Villar Pedro

Algoritmos 2 - TP

Prácticos

Índice general

1 Práctico 1 - Parte 1 **5**
- 1.1 Ejercicio 1 . . . 5
  - 1.1.1 Solución (a) . . . 5
  - 1.1.2 Solución (b) . . . 6
  - 1.1.3 Solución (c) . . . 6
  - 1.1.4 Solución (d) . . . 6
- 1.2 Ejercicio 2 . . . 6
  - 1.2.1 Solución (a) . . . 6
  - 1.2.2 Solución (b) . . . 7
  - 1.2.3 Solución (c) . . . 7
  - 1.2.4 Solución (d) . . . 7
- 1.3 Ejercicio 3 . . . 7
  - 1.3.1 Solución . . . 7
- 1.4 Ejercicio 4 . . . 8
  - 1.4.1 Solución (a) . . . 8
  - 1.4.2 Solución (b) . . . 8
  - 1.4.3 Solución (c) . . . 9
- 1.5 Ejercicio 5 . . . 9
  - 1.5.1 Solución (a) . . . 9
  - 1.5.2 Solución (b) . . . 10
- 1.6 Ejercicio 6 . . . 11
  - 1.6.1 Solución (a) . . . 12
  - 1.6.2 Solución (b) . . . 12
- 1.7 Ejercicio 7 . . . 12
  - 1.7.1 Solución (a) . . . 12
  - 1.7.2 Solución (b) . . . 12
  - 1.7.3 Solución (c) . . . 12
- 1.8 Ejercicio 8 . . . 12
  - 1.8.1 Solución (a) . . . 14
  - 1.8.2 Solución (b) . . . 14
  - 1.8.3 Solución (c) . . . 14
  - 1.8.4 Solución (d) . . . 14
- 1.9 Ejercicio 9 . . . 14
  - 1.9.1 Solución . . . 15
- 1.10 Ejercicio 10 . . . 15
  - 1.10.1 Solución (a) . . . 15
  - 1.10.2 Solución (b) . . . 16

2 Práctico 1 - Parte 2 **17**
- 2.1 Ejercicio 1 . . . 17
  - 2.1.1 Solución (a) . . . 17
- 2.2 Ejercicio 2 . . . 18
  - 2.2.1 Solución (a) . . . 18
  - 2.2.2 Solución (b) . . . 20

2.3 Ejercicio 3 . . . 20
2.3.1 Solución (a) . . . 20
2.3.2 Solución (b) . . . 21
2.4 Ejercicio 4 . . . 21
2.4.1 Solución . . . 21
2.5 Ejercicio 5 . . . 22
2.5.1 Solución . . . 23
2.6 Ejercicio 6 . . . 23
2.6.1 Solución . . . 23

3 Práctico 1 - Parte 3 **25**
3.1 Ejercicio 1 . . . 25
3.1.1 Solución (a) . . . 25
3.1.2 Solución (b) . . . 26
3.2 Ejercicio 2 . . . 27
3.2.1 Solución (a) . . . 27
3.2.2 Solución (b) . . . 28
3.2.3 Solución (c) . . . 28
3.2.4 Solución (d) . . . 29
3.3 Ejercicio 3 . . . 29
3.3.1 Solución . . . 29
3.4 Ejercicio 4 . . . 30
3.4.1 Solución (a) . . . 30
3.4.2 Solución (b) . . . 31

4 Práctico 2 - Parte 1 **32**
4.1 Ejercicio 1 . . . 32
4.1.1 Solución . . . 32
4.2 Ejercicio 2 . . . 32
4.2.1 Solución (a) . . . 33
4.2.2 Solución (b) . . . 34
4.2.3 Solución (c) . . . 34
4.2.4 Solución (d) . . . 35
4.2.5 Solución (e) . . . 36
4.3 Ejercicio 3 . . . 36
4.3.1 Solución (a) . . . 37
4.3.2 Solución (b) . . . 37
4.4 Ejercicio 4 . . . 37
4.4.1 Solución (a) . . . 38
4.4.2 Solución (b) . . . 38
4.5 Ejercicio 5 . . . 38
4.5.1 Solución (a) . . . 39
4.5.2 Solución (b) . . . 40
4.5.3 Solución (c) . . . 40
4.6 Ejercicio 6 . . . 41
4.6.1 Solución . . . 41
4.7 Ejercicio 7 . . . 41
4.7.1 Solución . . . 41

