Práctico 1.1

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.
(b) Inicializar el arreglo con los primeros n números naturales positivos.
(c) Inicializar el arreglo con los primeros n números naturales impares.
(d) Incrementar las posiciones impares del arreglo y dejar intactas las posiciones pares.

1)

a)

```
proc inicializarEn0(out a : array[1..n] of nat)
     for j ≔ 0 to n do
          a[j] ≔ 0
     end for
end proc
```

b)

```
proc incializarEnÍndice(out a : array[1..n] of nat)
     for j ≔ 0 to n do
          a[j] ≔ j
     end for
end proc
```

c)

```
proc nombre (out a : array[1..n] of nat)
     for j ≔ 0 to n do
          a[j] ≔ 2 * j + 1
     end for
end proc
```

d)

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

```
      for j ≔ 0 to (n + 1) `div` 2 - 1 do
            var k : int
            k = 2 * j + 1
            a[k] ≔ a[k] + 1
      end for
end proc

proc nombre (in/out a : array[1..n] of nat)
      for j ≔ 0 to n do
            if impar(j) →
                  a[j] = a[j] + 1
            else
                  skip
            end if
      end for
end proc
```

3)

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

```
fun ordered(a[1..n]) ret res : bool
      var j ≔ 1 : int
      res ≔ true
      while j ≤ n - 1 ∧ res do
            res ≔ a[j] ≤ a[j + 1]
            j ≔ j + 1
      od
end fun
```

Que hace:
Determina si el arreglo recibido está ordenado o no.
(lo que dice la consigna)

Como lo hace:
Verifica que cada elemento sea menor o igual a su siguiente

Otro:

```
fun ordered(a[1..n]) ret res : bool
      res ≔ true
      for j ≔ 1 to n - 1 do
            res ≔ res ∧ a[j] ≤ a[j + 1]
      od
end fun
```

4)

4. Ordená los siguientes arreglos, utilizando el algoritmo de ordenación por selección visto en clase. Mostrá en cada paso de iteración cuál es el elemento seleccionado y cómo queda el arreglo después de cada intercambio.

(a) $[7, 1, 10, 3, 4, 9, 5]$ (b) $[5, 4, 3, 2, 1]$ (c) $[1, 2, 3, 4, 5]$

```
[7, 1, 10, 3, 4, 9, 5]
[1, 7, 10, 3, 4, 9, 5]
[1, 3, 10, 7, 4, 9, 5]
[1, 3, 4, 7, 10, 9, 5]
[1, 3, 4, 5, 10, 9, 7]
[1, 3, 4, 5, 7, 9, 10]

[5, 4, 3, 2, 1]
[1, 4, 3, 2, 5]
[1, 2, 3, 4, 5]

[1, 2, 3, 4, 5]
```

5)

5. Calculá de la manera más exacta y simple posible el número de asignaciones a la variable t de los siguientes algoritmos. Las ecuaciones que se encuentran al final del práctico pueden ayudarte.

(a)
```
t := 0
for i := 1 to n do
    for j := 1 to n^2 do
        for k := 1 to n^3 do
            t := t + 1
        od
    od
od
```

(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
```

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
```

$1 + n * n^2 * n^3$

$=$

$1 + n^6$

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

1 + ops(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)
=
1 + ∑i = 1 to n : ops(for j ≔ 1 to i do
                        for k ≔ j to j + 3 do
                            t ≔ t + 1
                        od
                    od)
=
1 + ∑i = 1 to n : (∑j = 1 to i : ops(for k ≔ j to j + 3 do
                                         t ≔ t + 1
                                     od))
=
1 + ∑i = 1 to n : (∑j = 1 to i : 4)
=
1 + 4 * ∑i = 1 to n : (∑j = 1 to i : 1)
=
1 + 4 * ∑i = 1 to n : i
=
1 + 2*n*(n+1)
```

6)

6. Descifrá qué hacen los siguientes algoritmos, explicar cómo 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
```

```
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] →
                  x ≔ j
            else skip fi
      od
end fun

proc ordenarArray (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
```

Si `T` es `Ord`, ordena el arreglo

7)

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

```
[7, 1, 10, 3, 4, 9, 5]
7 ≤ 1
[1, 7, 10, 3, 4, 9, 5]
7 ≤ 10
10 ≤ 3
[1, 7, 3, 10, 4, 9, 5]
7 ≤ 3
[1, 3, 7, 10, 4, 9, 5]
1 ≤ 3
```

```
10 ≤ 4
[1, 3, 7, 4, 10, 9, 5]
7 ≤ 4
[1, 3, 4, 7, 10, 9, 5]
3 ≤ 4
10 ≤ 9
[1, 3, 4, 7, 9, 10, 5]
7 ≤ 9
10 ≤ 5
[1, 3, 4, 7, 9, 5, 10]
9 ≤ 5
[1, 3, 4, 7, 5, 9, 10]
7 ≤ 5
[1, 3, 4, 5, 7, 9, 10]
4 ≤ 5
```

```
[5, 4, 3, 2, 1]
[4, 5, 3, 2, 1]
[4, 3, 5, 2, 1]
[3, 4, 5, 2, 1]
[3, 4, 2, 5, 1]
[3, 2, 4, 5, 1]
[2, 3, 4, 5, 1]
[2, 3, 4, 1, 5]
[2, 3, 1, 4, 5]
[2, 1, 3, 4, 5]
[1, 2, 3, 4, 5]
```

```
[1, 2, 3, 4, 5]
```

