AYED2 - Teórico Práctico - Práctico 1

1. Escribı algoritmos para resolver cada uno de los siguientes problemas sobre un arreglo a de posiciones 1 a n, utilizando do. Elegı 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 solo0 (out a:array[1..n] of nat)
    for i := 1 to n do →
        a[i] = 0;
    od
end proc
```

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

c)
```
proc impares(out a:array[1..n] of nat)
    var j : nat
    for i = 1 to n do →
        a[i] = j
        j = j + 2
    od
end proc
```

d)
```
proc incrImpar (in/out a:array[1..n] of nat)
    for i := 1 to n do →
        a[i] := a[i] + 1
        i := i + 1
    od
 end proc
```

2. Transformó a cada uno de los algoritmos anteriores en uno equivalente que utilice for . . . to (ya está)

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

```
fun es_ordenado(a:array[1..n] of nat) r : bool
    var i : nat
    i := 1
    while i < n ^ a[i] ≤  a[i+1] do
       i := i + 1
    od
    r := (i == n)
end fun
```

¿Qué hace?
Nos dice si un arreglo está ordenado
¿Cómo lo hace?
Va recorriendo el arreglo, toma un elemento y el siguiente, si el primero es menor al segundo avanza y se fija en la siguiente posición hasta que termine, si siempre el elemento que tomamos es menor que el de adelante, entonces el arreglo está ordenado, si en algún punto esa condición no se cumple, el algoritmo se termina y diremos que no está ordenado.

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

a)

arreglo a	min_pos_from = minpos	swap(minpos,i)
[**7**,**1**,10,3,4,9,5]	minpos = 2	swap(2,1)
[1,**7**,10,**3**,4,9,5]	minpos = 4	swap(4,2)
[1,3,**10**,7,**4**,9,5]	minpos = 5	swap(5,3)
[1,3,4,**7**,10,9,**5**]	minpos = 7	swap(7,4)
[1,3,4,5,**10**,9,**7**]	minpos = 7	swap(7,5)
[1,3,4,5,7,**9**,**10**]	minpos = 6	swap(6,6)
[1,3,4,5,7,9,**10**]	minpos = 7	swap(7,7)

b)

arreglo a	min_pos_from = minpos	swap(minpos,i)
[**5**,4,3,2,**1**]	minpos = 5	swap(5,1)
[1,**4**,3,**2**,5]	minpos = 4	swap(4,2)
[1,2,**3**,4,5]	minpos = 3	swap(3,3)
[1,2,3,**4**,5]	minpos = 4	swap(4,4)
[1,2,3,4,**5**]	minpos = 5	swap(5,5)

c)

arreglo a	min_pos_from = minpos	swap(minpos,i)
[**1**,2,3,4,5]	minpos = 1	swap(1,1)
[1,**2,**3,4,5]	minpos = 2	swap(2,2)
[1,2**,3**,4,5]	minpos = 3	swap(3,3)
[1,2**,**3,**4**,5]	minpos = 4	swap(4,4)
[1,2**,**3,4,**5**]	minpos = 5	swap(5,5)

5. Calcuĺa de la manera mas exacta y simple posible el numero 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
```

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

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=m}^{n} (k * a_i) = k * (\sum_{i=m}^{n} a_i)$$

$$\sum_{i=m}^{n} 1 = n - m + 1 \quad \text{si } n \geq m - 1 \qquad \sum_{i=m}^{n} (a_i + b_i) = (\sum_{i=m}^{n} a_i) + (\sum_{i=m}^{n} b_i)$$

$$\sum_{i=1}^{n} i = \frac{n * (n+1)}{2} \qquad \sum_{i=m}^{n} (a_i - b_i) = (\sum_{i=m}^{n} a_i) - (\sum_{i=m}^{n} b_i)$$

$$\sum_{i=1}^{n} i^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} \qquad \sum_{i=0}^{n} a_{n-i} = \sum_{i=0}^{n} a_i$$