5 Práctico 2 - Parte 2 **42**
5.1 Ejercicio 1 . . . 42
5.1.1 Solución . . . 42
5.2 Ejercicio 2 . . . 45
5.2.1 Solución . . . 45
5.3 Ejercicio 3 . . . 46

5.3.1 Solución 46
5.4 Ejercicio 4 47
5.4.1 Solución (a) 47
5.4.2 Solución (b) 48
5.4.3 Solución (c) 50
5.5 Ejercicio 5 50
5.5.1 Solución 50

CAPÍTULO 1

PRÁCTICO 1 - PARTE 1

1.1 EJERCICIO 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.

1.1.1 Solución (a)

```
proc inicializarConCero (out a: array [1..n] of nat)
    i := 1
    do i <= n ->
    a[i] := 0
    i := i + 1
    od
end proc
```

1.1.2 Solución (b)

```
proc inicializarConNaturales (out a: array [1..n] of nat)
    i := 1
    do i <= n ->
    a[i] := i
    i := i + 1
    od
end proc
```

1.1.3 Solución (c)

```
proc inicializarConImpares (out a: array [1..n] of nat)
    i := 1
    do i <= n ->
    a[i] := 2 * i - 1
    i := i + 1
    od
end proc
```

1.1.4 Solución (d)

```
proc incrementarImpares (in/out a: array [1..n] of nat)
    i := 1
    do i <= n ->
    if i mod 2 = 1 then
        a[i] := a[i] + 1
    fi
    i := i + 1
    od
end proc
```

1.2 EJERCICIO 2

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

1.2.1 Solución (a)

```
proc inicializarConCero (out a: array [1..n] of nat)
    for i := 1 to n do
    a[i] := 0
    od
end proc
```

1.2.2 Solución (b)

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

1.2.3 Solución (c)

```
proc inicializarConImpares (out a: array [1..n] of nat)
    for i := 1 to n do
    a[i] := 2 * i - 1
    od
end proc
```

1.2.4 Solución (d)

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

1.3 EJERCICIO 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.

1.3.1 Solución

```
fun estaOrdenado (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
```

¿Qué hace el algoritmo? El algoritmo recibe un arreglo de números naturales y retorna un valor booleano que indica si el arreglo está ordenado o no.
¿Cómo lo hace? El algoritmo recorre el arreglo desde la primera posición hasta la penúltima, comparando cada elemento con el siguiente. Si encuentra un elemento mayor que el siguiente, retorna **false**, indicando que el arreglo no está ordenado. Si recorre todo el arreglo sin encontrar un elemento mayor que el siguiente, retorna **true**, indicando que el arreglo está ordenado.

1.4 EJERCICIO 4

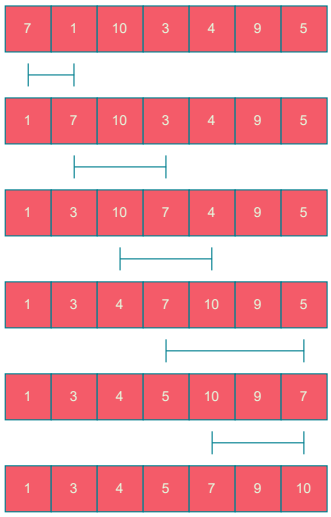
Ordená los siguientes arreglos, utilizando el algoritmo de ordenación por selección visto en clase. Mostrá en cada paso de iteración cuál es el elemento seleccionado y cómo queda el arreglo después de cada intercambio.

