Práctico 1.1 - Ord. Elemental

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

El mas adecuado es el encabezado numero 2, ya que como vamos a inicializar un array, no me interesa lo que tiene, así que no lo quiero leer, solo lo quiero escribir y retornar. \

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

```
proc incializarNumNat(out a : array[1..n] of
    for i ≔ 0 to n do
        a[i] ≔ i
    od
end proc
```

En este caso también es mas adecuado el encabezado 2, por la misma justificación que el anterior.

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

```
proc inicializarNumNatImp(out a : array[1..n] of nat)
    for i ≔ 0 to n do
        a[i] ≔ 2 * i + 1 {- Asi nos aseguramos que es impar -}
    od
end proc

{- Usamos el encabezado 2 por la misma justificacion -}
```

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

```
proc incPosImpares(in/out a : array[1..n] of nat)
    for i ≔ 1 to n do
        if (i % 2 != 0) ->
            a[i] ≔ a[i] + 1
        fi
    od
end proc
```

En este caso si el adecuado es usar el encabezado 1, ya que vamos a tener que leer los datos del arreglo, lo vamos a modificar y vamos a retornar ese mismo arreglo modificado, entonces usamos (in/out)

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

Mil disculpas pero en el ejercicio 1 no decía usar for, y yo use for loco me hubiesen dicho antes.

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

```
fun sorted(a : array[1..n] of nat) ret res : bool
    res ≔ true
    for i ≔ 1 to n - 1 do  {- Vamos hasta n-1 si no nos pasariamos de las cotas -}
        res ≔ res ∧ a[i] ≤ a[i + 1]
    od
end fun
```

- **Que hace:** Determina si un arreglo esta ordenado o no. (Con ordenado me refiero a ordenado en orden ascendente)
- **Como lo hace:** Recorre el array y se va fijando si el elemento actual es menor o igual que el elemento que le sigue.

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

Recordemos como es que funciona el algoritmo de ordenación por selección:

- Recorre el arreglo desde la primera posición hasta la última.
- En cada iteración, se busca el elemento más pequeño del resto del arreglo (sin considerar los elementos ya ordenados).
- Se intercambia el elemento más pequeño con el elemento que se encuentra en la posición actual de la iteración.
- Se repiten los pasos 1 y 2 hasta que todos los elementos del arreglo estén ordenados.

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

   Paso 1:

   Elemento seleccionado: 1

   Arreglo después del intercambio: [1, 7, 10, 3, 4, 9, 5]

   Paso 2:

Elemento seleccionado: 3

Arreglo después del intercambio: [1, 3, 10, 7, 4, 9, 5]

Paso 3:

Elemento seleccionado: 4

Arreglo después del intercambio: [1, 3, 4, 7, 10, 9, 5]

Paso 4:

Elemento seleccionado: 5

Arreglo después del intercambio: [1, 3, 4, 5, 7, 10, 9]

Paso 5:

Elemento seleccionado: 7

Arreglo después del intercambio: [1, 3, 4, 5, 7, 9, 10]

Paso 6:

Elemento seleccionado: 9

Arreglo después del intercambio: [1, 3, 4, 5, 7, 9, 10]

El arreglo ordenado es: **[1, 3, 4, 5, 7, 9, 10]**.

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

   Ahora vamos a hacerlo mas rápido sin tanto detalles:

   - [5, 4, 3, 2, 1] (selecciono el 1)
   - [1, 4, 3, 2, 5] (selecciono el 2)
   - [1, 2, 3, 4, 5]

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

   Ya esta ordenado perrrooo 😎

5. Calcula 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 practico pueden ayudarte.

```
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
```

Para calcular el numero de asignaciones podemos hacer lo siguiente:

- El primer bucle se ejecuta n veces
- El segundo bucle se ejecuta n^2 veces
- El tercer bucle se ejecuta n^3 veces

Entonces para calcular el número total de asignaciones, multiplicamos el número de iteraciones de cada bucle por el número de asignaciones que se realizan dentro de cada iteración:

$$NumAsignaciones = 1 + n * n^2 * n^3 = 1 + n^6$$

```
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
```

$$\begin{aligned}
\text{Total} &= 1 + \text{ops}\left(t := 0 \text{ for } i := 1 \text{ to } n \text{ do for } j := 1 \text{ to } i \text{ do for } k := j \text{ to } j + 3 \text{ do } t := t + 1 \text{ od od od}\right) \\
&= 1 + \sum_{i=1}^{n} \text{ops}\left(\text{for } j := 1 \text{ to } i \text{ do for } k := j \text{ to } j + 3 \text{ do } t := t + 1 \text{ od od}\right) \\
&= 1 + \sum_{i=1}^{n} \left(\sum_{j=1}^{i} \text{ops}\left(\text{for } k := j \text{ to } j + 3 \text{ do } t := t + 1 \text{ od}\right) \right) \\
&= 1 + \sum_{i=1}^{n} \left(\sum_{j=1}^{i} 4 \right) \\
&= 1 + 4 \sum_{i=1}^{n} \left(\sum_{j=1}^{i} 1 \right) \\
&= 1 + 4 \sum_{i=1}^{n} i \\
&= 1 + 4 \left(\frac{n(n+1)}{2} \right) \\
&= 1 + 2n(n+1)
\end{aligned}$$

