c) for i := i to n do
       t := i
       do t > 0
           t := t div 2
       od
   od
```

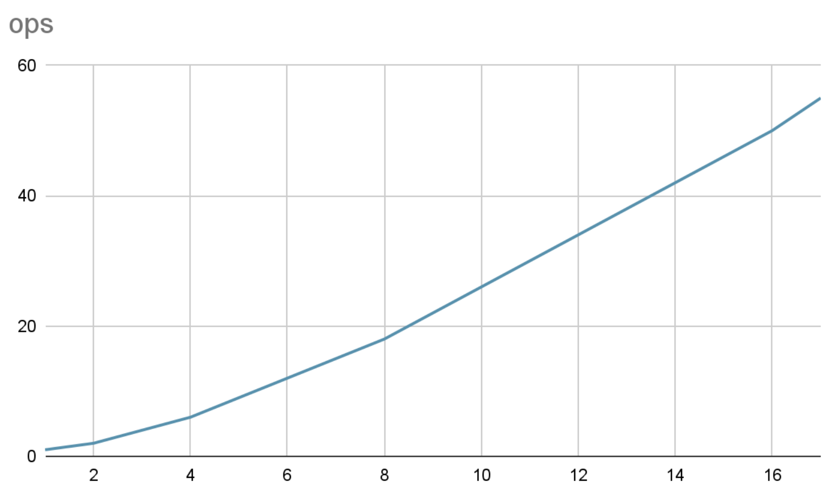
n	1	2	3	4	5	6	7	8	9
ops	1	ops(1)+2	ops(2)+2	ops(3)+3				ops(7)+4	ops(8)+4
	1	3	5	8	11	14	17	21	25



luego, ops(n) ~ n^2

```
d) for i := 1 to n do
       t := i
       do t > 0
           t := t - 2
       od
   od
```

n	1	2	3	4	5	6	7	8	9
ops	1	ops(1)+1	ops(2)+2	ops(3)+2	ops(4)+3				ops(8)+4
ops	1	2	4	6	9	12	15	18	22



luego, ops(n) ~ n^2

Ejercicio 9

Calcula el orden del número de comparaciones del algoritmo del ejercicio 3.

```
proc esta_ordenado(in a: array[1..n] of nat, out res: bool)
    res := true
    var counter: int
```

```
    counter := 1
    while (counter < n ∧ 1 < n) do
        if (a[i + 1] < a[i] ∧ res = true) then
            res := false
        if
        counter := counter + 1
    od
end proc
```

Este es el algoritmo original, lo voy a llevar a otra versión en la que se utilice **for … od** para calcular de manera más simple la cantidad de operaciones.

```
proc esta_ordenado(in a: array[1..n] of nat, out res: bool)
    res := true
    for 1 to n - 1 do
        if a[i + 1] < a[i] then
            res := false
        else
            skip
        fi
    od
end proc
```

ops(esta_ordenado) =
= ops(res := true) + ops(**for** 1 **to** n-1 **do**(**if** a[i + 1] < a[i] **then** res := false **else** skip **fi**))
= 1 + ops(**for** 1 **to** n-1 **do**(ops(res := false) **o** ops(skip)))
= 1 + ops(Σ(1 **to** n-1) 1)
= 1 + (n-1 * 1))
= **n**

El número de comparaciones de esta_ordenado es del orden de n.

Ejercicio 10

Descifra que hacen los siguientes algoritmos, explicar como lo hacen y reescribirlos asignando nombres adecuados a todos los identificadores:

```
proc q (in/out a: array[1..n] of T)
    for i := n-1 downto 1 do
        r(a,i)
    od
end proc

proc r (in/out a: array[1..n] of T, in i: nat)
    var j: nat
    j := i
    while j < n ∧ a[j] > a[j+1] do
        swap(a,j+1,j)
        j := j + 1
    od
end proc
```

El algoritmo ordena un arreglo de menor a mayor recorriendo de derecha a izquierda dicho arreglo, el procedimiento de ordenación es similar al de la ordenación por inserción.

```
proc instertion_sort_downto (in/out a: array[1..n] of T)
    for i := n-1 downto 1 do
        insert_up_to(a,i)
    od
end proc

proc insert_up_to (in/out a: array[1..n] of T, in pos: nat)
    var j: nat
    j := pos
    while j < n ∧ a[j] > a[j+1] do
        swap(a,j+1,j)
        j := j + 1
    od
end proc
```