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

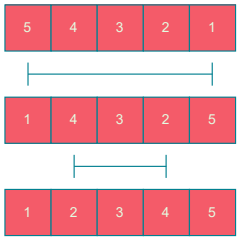
(b) $[5,4,3,2,1]$

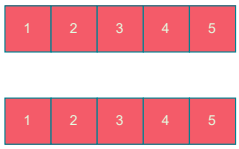
(c) $[1,2,3,4,5]$

1.4.1 Solución (a)



1.4.2 Solución (b)





1.4.3 Solución (c)

1.5 EJERCICIO 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.

```
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
```

```
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
```

1.5.1 Solución (a)

```
ops(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)
```

Escribo el bucle en forma de sumatorias

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{n^2}\right)\sum_{k=1}^{n^3} t := t+1\Big)\Big)$$

Ahora aplico la función ops a la secuencia:

$$ops(t := 0) + \sum_{i=1}^{n} \left(ops \left(\sum_{j=1}^{n^2} \sum_{k=1}^{n^3} t := t + 1 \right) \right)$$

= { $ops = \sum_{j=1}^{n^2}$ es una sumatoria de n^2 elementos, por lo que vale n^2 }

$$ops(t := 0) + \sum_{i=1}^{n} \left(n^2 \cdot ops \left(\sum_{k=1}^{n^3} t := t + 1 \right) \right)$$

= { $ops(t := 0)$ es una asignación, por lo que vale 1 }

$$1 + \sum_{i=1}^{n} \left(n^2 \cdot ops \left(\sum_{k=1}^{n^3} t := t + 1 \right) \right)$$

= { $ops \left(\sum_{k=1}^{n^3} t := t + 1 \right)$ es una sumatoria de n^3 elementos, por lo que vale n^3 }

$$1 + \sum_{i=1}^{n} \left(n^2 \cdot n^3 \right)$$

= { Resuelvo la sumatoria interna }

$$1 + \sum_{i=1}^{n} \left(n^5 \right)$$

= { Resuelvo la sumatoria externa }

$$1 + n \cdot n^5$$

= { Resuelvo la multiplicación }

$$1 + n^6$$

1.5.2 Solución (b)

```
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)
```

Escribo el bucle en forma de sumatorias

$$\sum_{i=1}^{n} \left(\sum_{j=1}^{i} \right) \sum_{k=j}^{j+3} t := t + 1 \Bigg) \Bigg)$$

Ahora aplico la función ops a la secuencia:

$$ops(t := 0) + \sum_{i=1}^{n} \left(ops \left(\sum_{j=1}^{i} \sum_{k=j}^{j+3} t := t + 1 \right) \right)$$

= { $ops = \sum_{j=1}^{i}$ es una sumatoria de i elementos, por lo que vale i }

$$ops(t := 0) + \sum_{i=1}^{n} \left(i \cdot ops \left(\sum_{k=j}^{j+3} t := t + 1 \right) \right)$$

= { $ops(t := 0)$ es una asignación, por lo que vale 1 }

$$1 + \sum_{i=1}^{n} \left(i \cdot ops \left(\sum_{k=j}^{j+3} t := t + 1 \right) \right)$$

= { $ops\left(\sum_{k=j}^{j+3} t := t+1\right)$ es una sumatoria de 4 elementos, por lo que vale 4 }

