AYED II
Lautaro Gastón Peralta

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.

(a)
```
proc init_0 (in/out a:array [1..N] of nat)
    var i : nat
    i := 1
    while i < N do
        a[i]:=0
        i:= i+1
    od
end proc
```

(b)
```
proc init_naturales (in/out a:array [1..N] of nat)
    var i : nat
    var j : nat
    i := 1
    j := 0
    while i < N do
        a[i]:=j
        i:=i+1
        j:=j+1
    od
end proc
```

(c)
```
proc init_naturales_impar (in/out a:array [1..N] of nat)
    var i : nat
    var j : nat
    i := 1
    j := 0
    while i < N do
        if j mod 2 != 0 then
            a[i]:=j
```

```
        fi
        i:=i+1
        j:=j+1
    od
end proc
```

(d)

```
proc mod_pos_imp (in/out a:array [1..N] of nat)
    var i : nat
    i := 1
    while i < N do
        if i mod 2 != 0 then
            a[i]:=a[i] + 1
        fi
        i:=i+1
    od
end proc
```

2. Transformá cada uno de los algoritmos anteriores en uno equivalente que utilice **for** . . . **to** .

(a)

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

(b)

```
proc init_naturales (in/out a:array [1..N] of nat)
    for i:=1 to N do
        for j:=0 to N do
            a[i]:=j
        od
    od
end proc
```

(c)

```
proc init_naturales_impar (in/out a:array [1..N] of nat)
    for i:= 1 to N do
        for j:=0 to N do
            if j mod 2 != 0 then
                a[i]:=j
            fi
        od
    od
end proc
```

(d)

```
proc mod_pos_imp (in/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
```

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 ordenado (a : array [1..N] of nat) ret res : bool
    res:= true
    for i:=1 to N-1 do
        if a[i] >= a[i+1] then
            res:= false
        fi
    od
end fun
```

ejemplo:
a = [1,2,3,5,4,6]   res:= true   N:=6
-primer iteracion:
    i:=1 to N-1{5} if a[1] >= a[2] => if 1 >= 2 => false => res:=true {-No cambia-}
-segunda iteracion:
    i:=2 to N-1{5} if a[2] >= a[3] => if 2 >= 3 => false => res:=true {-No cambia-}
-tercera iteracion:
    i:=3 to N-1{5} if a[3] >= a[4] => if 3 >= 5 => false => res:=true {-No cambia-}
-cuarta iteracion:
    i:=4 to N-1{5} if a[4] >= a[5] => if 5 >= 4 => **true** => **res:=false** {-Cambia-}
-quinta iteracion:
    i:=5 to N-1{5} if a[5] >= a[6] => if 4 >= 6 => false => **res:=false** {-No cambia-}

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

*Selection sort:

```
proc selection_sort (in/out a:array [1..N] of T)
    var i, minp : nat
    i:=1
    do i < n ->  minp:= min_pos_from(a,i)
                 swap (a,i,minp)
                 i:=i+1
    od
end proc
```

*Swap:

```
proc swap (in/out a:array[1..N] of T, in i,j : nat)
```

```
        var tmp : nat
        tmp := a[i]
        a[i] := a[j]
        a[j] := tmp
end proc
```

*Min_pos_from:

```
fun min_pos_from (a : array [1..N] of T, i : nat) ret minp : nat
        var j : nat
        minp:=i
        j:= i+1
        do j<= n -> if a[j] < a[minp] then minp:= j fi
                        j:=j+1
        od
end fun
```

(a)

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

selection sort (a[1..7])

primer iter.-> i := 1 **->** i < 7
minp:= min_pos_from (a,1)
***min_pos_from(a,1) => minp:=2**
swap (a,1,2{minp})
[**1,**7,10,3,4,9,5]
i:=2
segunda iter.-> i := 2 **->** i < 7
minp:= min_pos_from (a,2)
***min_pos_from(a,2) => minp:=4**
swap (a,2,4{minp})
[**1,3**,10,7,4,9,5]
i:=3
tercera iter.-> i := 3 **->** i < 7
minp:= min_pos_from (a,3)
***min_pos_from(a,3) => minp:=5**
swap (a,3,5{minp})
[**1,3**,**4**,7,10,9,5]
i:=4
cuarta iter.-> i := 4 **->** i < 7
minp:= min_pos_from (a,4)
***min_pos_from(a,4) => minp:=7**
swap (a,4,7{minp})
[**1,3**,**4**,**5**,10,9,7]
i:=5
quinta iter.-> i := 5 **->** i < 7
minp:= min_pos_from (a,5)
***min_pos_from(a,5) => minp:=7**
swap (a,5,7{minp})
[**1,3**,**4**,**5**,**7**,9,10]

i:=6
sexta iter.-> i := 6 **->** i < 7
minp:= min_pos_from (a,6)
***min_pos_from(a,6) => minp:=6**
swap (a,6,6{minp})
[**1**,**3**,**4**,**5**,**7**,**9**,10]
i:=7
septima iter.-> i := 7 **->** i < 7 {TERMINA}
[1,3,4,5,7,9,10]

(b)
[5,4,3,2,1]
selection sort (a[1..5])
primer iter.-> i := 1 **->** i < 5
minp:= min_pos_from (a,1)
***min_pos_from(a,1) => minp:=5**
swap (a,1,5{minp})
[**1**,4,3,2,5]
i:=2
segunda iter.-> i := 2 **->** i < 5
minp:= min_pos_from (a,2)
***min_pos_from(a,2) => minp:=4**
swap (a,2,4{minp})
[**1**,**2**,3,4,5]
i:=3
tercera iter.-> i := 3 **->** i < 5
minp:= min_pos_from (a,3)
***min_pos_from(a,3) => minp:=3**
swap (a,3,3{minp})
[**1**,**2**,**3**,4,5]
i:=4
cuarta iter.-> i := 4 **->** i < 5
minp:= min_pos_from (a,4)
***min_pos_from(a,4) => minp:=4**
swap (a,4,4{minp})
[**1**,**2**,**3**,**4**,5]
i:=5
quinta iter.-> i := 5 **->** i < 5 {TERMINA}
[**1**,**2**,**3**,**4**,**5**]

(c)
[1,2,3,4,5]
selection sort (a[1..5])
primer iter.-> i := 1 **->** i < 5
minp:= min_pos_from (a,1)
***min_pos_from(a,1) => minp:=1**
swap (a,1,1{minp})
[**1**,2,3,4,5]
i:=2
segunda iter.-> i := 2 **->** i < 5

minp:= min_pos_from (a,2)

***min_pos_from(a,2) => minp:=2**

swap (a,2,2{minp})

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

i:=3

tercera iter.-> i := 3 **->** i < 5

minp:= min_pos_from (a,3)

***min_pos_from(a,3) => minp:=3**

swap (a,3,3{minp})

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

i:=4

cuarta iter.-> i := 4 **->** i < 5

minp:= min_pos_from (a,4)

***min_pos_from(a,4) => minp:=4**

swap (a,4,4{minp})

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

i:=5

quinta iter.-> i := 5 **->** i < 5 {TERMINA}

[**1**,**2**,**3**,**4**,**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 n² do
        for k := 1 to n³ 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
```

En las ecuaciones que siguen $n, m \in \mathbb{N}$ y k es una constante arbitraria:

$$\sum_{i=1}^{n} 1 = n$$

$$\sum_{i=m}^{n} 1 = n - m + 1 \quad \text{si } n \geq m - 1$$

$$\sum_{i=1}^{n} i = \frac{n * (n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

$$\sum_{i=1}^{n} i^3 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4}$$

$$\sum_{i=m}^{n} (k * a_i) = k * (\sum_{i=m}^{n} a_i)$$

$$\sum_{i=m}^{n} (a_i + b_i) = (\sum_{i=m}^{n} a_i) + (\sum_{i=m}^{n} b_i)$$

$$\sum_{i=m}^{n} (a_i - b_i) = (\sum_{i=m}^{n} a_i) - (\sum_{i=m}^{n} b_i)$$

$$\sum_{i=0}^{n} a_{n-i} = \sum_{i=0}^{n} a_i$$

La última ecuación de la derecha dice simplemente que:

$$a_n + a_{n-1} + \ldots + a_1 + a_0 = a_0 + a_1 + \ldots + a_{n-1} + a_n$$

(a)

ops(t:=0 ; for i:=1 to n do … od)
ops(t:=0) + ops (for i:=1 to n do … od)
1 + Σ **i:=1 to n** ops(for j:=1 to n ^2 do … od)
1 + Σ **i:=1 to n** (Σ **j:=1 to n^2** ops(for k:=1 to n ^3 do (ops (t:=t+1) od)
1 + Σ **i:=1 to n** (Σ **j:=1 to n^2** [Σ **k:=1 to n^3** ops((t:=t+1)])
1 + Σ **i:=1 to n** (Σ **j:=1 to n^2** [Σ **k:=1 to n^3** ops(1)])
1 + Σ **i:=1 to n** (Σ **j:=1 to n^2** [**n^3**])
1 + Σ **i:=1 to n** (**n^2** [**n^3**])
1 + **n** (**n^2** [**n^3**])
1 + n^6