8)

8. Calculá el orden del número de asignaciones a la variable t de los siguientes algoritmos.

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

(b)
$t := n$
do $t > 0$
  $t := t$ **div** 2
od

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

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

8a)

```
t ≔ 1
while t < n do
      t ≔ t * 2
od
```

```
      ops(t ≔ 1
            while t < n do
                  t ≔ t * 2
            od)
=
      1 + ops(while t < n do
                  t ≔ t * 2
            od)
=
      1 + ⌈log₂(n)⌉
```

La complejidad es logarítmica

8b)

```
t ≔ n
while t > 0 do
      t ≔ t `div` 2
od
```

La complejidad es logarítmica, demostración:

Primero lo pruebo por inducción para potencias de 2:

Sea `n = 2^m`:

Si `m = 0` (`n = 1`):

```
      ops(t ≔ 1
               while t > 0 do
                   t ≔ t `div` 2
               od)
```

= {El ciclo se ejecuta una vez, realizándose la asignación `t ≔ 0`}

```
      2
```

Caso inductivo para `m+1` suponiendo que vale para `m`:

```
      ops(t ≔ 2^(m+1)
```

```
            while t > 0 do
                t ≔ t `div` 2
            od)
```
= {Una vez entra seguro al ciclo}
```
    ops(
        t ≔ 2^(m+1)
        t ≔ t `div` 2
        while t > 0 do
            t ≔ t `div` 2
        od
      )
```
```
= {2^(m+1) `div` 2 = 2^m}
    1 + ops(
        t ≔ 2^m
        while t > 0 do
            t ≔ t `div` 2
        od
        )
```
≃ {Hipótesis inductiva}
```
    1 + log(n)
```
≃
```
    log(n)
```

Si n no fuera una potencia de 2, es claro que la cantidad de operaciones sería menor o igual que la cantidad de operaciones de cuando es la siguiente potencia de 2, ya que `a < b ⇒ a `div` 2 ≤ b `div` 2`, por lo cual no cambiaría el orden del algoritmo.

8c)

```
for i ≔ 1 to n do
    t ≔ i
    while t > 0 do
        t ≔ t div 2
    od
od
```

```
    ∑i = 1 to n : (2 + ⌊log₂(i)⌋)
=
    2 * n + ∑i = 1 to n : ⌊log₂(i)⌋
```
≃ {Del mismo orden}
```
    2 * n + ∑i = 1 to n : log(i)
=
    2 * n + log(n!)
```
≃ {Del mismo orden}
```
    2 * n + log(n^n)
=
    2 * n + n * log(n)
```
≃ {Del mismo orden}

```
    n * log(n)
```

8d)

```
for i ≔ 1 to n do
    t ≔ i
    while t > 0 do
        t ≔ t − 2
    od
od
```

```
    ops(for i ≔ 1 to n do
                t ≔ i
                while t > 0 do
                    t ≔ t − 2
                od
        od)
=
    ∑ i = 1 to n : ops(t ≔ i
                        while t > 0 do
                            t ≔ t − 2
                        od)
≃ {Del mismo orden}
    ∑ i = 1 to n : (1 + i/2)
=
    n + ∑ i = 1 to n : i/2
=
    n + n*(n+1)/2/2
=
    n²/4 + 5*n/4
≃ {Del mismo orden}
    n²
```

La complejidad es cuadrática

9)

9. Calculá el orden del número de comparaciones del algoritmo del ejercicio 3.

```
res ≔ true
for j ≔ 1 to n - 1 do
    res ≔ res ∧ a[j] ≤ a[j + 1]
od
```

```
     opc(res ≔ true
          for j ≔ 1 to n - 1 do
               res ≔ res ∧ a[j] ≤ a[j + 1]
          od)
=
     ∑j = 1 to n - 1 : opc(res ≔ res ∧ a[j] ≤ a[j + 1])
=
     ∑j = 1 to n - 1 : 1
=
     n - 1
```

10)

10. Descifrá 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
      do j < n ∧ a[j] > a[j+1] → swap(a,j+1,j)
                                  j:= j+1
      od
end proc
```

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

proc reverseInsert (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 algoritmos ordena el arreglo a