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

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(C) = ops(t:=0) + ops(for i := 1 to n do C´(i) od)
= 1 + ops(C´(1)) + ops(C´(2)) + ….. + ops(C´(n))
= 1 + σ i = 1..n ops(C´(i))
= 1 + σ i = 1..n ops(for i to n^2 do C´´(i,j) od)
= 1 + σ i = 1..n (σ j = 1..n^2 (C´´(i,j)))
= 1 + σ i = 1..n (σ j = 1..n^2 ops(for k := 1 to n^3 do t := t + 1 od))
= 1 + σ i = 1..n (σ j = 1..n^2 + σ k = 1..n^3 ops(t := t + 1))
= 1 + σ i = 1..n (σ j = 1..n^2 (σ k = 1..n^3 (1)))
= 1 + σ i = 1..n (σ j = 1..n^2 n^3)
= 1 + (σ i = 1..n (n^2)(n^3))
= 1 + n (n^2) (n^3)
= 1 + n^5

b) ops(C) = ops(t:= 0) + ops(for i := 1 to n do C´(i) od)
= 1 + σ i = 1..n (C´(i))
= 1 + σ i = 1..n (ops(for j = 1 to i do C´´(j)))
= 1 + σ i = 1..n (σ j = 1..i (C´´(j)))
= 1 + σ i = 1..n (σ j = 1..i (ops(for k := j to j + 3 do t := t + 1 od)))
= 1 + σ i = 1..n (σ j = 1..i (σ k = j.. j+3 (t := t + 1))))
= 1 + σ i = 1..n (σ j = 1..i (σ k = j.. j+3 (1))))
= 1 + σ i = 1..n (σ j = 1..i (4))
= 1 + σ i = 1..n (4 * i)
= 1 + 4 * (σ i = 1..n (i))
= 1 + 4 * (n*(n+1) / 2)
= 1 + 2 * n * (n+1)