$$1 + \sum_{i=1}^{n} (4 \cdot i)$$

= { Resuelvo la sumatoria }

$$1 + 4 \cdot \sum_{i=1}^{n} (i)$$

= { Resuelvo la sumatoria }

$$1 + 4 \cdot \frac{n \cdot (n+1)}{2}$$

= { Resuelvo la multiplicación }

$$1 + 2 \cdot n \cdot (n+1)$$

= { Resuelvo la multiplicación }

$$1 + 2 \cdot n^2 + 2 \cdot n$$

1.6 EJERCICIO 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
```

1.6.1 Solución (a)

Algoritmo p: El algoritmo recibe un arreglo de elementos de tipo T y lo ordena de manera descendente. Para ello, recorre el arreglo desde la última posición hasta la segunda, en cada iteración busca el elemento más grande en el subarreglo que va desde la primera posición hasta la posición actual y lo intercambia con el elemento en la posición actual. Se podria escribir de la siguiente manera:

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

1.6.2 Solución (b)

Función f: La función recibe un arreglo de elementos de tipo T y un número natural i, y retorna la posición del elemento más grande en el subarreglo que va desde la primera posición hasta la posición i. Para ello, recorre el subarreglo desde la segunda posición hasta la posición i, en cada iteración compara el elemento actual con el elemento más grande encontrado hasta el momento y si el elemento actual es mayor, actualiza la posición del elemento más grande. Se podria escribir de la siguiente manera:

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

1.7 EJERCICIO 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.

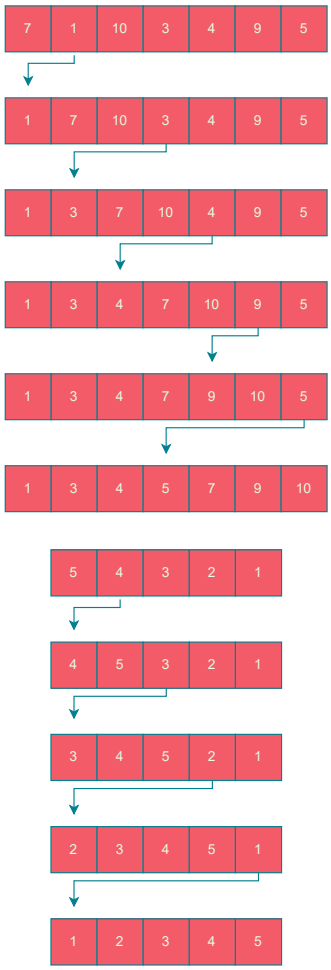
1.7.1 Solución (a)

1.7.2 Solución (b)

1.7.3 Solución (c)

1.8 EJERCICIO 8

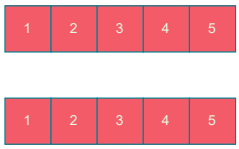
Calculá el orden del número de asignaciones a la variable t de los siguientes algoritmos.




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

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

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




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

1.8.1 Solución (a)

El algoritmo realiza una asignación y luego realiza una multiplicación en cada iteración. El número de iteraciones es el menor número k tal que $2^k \geq n$. Por lo tanto, el orden del número de asignaciones es $O(\log_2 n)$.

1.8.2 Solución (b)

El algoritmo realiza una asignación y luego realiza una división en cada iteración. El número de iteraciones es el menor número k tal que $n \div 2^k = 0$. Por lo tanto, el orden del número de asignaciones es $O(\log_2 n)$.

1.8.3 Solución (c)

El algoritmo realiza una asignación y luego realiza una división en cada iteración. El número de iteraciones es el número n. Por lo tanto, el orden del número de asignaciones es $O(n \cdot \log_2 n)$.

1.8.4 Solución (d)

El algoritmo realiza una asignación y luego realiza una resta en cada iteración. El número de iteraciones es el número n. Por lo tanto, el orden del número de asignaciones es $O(n)$.

1.9 EJERCICIO 9

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

1.9.1 Solución

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

El algoritmo realiza una comparación en cada iteración del bucle. El número de iteraciones es el menor número k tal que $i + k \geq n - 1$. Por lo tanto, el orden del número de comparaciones es $O(n)$.

$$ops\left(\sum_{i=1}^{n-1}(if...else)\right)$$

$$ops\left(\sum_{i=1}^{n-1}1\right) = n$$

1.10 EJERCICIO 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
```

1.10.1 Solución (a)

Algoritmo q: El algoritmo recibe un arreglo de elementos de tipo T y lo ordena de manera ascendente. Para ello, recorre el arreglo desde la penúltima posición hasta la primera, en cada iteración llama a la función r con el arreglo y la posición actual. Se podria escribir de la siguiente manera:

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

1.10.2 Solución (b)

Algoritmo r: El algoritmo recibe un arreglo de elementos de tipo T y un número natural i, y realiza un intercambio entre el elemento en la posición j y el elemento en la posición $j+1$ hasta que el elemento en la posición j sea menor o igual que el elemento en la posición $j+1$ o hasta que la posición j sea la última posición del arreglo. Se podria escribir de la siguiente manera:

```
proc ordenar (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
```

