Ejercicio 1

Escribir algoritmos para resolver cada uno de los siguientes problemas sobre un arreglo a de posiciones 1 a n, utilizando **do**. Elegir 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: nat
    counter:= 1
    while (counter ≤ n) do
        a[counter]:= 0
        counter:= counter + 1
    od
end proc
```

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

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

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

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

De a) a c) se utiliza proc nombre (out a: array[1..n] of nat) ya que no debo leer las posiciones del arreglo, como sí ocurre en d).

Ejercicio 2

Transformar 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)
    var i: nat
    for i:= 1 to n do
        a[i]:= 0
    od
end proc
```

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

```
proc init_par (out a:array[1..n] of nat)
    var i: 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)
    var i: 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_par (in/out a:array[1..n] of nat)
    var i: nat
    for i:= 1 to n do
        if (i mod 2 = 1) then a[i]:= a[i] + 1 fi
    od
end proc
```

Ejercicio 3

Escribir 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 **qué** hace el algoritmo. Explica en palabras **cómo** lo hace.

```
proc its_sorted (in a:array[1..n] of nat, out res: bool)
```

```
    res:= true
    if (n = 1) → skip
       (2 ≤ n) → for i:= 1 to n-1 do
                      if (a[i] > a[i+1]) then res:= false fi
                 od
    fi
end proc
```

El algoritmo determina si el arreglo recibido está ordenado de menor a mayor, para ello primero trabaja con una variable booleana que comienza con un valor de true y cambia a false si o solo si detecta un par desordenado, también evalúa la longitud del arreglo:
- si es de un elemento está ordenado
- si es de 2 o más elementos, evalúa de a pares, si un elemento es mayor al que le sigue determina que el arreglo no está ordenado.

Ejercicio 4

Ordenar 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 menor elemento -}
[7, **1**, 10, 3, 4, 9, 5] {- intercambio con el elemento de la primera posición -}
[**1**, 7, 10, 3, 4, 9, 5] {- busco el menor elemento de los restantes -}
[1, 7, 10, **3**, 4, 9, 5] {- intercambio con el elemento de la segunda posición -}
[1, **3**, 10, 7, 4, 9, 5] {- busco el menor elemento de los restantes -}
[1, **3**, 10, 7, 4, 9, 5] {- intercambio con el elemento de la tercera posición -}
[1, 3, **4**, 7, 10, 9, 5] {- busco el menor elemento de los restantes -}
[1, 3, 4, 7, 10, 9, **5**] {- intercambio con el elemento de la cuarta posición -}
[1, 3, 4, **5**, 10, 9, 7] {- busco el menor elemento de los restantes -}
[1, 3, 4, 5, 10, 9, **7**] {- intercambio con el elemento de la cuarta posición -}
[1, 3, 4, 5, 7, 9, 10] {- busco el menor elemento de los restantes -}
[1, 3, 4, 5, 7, **9**, 10] {- está en la posición correcta, el arreglo está ordenado -}

b) [5, 4, 3, 2, 1] {- busco el menor elemento -}
[5, 4, 3, 2, **1**] {- intercambio con el elemento de la primera posición -}

[**1**, 4, 3, 2, 5] {- busco el menor elemento de los restantes -}
[1, 4, 3, **2**, 5] {- intercambio con el elemento de la segunda posición -}
[1, **2**, 3, 4, 5] {- busco el menor elemento de los restantes -}
[1, 2, **3**, 4, 5] {- está en la posición correcta, busco el menor elemento de los restantes -}
[1, 2, 3, **4**, 5] {- está en la posición correcta, el arreglo está ordenado -}

c) [1, 2, 3, 4, 5] {- busco el menor elemento -}
[**1**, 2, 3, 4, 5] {- está en la posición correcta, busco el menor elemento de los restantes -}
[1, **2**, 3, 4, 5] {- está en la posición correcta, busco el menor elemento de los restantes -}
[1, 2, **3**, 4, 5] {- está en la posición correcta, busco el menor elemento de los restantes -}
[1, 2, 3, **4**, 5] {- está en la posición correcta, el arreglo está ordenado -}