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
```

¿Qué hace p?
p ordena una secuencia de números de forma creciente
¿Cómo lo hace?
El procedimiento p recorre el array desde el final hasta el segundo elemento, en cada iteración, llama a la función “f” para encontrar el índice del máximo elemento dentro de la subsecuencia, es decir, desde el primer elemento hasta el índice actual de iteración. Luego intercambia este elemento máximo con el elemento en la posición actual de interacción

¿Qué hace f?
La función “f” encuentra el índice del máximo elemento en la subsecuencia desde el primer elemento hasta el índice “i”
¿Cómo lo hace?
toma una secuencia y un índice, recorre la secuencia desde la segunda posición hasta el índice dado. Luego devuelve la posición del máximo en ese intervalo. El valor de la misma comienza en uno, puesto que la función comienza comparando esta con el índice actual y si encuentra un número mayor, actualiza este valor hasta recorrer todo el intervalo.

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

a)

arreglo a	insert(a,i)	swap(a,j-1,j)	(i,j)
[7,**1**,10,3,4,9,5]	insert(a,2)	swap(a,1,2)	(2,2)
[1,7,**10**,3,4,9,5]	insert(a,3)		(3,3)
[1,7,10,**3**,4,9,5]	insert(a,4)	swap(a,3,4)	(4,4)
[1,7,3,**10**,4,9,5]		swap(a,2,3)	(4,3)
[1,3,7,**10**,4,9,5]			(4,2)
[1,3,7,10,**4**,9,5]	insert(a,5)	swap(a,4,5)	(5,5)
[1,3,7,4,**10**,9,5]		swap(a,3,4)	(5,4)
[1,3,4,7,**10**,9,5]			(5,3)
[1,3,4,7,10,**9**,5]	insert(a,6)	swap(a,5,6)	(6,6)
[1,3,4,7,9,**10**,5]			(6,5)
[1,3,4,7,9,10,**5**]	insert(a,7)	swap(a,6,7)	(7,7)
[1,3,4,7,9,5,**10**]		swap(a,5,6)	(7,6)
[1,3,4,7,5,9,**10**]		swap(a,4,5)	(7,5)
[1,3,4,5,7,9,**10**]			

b)

arreglo a	insert(a,i)	swap(a,j-1,j)	(i,j)
[5,**4**,3,2,1]	insert(a,2)	swap(a,1,2)	(2,2)
[4,**5**,3,2,1]			(2,1)
[4,5,**3**,2,1]	insert(a,3)	swap(a,2,3)	(3,3)
[4,3,**5**,2,1]		swap(a,1,2)	(3,2)
[3,4,**5**,2,1]			(3,1)
[3,4,5,**2**,1]	insert(a,4)	swap(a,3,4)	(4,4)
[3,4,2,**5**,1]		swap(a,2,3)	(4,3)
[3,2,4,**5**,1]		swap(a,1,2)	(4,2)

[2,3,4,5,**1**]	insert(a,5)	swap(a,4,5)	(5,5)
[2,3,4,1,**5**]		swap(a,3,4)	(5,4)
[2,3,1,4,**5**]		swap(a,2,3)	(5,3)
[2,1,3,4,**5**]		swap(a,1,2)	(5,2)
[1,2,3,4,**5**]			(5,1)

c)

arreglo a	insert(a,i)	swap(a,j-1,j)	(i,j)
[1, **2**, 3, 4, 5]	insert(a,2)		(2,2)
[1, 2, **3**, 4, 5]	insert(a,3)		(3,3)
[1, 2, 3, **4**, 5]	insert(a,4)		(4,4)
[1, 2, 3, 4, **5**]	insert(a,5)		(5,5)

8. Calcula 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
ops	1	2	3	3	4	4	4	4	5	5	5
t	1	2	4	4	8	8	8	8	16	16	16

n	12	13	14	15	16	17	18	19	20	21	…
ops	5	5	5	5	5	6	6	6	6	6	6
t	16	16	16	16	16	32	32	32	32	32	32

Graficando en dimensiones (eje x: n. eje y: ops)

Obs:

- ¿Cuál es la relación entre t y ops? t = 2^(ops-1). despejando ops = log_2(t) + 1
- ¿Cuál es la relación entre n y t? t es la potencia de 2 mas chica tq es ≥ n, son proporcionales crecen al mismo ritmo: n ~ t, entonces ops ~ log_2(n)

(b)

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

Obs: en el ejercicio anterior se quiere saber cuántas veces podemos multiplicar a t por 2 tq t < n. En este se busca ver cuántas veces podemos dividir a t por 2 tq t > 0. Es como ir de adelante para atrás en el inciso a), por lo tanto es el mismo orden. ops = log_2(n)

(c)

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

sabemos que el ciclo de adentro es ops = log_2(n), pero dado que tenemos un ciclo que itera en cada ops, esto es ops = log_2(i). De esta forma podemos plantear:
ops(for i := 1 to n do (t:= i) + log_2(i) od) ≡
σ i = 1..n (1 + log_2(i)) ≡ σ i = 1..n (log_2(i)) ≡ log_2(1) + …. + log_2(n)
por propiedad de logaritmo esto esto es
ops = log_2(1….n) = log_2(n!)

(d)

```
for i := 1 to n do
   t := i
   do t > 0
      t := t − 2
   od
od
```

Se quiere ver cuántas veces podemos restarle 2 a t tq t > 0. esto por algoritmo de división podemos plantear
t = 2.ops + r
despejamos
ops = (t - r)/2 ≡ t/2
esta forma de calcular ops es para cada i del bucle de afuera, por lo tanto, planteamos
ops(for i :=1 to n do t:=i ; t/2)
ops = σ i = 1.. n (ops(t:=i) + i/2)
ops =1/2 σ i = 1.. n (i)
ops =1/2 n(n+1)/2 = n(n+1)/4 = n^2/4 + n/4 = n^2

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