(b)
ops(t:=0 ; for i:=1 to n do … od)
ops(t:=0) + ops (for i:=1 to n do … od)
1 + Σ **i:=1 to n** ops(for j:=1 to i do … od)
1 + Σ **i:=1 to n** (Σ **j:=1 to i** ops(for k:=j to j+3 do (ops (t:=t+1) od)
1 + Σ **i:=1 to n** (Σ **j:=1 to i** [Σ **k:=j to j+3** ops((t:=t+1)])
1 + Σ **i:=1 to n** (Σ **j:=1 to i** [Σ **k:=j to j+3** ops(1)])
1 + Σ **i:=1 to n** (Σ **j:=1 to i** [j+3-j+1])
1 + Σ **i:=1 to n** (Σ **j:=1 to i** [4])
1 + Σ **i:=1 to n** (i[4])
1 + 4 Σ **i:=1 to n** (i)
1 + 4 (n*(n+1))/2
1 + 2 (n*(n+1))

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
```

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

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

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.

INSERTION:

```
proc insert (in/out a : array[1..N] of T,in i : nat)
        var j : nat
        j:=i
        do -> j>1 ^ a[j] < a[j-1] ->      swap(a,j-1,j)
                                          j:=j-1
        od
end proc
```

INSERTION_SORT:

```
proc insertion_sort (in/out a : array[1..N] of T)
        for i:=2 to n do
                insert(a,i)
        od
end proc
```

(a)

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

```
                insertion_sort(a[1..7])
                        i:=2    *insert(a,2)
                                        j:=2 ; a[2] < a[1] ^ 2>1 -> 1 < 7 ->
                                                swap(a,1,2)
                                                j:=1
                                                [1,7,10,3,4,9,5]
                                        j:=1 ; a[1] < a[0] (no existe) ^ 1>1 -> termina insert
                        i:=3    *insert(a,3)
                                        j:=3 ; a[3] < a[2] ^ 3>1 -> 10 < 7 -> NO PASA NADA
                                                [1,7,10,3,4,9,5]
                        i:=4    *insert(a,4)
                                        j:=4 ; a[4] < a[3] ^ 4>1 -> 3 < 10 ->
                                                swap(a,3,4)
                                                j:=3
                                                [1,7,3,10,4,9,5]
                                        j:=3 ; a[3] < a[2] ^ 3>1 -> 3 < 7 ->
                                                swap(a,2,3)
                                                j:=2
                                                [1,3,7,10,4,9,5]
                                        j:=2 ; a[2] < a[1] ^ 2>1 -> 3 < 1 -> NO PASA NADA
                                                [1,3,7,10,4,9,5]
                        i:=5    *insert(a,5)
                                        j:=5 ; a[5] < a[4] ^ 5>1 -> 4 < 10 ->
                                                swap(a,4,5)
                                                j:=4
                                                [1,3,7,4,10,9,5]
```

```
                j:=4 ; a[4] < a[3] ^ 4>1 -> 4 < 7 ->
                    swap(a,3,4)
                    j:=3
                    [1,3,4,7,10,9,5]
                j:=3 ; a[3] < a[2] ^ 3>1 -> 4 < 3 -> NO PASA NADA
                    [1,3,4,7,10,9,5]
        i:=6    *insert(a,6)
                j:=6 ; a[6] < a[5] ^ 6>1 -> 9 < 10 ->
                    swap(a,5,6)
                    j:=5
                    [1,3,4,7,9,10,5]
                j:=5 ; a[5] < a[4] ^ 5>1 -> 9 < 7 -> NO PASA NADA
                    [1,3,4,7,9,10,5]
        i:=7    *insert(a,7)
                j:=7 ; a[7] < a[6] ^ 7>1 -> 5 < 10 ->
                    swap(a,6,7)
                    j:=6
                    [1,3,4,7,9,5,10]
                j:=6 ; a[6] < a[5] ^ 6>1 -> 5 < 9 ->
                    swap(a,5,6)
                    j:=5
                    [1,3,4,7,5,9,10]
                j:=5 ; a[5] < a[4] ^ 5>1 -> 5 < 7 ->
                    swap(a,4,5)
                    j:=4
                    [1,3,4,5,7,9,10]
                j:=4 ; a[4] < a[3] ^ 4>1 -> 5 < 4 -> NO PASA NADA
                    [1,3,4,5,7,9,10]
        FINAL: [1,3,7,4,5,9,10]
```

(b)
[5,4,3,2,1]