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
```

- **Que hace:**

- El procedimiento `p` realiza una ordenación parcial del arreglo `a`.
- La función `f` encuentra el índice del mayor elemento en el subarreglo `a[1..i].`
- `swap` intercambia este elemento con el elemento en la posición `i`.

- **Reescribiendo:**

```
proc sortArray (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
```

8. **Calcula el orden del numero de asignaciones a la variable t de los siguientes algoritmos**

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

Inicialmente, `t = 1`. En cada iteración, `t` se duplica:

Inicialmente, `t = 1`. En cada iteración, `t` se duplica:

- Iteración 1: t = 1 × 2 = 2
- Iteración 2: t = 2 × 2 = 4
- Iteración 3: t = 4 × 2 = 8
- Iteración 4: t = 8 × 2 = 16
- .....

En general, después de k iteraciones, t es igual a 2^k

El bucle se detendrá cuando t≥n.

Por lo tanto, necesitamos encontrar el valor de k tal que: 2^k ≥ n

Tomamos log2 en cada lado:

k ≥ log2(n)

Así que el bucle ejecuta aproximadamente log2(n) iteraciones.

En cada iteración del bucle, hay una asignación a la variable `t`:

Entonces, el número total de asignaciones a `t` dentro del bucle es aproximadamente log2(n)

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

Iteraciones:

- Iteración 1: t = n÷2
- Iteración 2: t = (n÷2)÷2 = n÷4
- Iteración 3: t = (n÷4)÷2 = n÷8
- ...

En general, después de k iteraciones, t es igual a n/2^k

El bucle se detendrá cuando t≤0

Pero como estamos usando división entera, se detendrá cuando t sea menor que 1, es decir, cuando n/2^k<1n / 2^k, lo cual se puede escribir como 2^k ≥ n

Entonces tamos log2 en ambos lados.

k ≥ log2(n)

Por lo tanto, el bucle ejecuta aproximadamente log2(n) iteraciones.

En cada iteración del bucle, hay una asignación a la variable `t`:

Entonces, el número total de asignaciones a `t` dentro del bucle es aproximadamente log2(n)

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

Para un valor dado de i, el bucle interno:

- Inicializa t a i
- Divide t por 2 en cada iteración hasta que t sea 0.

El número de iteraciones del bucle interno para un valor dado de i es el número de veces que puedes dividir i por 2 antes de llegar a 0.

Esto es aproximadamente log2(i)

El bucle externo se ejecuta n veces, una para cada valor de i de 1 a n.

Para calcular el número total de iteraciones del bucle interno a lo largo de todas las iteraciones del bucle externo, sumamos log2(i) para i de 1 a n:

$$\sum_{i=1}^{n} \log_2(i)$$

Esta suma es aproximadamente igual a log2(n!)

Usando la aproximación de Stirling para el factorial:

$$\log_2(n!) \approx \log_2(\sqrt{2\pi n}) + \log_2\left(\left(\frac{n}{e}\right)^n\right)$$

Simplificando:

$$\log_2(n!) \approx \frac{1}{2}\log_2(2\pi n) + n\log_2(n) - n\log_2(e)$$

Por lo tanto podemos aproximarlo como:

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

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

Para un valor dado de i, el bucle interno:

- Inicializa t a i.
- Resta 2 a t en cada iteración hasta que t sea menor o igual a 0.

El número de iteraciones del bucle interno para un valor dado de i es aproximadamente i/2

El bucle externo se ejecuta n veces, una para cada valor de i de 1 a n.

$$\sum_{i=1}^{n} \frac{i}{2} \text{ Simplificado: } \sum_{i=1}^{n} \frac{i}{2} = \frac{1}{2} \sum_{i=1}^{n} i = \frac{1}{2} \cdot \frac{n(n+1)}{2} = \frac{n(n+1)}{4}$$

10. **Descifra que hacen los siguientes algoritmos, explicar como lo hacen y reescribirlos asignando nombres adecuados a todos los identificadores. (mal)**

```
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
```

Algoritmo `r`:

- Recibe un array `a` y un índice `i`.
- Inicializa `j` con el valor de `i`.
- Mientras `j` sea menor que `n` y `a[j]` sea mayor que `a[j+1]`, intercambia los elementos `a[j]` y `a[j+1]` y luego incrementa `j`.

Muy parecido al bubble sort

Algoritmo `q`:

- Itera desde `n-1` hasta `1` y llama al proc `r` con el índice actual.
- Esencialmente, está realizando múltiples pasadas al bubble sobre el array, asegurando que el array se ordene completamente.

Renombrando:

```
proc bubbleSort(in/out array: array[1..n] of T)
    for index := n-1 downto 1 do
        bubbleUp(array, index)
    od
end proc
```

```
proc bubbleUp (in/out array: array[1..n] of T, in index: nat)
    var currentIndex: nat
    currentIndex := index
    do currentIndex < n ∧ array[currentIndex] > array[currentIndex+1] →
        swap(array, currentIndex+1, currentIndex)
        currentIndex := currentIndex + 1
    od
end proc
```