```
fun es_ordenado(a:array[1..n] of nat) ret r : bool
    r := true
    for i := 1 to n-1 do
       if (a[i] > a[i+1]) then
          r := false
       else
          skip
       fi
     od
end fun
```

- Queremos contar las operaciones de **comparación**: observar que la única comparación está en la guarda del if. Podemos contar de adentro hacia afuera.

- Empecemos con el if:

ops(if a[i] > a[i+1] then S1 else S2 fi)
≡ ops(a[i] > a[i+1]) + 0 (ninguna de las ramas tiene una comparación S1 o S2)
≡ 1

- Seguimos con el for:

ops(for i:= 1 to n-1 do IF..FI od)
≡ σ i = 1 ..(n - 1) (ops (IF..FI))
≡ σ i = 1 .. (n-1) 1
≡ (n - 1)

- no nos interesa la inicialización pq no es una comparación
- Resultado final: n - 1 comparaciones

OTRA VERSIÓN:

```
fun es_ordenado(a:array[1..n] of nat) r : bool
    var i : nat
    i := 1
    while i < n ^ a[i] ≤  a[i+1] do
       i := i + 1
    od
    r := (i == n)
end fun
```

- **Aclaración:** de las comparaciones, solo nos interesan las comparaciones entre elementos del arreglo (no la que dice i < n). Es lo mismo que hacemos cuando analizamos los algoritmos de ordenación.
- si nos preguntan orden, en este caso debemos calcular la cantidad de operaciones en el **peor caso** posible

- **Peor caso**: (Cuando el while itera la cantidad máxima posible de veces) ¿Cuando? Cuando el arreglo está ordenado y i llega a valer n, o sea, llegamos a recorrer todo el arreglo. (todo el arreglo está ordenado salvo quizás el penúltimo con el último elemento) ¿Cuántas comparaciones son? (n - 1)

- **Mejor caso:** cuando el While itera la cantidad mínima posible de veces, esto sucede cuando el primer elemento es ≥ que el segundo (a[0] ≥ 0) o si el arreglo tiene 1 elemento. Comparaciones 1.

Contando operaciones cuando hay un if:

A veces (o casi siempre) no puedo saber “a priori” lo que vale la guarda. En estos casos, nos quedamos con la cantidad de operaciones de la rama que sea “más lenta”.

ops(if b then C else D fi) = ops(b) + max(ops(C), ops(D))

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

arreglo a	proc q valor de i	proc r valor de j	swap(a,j+1,j)
[1,4,9,**6**,2]	i = 4	j = 4	swap(a,5,4)
[1,4,9,**2**,6]		j = 5	
[1,4,**9**,2,6]	i = 3	j = 3	swap(a,4,3)
[1,4,**2**,9,6]		j = 4	swap(a,5,4)

[1,4,**2**,6,9]		j = 5	
[1,**4**,2,6,9]	i = 2	j = 2	swap(a,3,2)
[1,**2**,4,6,9]		j = 3	
[**1**,2,4,6,9]	i = 1	j = 1	
[1,2,4,6,9]			

- **¿Qué hace q?** Ordena una secuencia de números de forma ascendente

- **¿Cómo lo hace?** Recorre la secuencia desde la anteúltima posición hasta la primera (de derecha a izquierda). Con un proceso auxiliar, le manda el índice actual y la secuencia entera para que este ordene la secuencia comparando elementos, si un elemento es mayor al siguiente, se intercambian posiciones.

- **¿Qué hace r?** Ordena una secuencia de números desde un índice dado hasta el anteúltimo elemento de la secuencia.

- **¿Cómo lo hace?** Toma un índice dado y recorre la secuencia hasta la anteúltima posición, compara un elemento con el siguiente, y siempre que se cumpla que este sea mayor al siguiente, intercambia de posiciones con swap. Finalmente aumenta el índice actual para seguir con el siguiente elemento mientras siga siendo menor que la última posición. Si alguna de las dos cosas no se cumple, el recorrido no se ejecuta y el procedimiento pide otra indice.