```
    insertion_sort(a[1..5])
        i:=2    *insert(a,2)
                j:=2 ; a[2] < a[1] ^ 2>1 -> 4 < 5 ->
                    swap(a,1,2)
                    j:=1
                    [4,5,3,2,1]
                j:=1 ; a[1] < a[0] (no existe) ^ 1>1 -> termina insert
        i:=3    *insert(a,3)
                j:=3 ; a[3] < a[2] ^ 3>1 -> 3 < 5 ->
                    swap(a,2,3)
                    j:=2
                    [4,3,5,2,1]
                j:=2 ; a[2] < a[1] ^ 2>1 -> 3 < 4 ->
                    swap(a,1,2)
                    j:=1
                    [3,4,5,2,1]
```

j:=1 ; a[1] < a[0] (NO EXISTE) ^ 2>1 -> 3 < 4 ->
[3,4,5,2,1]
i:=4 *insert(a,4)
j:=4 ; a[4] < a[3] ^ 4>1 -> 2 < 5 ->
swap(a,3,4)
j:=3
[3,4,2,5,1]
j:=3 ; a[3] < a[2] ^ 3>1 -> 2 < 4 ->
swap(a,2,3)
j:=2
[3,2,4,5,1]
j:=2 ; a[2] < a[1] ^ 2>1 -> 2 < 3 ->
swap(a,1,2)
j:=1
[2,3,4,5,1]
j:=1 ; a[1] < a[0] (NO EXISTE)^ 2>1 -> 2 < 3 ->
[2,3,4,5,1]
i:=5 *insert(a,5)