CAPÍTULO

2

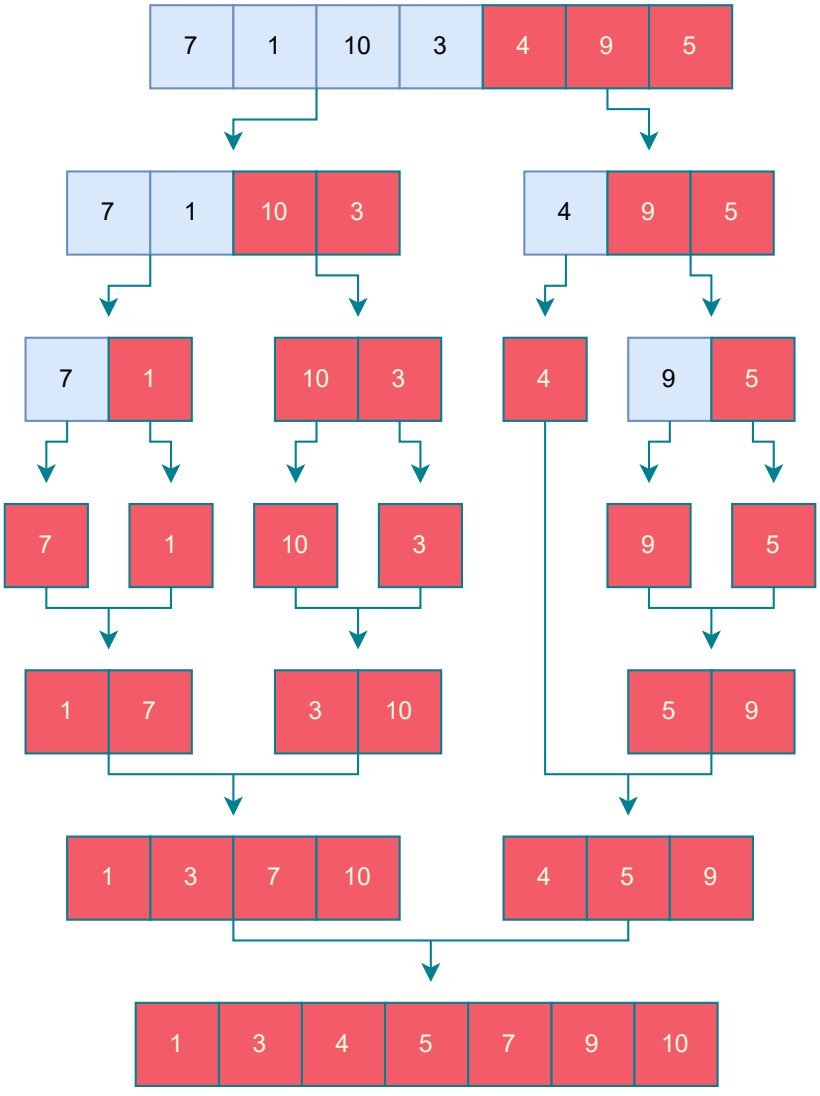
PRÁCTICO 1 - PARTE 2

2.1 EJERCICIO 1

a) Ordená los arreglos del ejercicio 4 del práctico anterior utilizando el algoritmo de ordenación por intercalación.

b) En el caso del inciso a) del ejercicio 4, dar la secuencia de llamadas al procedimiento merge_sort_rec con los valores correspondientes de sus argumentos.

2.1.1 Solución (a)



Iteración	Llamada	Condición rgt >lft	mid	a[lft,rgt]
0	merge_sort_rec(a,1,7)	True	4	[7,1,10,3,4,9,5]
1	merge_sort_rec(a,1,4)	True	2	[7,1,10,3]
2	merge_sort_rec(a,1,2)	True	1	[7,1]
3	merge_sort_rec(a,1,1)	False	-	[7]
4	merge_sort_rec(a,2,2)	False	-	[1]
2	merge(a,1,1,2)	-	1	[1,7]
2	merge_sort_rec(a,3,4)	True	3	[10,3]
3	merge_sort_rec(a,3,3)	False	-	[10]
4	merge_sort_rec(a,4,4)	False	-	[3]
2	merge(a,3,3,4)	-	3	[3,10]
0	merge_sort_rec(a,5,7)	True	6	[4,9,5]
1	merge_sort_rec(a,5,6)	True	5	[4,9]
2	merge_sort_rec(a,5,5)	False	-	[4]
3	merge_sort_rec(a,6,6)	False	-	[9]
1	merge(a,5,5,6)	-	5	[4,9]
1	merge_sort_rec(a,7,7)	False	-	[5]
1	merge(a,1,2,4)	-	2	[1,3,7,10]
0	merge(a,5,6,7)	-	6	[4,5,9]
0	merge(a,1,4,7)	-	4	[1,3,4,5,7,9,10]