Ejercicio 5

Calcular 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
    for j:= 1 to n² do
        for k:= 1 to n³ do
            t:= t + 1
        od
    od
od
```

Si bien puedo notar a ojo que el algoritmo hace n^6 comparaciones, lo hacemos de forma analítica:

Sea A el algoritmo:

$$
\begin{aligned}
ops(A) &= ops(t := 0) + ops(\textbf{for } i := 1 \textbf{ to } n \textbf{ do } (\textbf{for } j := 1 \textbf{ to } n^2 \textbf{ do } (\textbf{for } k := 1 \textbf{ to } n^3 \textbf{ do } t := t + 1 \textbf{ od})\textbf{od})\textbf{od}) \\
&= 1 + ops(\textbf{for } i := 1 \textbf{ to } n \textbf{ do } (\textbf{for } j := 1 \textbf{ to } n^2 \textbf{ do } (\Sigma(1 \text{ to } n^3)\ (1))\ \textbf{od})\ \textbf{od}) \\
&= 1 + ops(\textbf{for } i := 1 \textbf{ to } n \textbf{ do } (\Sigma(1 \text{ to } n^2)\ (\Sigma(1 \text{ to } n^3)\ (1)))\ \textbf{od}) \\
&= 1 + ops(\Sigma(1 \text{ to } n)\ (\Sigma(1 \text{ to } n^2)\ (\Sigma(1 \text{ to } n^3)\ (1)))) \\
&= 1 + ops(\Sigma(1 \text{ to } n)\ (\Sigma(1 \text{ to } n^2)\ (n^3 * 1))) \\
&= 1 + ops(\Sigma(1 \text{ to } n)\ (n^2\ (n^3 * 1)))
\end{aligned}
$$

```
           = 1 + ops(n (n² (n³ * 1)))
           = 1 + n * n² * n³
           = 1 + n⁶
           {- El 1 es despreciable, lo puedo obviar -}
```

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(A) = 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 od) od) od)
       = 1 + ops(for i := 1 to n do (for j := 1 to i do (Σ(j to j+3) (1)) od) od)
       = 1 + ops(for i := 1 to n do (Σ(1 to i) (Σ(j to j+3) (1))) od)
       = 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
       {- en el enunciado tenemos que Σ(i to n) i = n*(n+1) / 2 -}
       = 1 + ops(Σ(1 to n) * (n*(n+1) / 2) * 4
       = 1 + n * (n*(n+1) / 2) * 4
       = 1 + 4n²(n+1) / 2
       = 1 + 2n²(n+1)
       {- El 1 es despreciable, el resultado es 2n²(n+1) -}
```

Ejercicio 6

Descifrar qué 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
```

Los algoritmos ordenan un arreglo de menor a mayor, éstos actúan como una especie de selection sort, pero ordenando de derecha a izquierda, buscando el elemento máximo del arreglo y haciendo un swap con la última posición, luego se busca el máximo elemento de los restantes y se hace swap con la ante última posición, se sigue de la misma manera hasta llegar a los dos últimos elementos, que se comparan y se hace el swap en caso de ser necesario.

Se podría reescribir con mejores nombres de variables de la siguiente manera:

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


fun find_max_pos (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 y 1 -}
[**7, 1,** 10, 3, 4, 9, 5] {- swap -}
[**1, 7,** 10, 3, 4, 9, 5] {- comparo 10 y 7 -}
[1, **7, 10,** 3, 4, 9, 5] {- comparo 3 y 10 -}
[1, 7, **10, 3,** 4, 9, 5] {- swap -}
[1, 7, **3, 10,** 4, 9, 5] {- comparo 3 y 7 -}
[1, **7, 3,** 10, 4, 9, 5] {- swap -}
[1, **3, 7,** 10, 4, 9, 5] {- comparo 3 y 1 -}
[**1, 3,** 7, 10, 4, 9, 5] {- comparo 4 y 10 -}
[1, 3, 7, **10, 4,** 9, 5] {- swap -}
[1, 3, 7, **4, 10,** 9, 5] {- comparo 4 y 7 -}
[1, 3, **7, 4,** 10, 9, 5] {- swap -}
[1, 3, **4, 7,** 10, 9, 5] {- comparo 4 y 3 -}
[1, **3, 4,** 7, 10, 9, 5] {- comparo 9 y 10 -}
[1, 3, 4, 7, **10, 9,** 5] {- swap -}
[1, 3, 4, 7, **9, 10,** 5] {- comparo 9 y 7 -}
[1, 3, 4, 7, **9, 10,** 5] {- comparo 10 y 5 -}
[1, 3, 4, 7, 9, **10, 5**] {- swap -}
[1, 3, 4, 7, 9, **5, 10**] {- comparo 5 y 9 -}
[1, 3, 4, 7, **9, 5,** 10] {- swap -}
[1, 3, 4, 7, **5, 9,** 10] {- comparo 5 y 7 -}
[1, 3, 4, **7, 5,** 9, 10] {- swap -}
[1, 3, 4, **5, 7,** 9, 10] {- comparo 5 y 4 -}
[1, 3, 4, **5, 7,** 9, 10] {- el arreglo está ordenado -}

b) [5, 4, 3, 2, 1] {- comparo 5 y 4 -}
[**5, 4,** 3, 2, 1] {- swap -}
[**4, 5,** 3, 2, 1] {- comparo 3 y 5 -}
[4, **5, 3,** 2, 1] {- swap -}
[4, **3, 5,** 2, 1] {- comparo 3 y 4 -}
[**4, 3,** 5, 2, 1] {- swap -}
[**3, 4,** 5, 2, 1] {- comparo 2 y 5 -}
[3, 4, **5, 2,** 1] {- swap -}
[3, 4, **2, 5,** 1] {- comparo 4 y 2 -}
[3, **4, 2,** 5, 1] {- swap -}
[3, **2, 4,** 5, 1] {- comparo 3 y 2 -}
[**3, 2,** 4, 5, 1] {- swap -}
[**2, 3,** 4, 5, 1] {- comparo 1 y 5 -}