j:=5 ; a[5] < a[4] ^ 5>1 -> 1 < 5 ->
swap(a,4,5)
j:=4
[2,3,4,1,5]
j:=4 ; a[4] < a[3] ^ 4>1 -> 1 < 4 ->
swap(a,3,4)
j:=3
[2,3,1,4,5]
j:=3 ; a[3] < a[2] ^ 3>1 -> 1 < 3 ->
swap(a,2,3)
j:=2
[2,1,3,4,5]
j:=2 ; a[2] < a[1] ^ 2>1 -> 1 < 2 ->
swap(a,1,2)
j:=1
[1,2,3,4,5]
j:=1 ; a[1] < a[0] ^ 1>1 -> 1 < 2 -> NO PASA NADA
[1,2,3,4,5]
FINAL : [1,2,3,4,5]

(c)
[1,2,3,4,5]
insertion_sort(a[1..5])
i:=2 *insert(a,2)
j:=2 ; a[2] < a[1] ^ 2>1 -> 1 < 2 -> NO PASA NADA
[1,2,3,4,5]
i:=3 *insert(a,3)
j:=3 ; a[3] < a[2] ^ 3>1 -> 2 < 3 -> NO PASA NADA
[1,2,3,4,5]
i:=4 *insert(a,4)
j:=4 ; a[4] < a[3] ^ 4>1 -> 3 < 4 -> NO PASA NADA

[1,2,3,4,5]

i:=5 *insert(a,5)

j:=5 ; a[5] < a[4] ^ 5>1 -> 4 < 5 -> NO PASA NADA

[1,2,3,4,5]

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

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

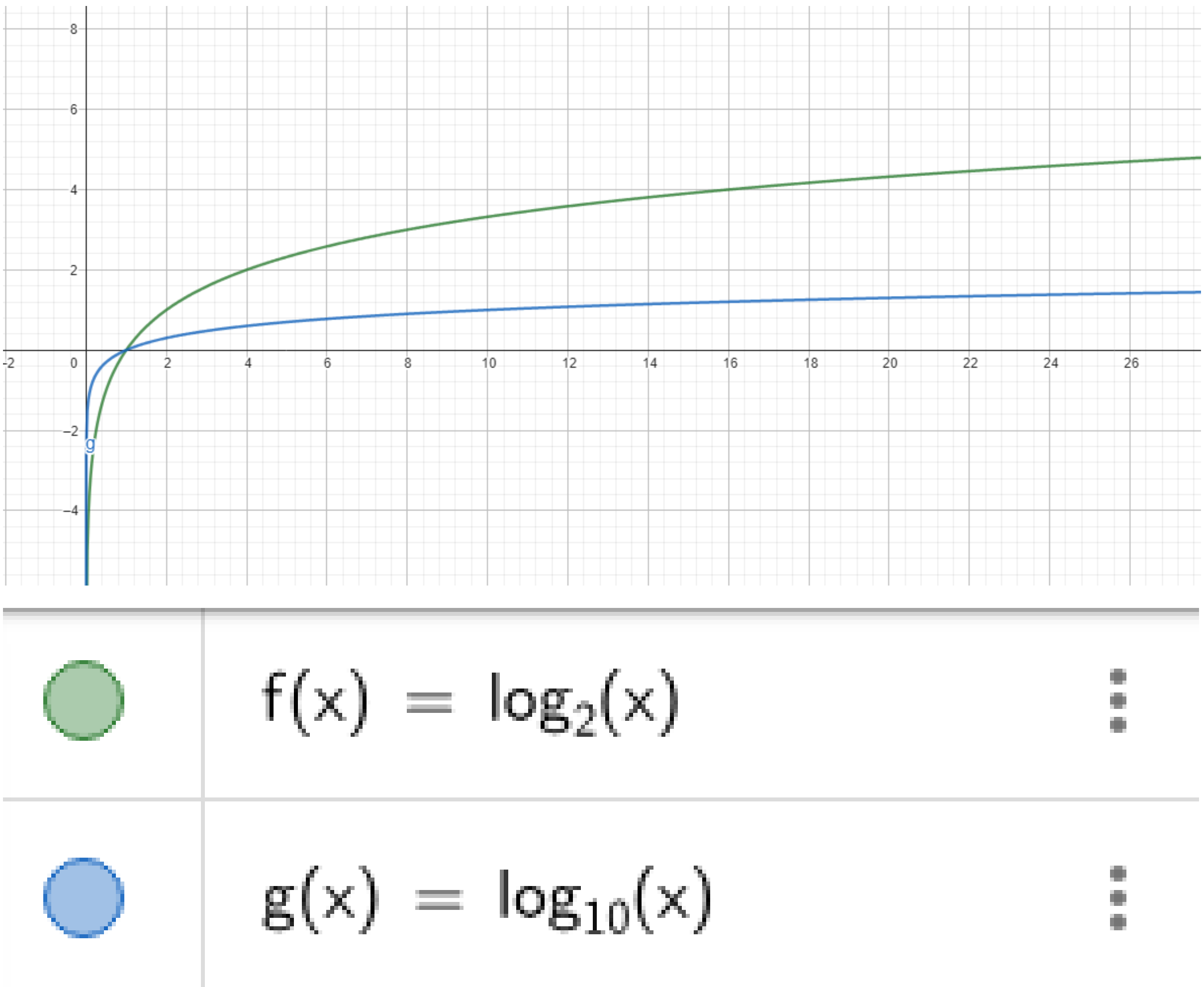
(a)
```
t := 1
do t < n
    t := t * 2
od
```

(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
```