2.2 EJERCICIO 2

a) Escribí el procedimiento "`intercalar_cada`"que recibe un arreglo $a : array[1.,2^n]of int$ y un número natural $i : nat$; e intercala el segmento $a[1,2^i]$ con $a[2^i+1,2*2^i]$, el segmento $a[2*2^i+1,3*2^i]$ con $a[3*2^i+1,4*2^i]$, etc. Cada uno de dichos segmentos se asumen ordenados. Por ejemplo, si el arreglo contiene los valores $3,7,1,6,1,5,3,4$ y se lo invoca con con $i = 1$ el algoritmo deberá devolver el arreglo $1,3,6,7,1,3,4,5$. Si se lo vuelve a invocar con este nuevo arreglo y con $i = 2$, devolverá $1,1,3,3,4,5,6,7$ que ya está completamente ordenado. El algoritmo asume que cada uno de estos segmentos está ordenado, y puede utilizar el procedimiento de intercalación dado en clase

b) Utilizar el algoritmo "`intercalar_cada`"para escribir una versión iterativa del algoritmo de ordenación por intercalación. La idea es que en vez de utilizar recursión, invoca al algoritmo del inciso anterior sucesivamente con $i = 0,1,2,3$, etc.

2.2.1 Solución (a)

Primero para definir el procedimiento, va a recibir un arreglo `a : array[1..`2^n`] of int` y un número natural $i : nat$:

```
proc intercalar_cada(in/out a: array[1..2^n] of Int, in i: nat)
...
end proc
```

para poder intercalar el arreglo, se va a tener que llamar a la función `merge`, que toma un arreglo, y 3 posiciones `lft`, `mid`, `rgt`. Entonces hay que definirlas:

```
proc intercalar_cada(in/out a: array[1..2^n] of Int, in i: nat)
    var lft, rgt, mid: nat
    ...
end proc
```

Se deberia agregar una variable natural para recorrer cada segmento del arreglo

```
proc intercalar_cada(in/out a: array[1..2^n] of Int, in i: nat)
    var lft, rgt, mid: nat {—Variables para llamar a merge—}
    var k: nat {—Variable para recorrer el arreglo—}
    ...
end proc
```

La idea principal es realizar la intercalación de segmentos del arreglo según el valor proporcionado `i`, cada segmento tiene un tamaño de 2^i elementos. Para ello se debe hacer un ciclo while que calcule cada uno de los índices para llamar a la función de intercalación.

```
proc intercalar_cada(in/out a: array[1..2^n] of Int, in i: nat)
    var lft, rgt, mid: nat {—Variables para llamar a merge—}
    var k: nat {—Variable para recorrer el arreglo—}
    while k ≤ 2^n do
    lft := ... {—Indice inicial del primer segmento—}
    mid := ... {—Indice final del primer segmento—}
    rgt := ... {—Indice final del segundo segmento—}
    merge(a,lft,mid,rgt) {—Llamada a la funcion para intercalar—}
    k := ...
    do
end proc
```

El índice lft será primero 1, siguiendo la estructura de los segmentos, deberia tomar el valor de $k * 2^i + 1$, luego el medio es $(j+1) * 2^i$ y el final es $(j+2) * 2^i$.

```
proc intercalar_cada(in/out a: array[1..2^n] of Int, in i: nat)
    var lft, rgt, mid: nat {—Variables para llamar a merge—}
    var k: nat {—Variable para recorrer el arreglo—}
    while k ≤ 2^n do
    lft := k * 2^i + 1 {—Indice inicial del primer segmento—}
    mid := (k+1) * 2^i {—Indice final del primer segmento—}
    rgt := (k+2) * 2^i {—Indice final del segundo segmento—}
    merge(a,lft,mid,rgt) {—Llamada a la funcion para intercalar—}
    k := k+2
    do
end proc
```