```
   [2, 3, 4, 5, 1] {- swap -}
   [2, 3, 4, 1, 5] {- comparo 1 y 4 -}
   [2, 3, 4, 1, 5] {- swap -}
   [2, 3, 1, 4, 5] {- comparo 1 y 3 -}
   [2, 3, 1, 4, 5] {- swap -}
   [2, 1, 3, 4, 5] {- comparo 1 y 2 -}
   [2, 1, 3, 4, 5] {- swap -}
   [1, 2, 3, 4, 5] {- el arreglo está ordenado -}
```

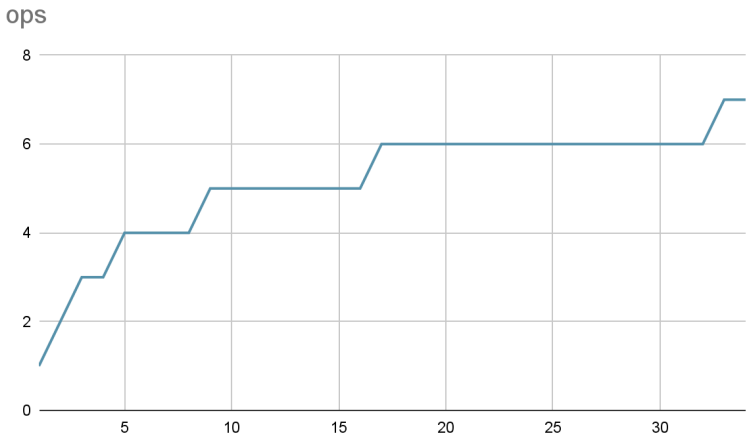
```
c) [1, 2, 3, 4, 5] {- comparo 2 y 1 -}
   [1, 2, 3, 4, 5] {- comparo 3 y 2 -}
   [1, 2, 3, 4, 5] {- comparo 4 y 3 -}
   [1, 2, 3, 4, 5] {- comparo 5 y 4 -}
   [1, 2, 3, 4, 5] {- el arreglo está ordenado -}
```

Ejercicio 8

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

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

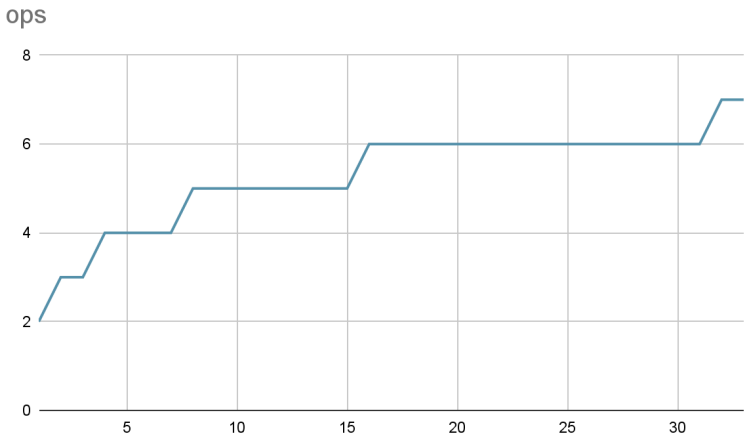
n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	31
ops	1	2	3	3	4	4	4	4	5	5	5	5	5	5	5	5	6	7



luego, ops(n) ~ log2(n)

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

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	32
ops	2	3	3	4	4	4	4	5	5	5	5	5	5	5	5	6	7	8

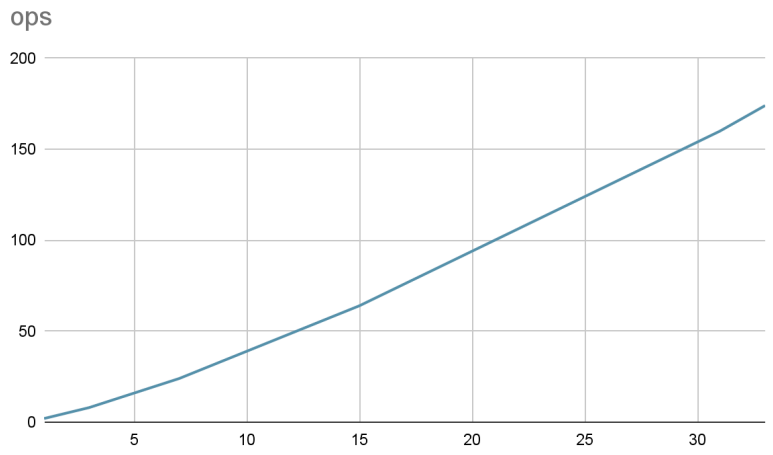


luego, ops(n) ~ log2(n)