DATOS IMPORTANTES:



Comportamiento Logarítmico (O(log n))

Un algoritmo tiene un comportamiento **logarítmico** cuando reduce el problema de manera **exponencial** en cada iteración, es decir, cuando el tamaño del problema se **divide por una constante** en cada paso. Esto significa que el número de iteraciones disminuye rápidamente a medida que avanzan los pasos.

Características comunes de algoritmos logarítmicos:

1. **División sucesiva del problema:** Si en cada iteración reduces el tamaño del problema a la mitad o a una fracción constante (por ejemplo, $t := t/2$), el número de iteraciones es proporcional a $\log n$.
   - Ejemplo: Búsqueda binaria, donde en cada paso divides el espacio de búsqueda en dos.
   - Ejemplo: Algoritmos como el caso **(a)** y **(b)** en tu imagen, donde t se multiplica o divide por 2 en cada iteración.
2. **Árboles binarios:** Si el problema puede modelarse como un árbol binario balanceado, la altura del árbol es $\log n$. Por ejemplo, al recorrer un árbol binario, el número de pasos suele ser logarítmico con respecto al número de nodos.
3. **Bucles anidados con divisiones:** Si en un bucle interno se reduce el tamaño de una variable por un factor constante (por ejemplo, $t := t/2$), tendrás un comportamiento logarítmico en ese bucle.

Comportamiento Cuadrático (O(n^2))

Un algoritmo tiene un comportamiento **cuadrático** cuando el número de iteraciones del algoritmo crece en función del **cuadrado del tamaño del problema**. Esto generalmente ocurre cuando tienes bucles **anidados** que iteran en función de n.

Características comunes de algoritmos cuadráticos:

1. **Bucles anidados:** Cuando tienes un bucle dentro de otro y ambos recorren desde 1 hasta n, el número total de iteraciones es el producto de ambos, resultando en $O(n^2)$.
   - Ejemplo: Algoritmos que comparan todos los pares de elementos en una lista, como en el algoritmo de ordenamiento por burbuja.
2. **Iteraciones lineales en bucles internos:** Si el bucle externo va de 1 a n y el bucle interno también tiene un número de iteraciones proporcional a n, entonces el comportamiento es cuadrático.
   - Ejemplo: El caso **(d)** en tu imagen, donde el bucle externo va de 1 a n y el bucle interno resta 2 en cada paso. El bucle interno itera $i/2$ veces para cada i, lo que resulta en $O(n^2)$.

Diferencias clave:

- **Logarítmico $O(\log n)$**: Sucede cuando reduces el tamaño del problema en forma exponencial (como dividir por 2). Los cambios en el tamaño del problema disminuyen rápidamente con cada iteración.
- **Cuadrático $O(n^2)$**: Ocurre cuando tienes bucles anidados que dependen linealmente del tamaño del problema. El número de operaciones crece proporcionalmente al cuadrado de n, lo que implica que el número de iteraciones crece mucho más rápidamente que en los casos logarítmicos.

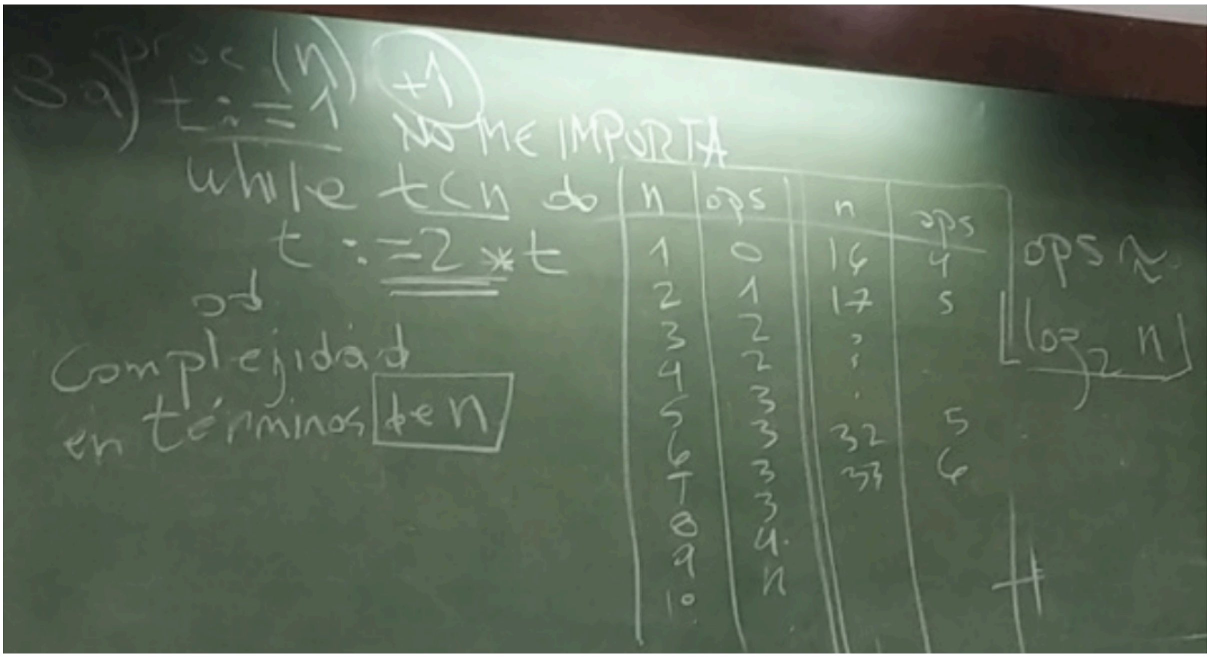
SOLUCIONES

(a)

El algoritmo comienza con `t := 1` y en cada iteración del bucle, se duplica el valor de `t` (`t := t * 2`) hasta que `t` sea mayor o igual a `n`.

El valor de `t` sigue una progresión geométrica de potencias de 2: 1, 2, 4, 8, ..., hasta alcanzar o superar `n`. El número de veces que `t` se duplica antes de superar `n` es aproximadamente el logaritmo en base 2 de `n`. Por lo tanto, el número de iteraciones es aproximadamente `log_2(n)`.

Conclusión: El orden del número de asignaciones a la variable `t` es **O(log n)**.



(b)

Este algoritmo empieza con `t := n` y en cada iteración se divide el valor de `t` por 2 (`t := t div 2`) hasta que `t` sea menor o igual a 0.

Aquí, el valor de `t` sigue una progresión decreciente de la forma `n, n/2, n/4, n/8, ...` hasta llegar a 0. El número de veces que puedes dividir `t` por 2 antes de llegar a 0 es también aproximadamente `log_2(n)`.

Conclusión: El orden del número de asignaciones a la variable `t` es **O(log n)**.

n	ops
0	0
1	1
2	1
3	2
4	3
5	3
⋮	⋮

$\longrightarrow = \log_2 n$

(c)

En este caso, el bucle externo itera desde `i := 1` hasta `i := n`, y dentro de cada iteración, se asigna `t := i`, luego se ejecuta un bucle donde `t` se divide por 2 hasta que `t` sea 0.

Para cada valor de `i`, el número de iteraciones del bucle interno es `log_2(i)`, ya que estamos dividiendo `t` por 2 en cada paso hasta que `t` llegue a 0.

El total de asignaciones a `t` será la suma de `log_2(i)` desde `i = 1` hasta `i = n`. Esto es aproximadamente igual a:

$$\sum_{i=1}^{n} \log_2(i) \approx \log_2(n!) = O(n \log n)$$

Conclusión: El orden del número de asignaciones a la variable `t` es **O(n log n)**.

En el caso (d), el bucle interno disminuye t en 2 en cada iteración. Entonces, para cada i, el número de veces que se ejecuta el bucle interno es aproximadamente $\frac{i}{2}$, ya que estamos restando 2 en cada paso.

Para calcular el número total de asignaciones, necesitamos sumar estas operaciones para cada valor de i desde 1 hasta n:

$$\text{Total de asignaciones} = \sum_{i=1}^{n} \frac{i}{2}$$

Esta suma es aritmética y resulta en:

$$\frac{1}{2} \sum_{i=1}^{n} i = \frac{1}{2} \cdot \frac{n(n+1)}{2} = O(n^2)$$

Por lo tanto, el crecimiento es **cuadrático.** Si el decrecimiento en el valor de t fuera más rápido (por ejemplo, dividiendo t en cada paso como en el caso (c)), entonces podría ser $O(n \log n)$, pero en este caso, como solo restamos 2, el crecimiento sigue siendo cuadrático $O(n^2)$.

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

```
fun ordenado (a : array [1..N] of nat) ret res : bool
        res:= true
        for i:=1 to N-1 do
                if a[i] >= a[i+1] then
                        res:= false
                fi
        od
end fun
```

ops(res:=0 ; for i:=1 to n-1 do … od)
ops(res:=0) + ops (for i:=1 to n-1 do … od)
0 + Σ **i:=1 to n-1** ops(**if** a[i] >= a[i+1] **then** res:=false **fi**)
0 + Σ **i:=1 to n-1** ops(**1**) -> pq se hace una comparación.
0 + n-1
n-1 **-> ES EQUIVALENTE AL O(N)**

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

proc insert_reverse (in/out a : array[1..N] of T, 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
```