```
c) for i := 1 to n do
       t := i
       do t > 0
           t := t div 2
       od
   od
```

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
ops	2	5	8	12	16	20	24	29	34	39	44	49	54	59	64	70	76

por cada n > 1 tengo que ops(n) = ops(n) + ops(n-1)



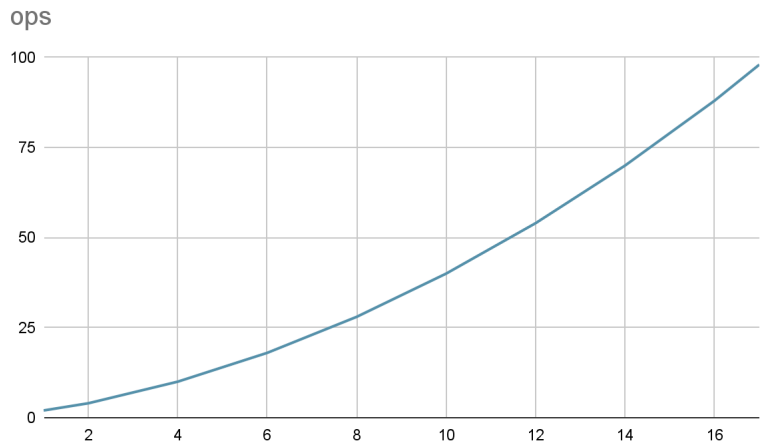
luego, ops(n) ~ n²

```
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	10	11	12	13	14	15	16	17
ops	2	4	7	10	14	18	23	28	34	40	47	54	62	70	79	88	98

por cada n > 1 tengo que ops(n) = ops(n) + ops(n-1)



luego, ops(n) ~ n^2

Ejercicio 9

Calcular el orden del número de comparaciones del algoritmo del ejercicio 3

```
proc its_sorted (in a:array[1..n] of nat, out res: bool)
    res:= true
    if (n = 1) → skip
       (2 ≤ n) → for i:= 1 to n-1 do
                      if (a[i] > a[i+1]) then res:= false fi
```

```
                od
    fi
end proc
```

```
{- No cuento las asignaciones a res ya que solo buscamos el orden del número de comparaciones -}

ops(its_sorted) = ops (for i:= 1 to n-1 do if (a[i] > a[i+1]) then res:= false fi od)
                = ops (for i:= 1 to n-1 do (ops (a[i] > a[i+1])))
                = ops (Σ(1 to n-1) (ops (a[i] > a[i+1])))
                = ops (Σ(1 to n-1) * 1)
                = (n - 1) * 1
                = n - 1
```

El orden del número de comparaciones del algoritmo es de n-1.

Ejercicio 10

Descifrar qué hacen los siguientes algoritmos, explicar cómo 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 enteros de menor a mayor, lo hace tomando de a pares de derecha a izquierda y haciendo swap si corresponde hasta ordenar el arreglo, es una especie de insertion sort.

Se podría reescribir con mejores nombres de variables de la siguiente manera:

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


proc insert_up (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
```