2.2.2 Solución (b)

```
proc intercalar_cada_iter (in/out array[1..2^n] of int)
    for i := 0 to n-1 do
    intercalar_cada(a,i)
    od
end proc
```

2.3 EJERCICIO 3

a) Ordená los arreglos del ejercicio 4 del práctico anterior utilizando el algoritmo de ordenación rápida.

b) En el caso del inciso a), dar la secuencia de llamadas al procedimiento quick_sort_rec con los valores correspondientes de sus argumentos.

2.3.1 Solución (a)

El arreglo es

7	1	10	3	4	9	5

El algoritmo QuickSort es un algoritmo de ordenamiento eficiente que utiliza la técnica de "divide y vencerás". Funciona seleccionando un elemento del arreglo como pivote y particionando el arreglo alrededor de ese pivote. Luego, ordena las dos particiones resultantes de forma recursiva.

1. Seleccionar un elemento como pivote. Por simplicidad, seleccionaremos el primer elemento del arreglo como pivote, es decir, 7.
2. Particionar el arreglo alrededor del pivote:
   - Elementos menores que el pivote van a la izquierda.
   - Elementos mayores que el pivote van a la derecha.
   - El pivote va en el lugar correcto.

   Después de la partición, el arreglo queda así: $[1,3,4,5,7,9,10]$.
3. Ordenar recursivamente las particiones izquierda y derecha:
   a) Partición izquierda: $[1,3,4,5]$
      1) Seleccionar pivote: 1
      2) Particionar: $[1,3,4,5]$ (no se necesita particionar más)
      3) Ordenar recursivamente las particiones vacías (caso base).

   b) Partición derecha: $[9,10]$
      1) Seleccionar pivote: 9
      2) Particionar: $[9,10]$ (no se necesita particionar más)
      3) Ordenar recursivamente las particiones vacías (caso base).
4. Combinar las particiones ordenadas: $[1,3,4,5,7,9,10]$.

Por lo tanto, el arreglo ordenado final es $[1,3,4,5,7,9,10]$.

2.3.2 Solución (b)

La secuencia de llamadas al procedimiento `quick_sort_rec` con los valores correspondientes de sus argumentos es la siguiente:

Llamada	Condición	Pivote	Arreglo
quick_sort_rec(a,1,7)	True	7	[7,1,10,3,4,9,5]
quick_sort_rec(a,1,4)	True	7	[7,1,10,3]
quick_sort_rec(a,1,2)	True	7	[7,1]
quick_sort_rec(a,1,1)	False	7	[7]
quick_sort_rec(a,2,2)	False	1	[1]
quick_sort_rec(a,3,4)	True	7	[10,3]
quick_sort_rec(a,3,3)	False	10	[10]
quick_sort_rec(a,4,4)	False	3	[3]
quick_sort_rec(a,5,7)	True	7	[4,9,5]
quick_sort_rec(a,5,6)	True	4	[4,9]
quick_sort_rec(a,5,5)	False	4	[4]
quick_sort_rec(a,6,6)	False	9	[9]
quick_sort_rec(a,7,7)	False	5	[5]

2.4 EJERCICIO 4

Escribí una variante del procedimiento partition que en vez de tomar el primer elemento del segmento `a[izq, der]` como pivot, elige el valor intermedio entre el primero, el último y el que se encuentra en medio del segmento. Es decir, si el primer valor es 4, el que se encuentra en el medio es 20 y el último es 10, el algoritmos deberá elegir como pivot al último.

2.4.1 Solución

Procedimiento `partition` original:

```
proc partition(in/out a: array[1..n] of T, in lft,rgt: nat, out ppiv: nat)
    var i,j: nat
    ppiv:= lft
    i:= lft+1
    j:= rgt
    do i <= j --> if a[i] <= a[ppiv] --> i:= i+1
                   a[j] >= a[ppiv] --> j:= j-1
                   a[i] > a[ppiv] && a[j] < a[ppiv] --> swap(a,i,j)
                fi
    od
    swap(a,ppiv,j)
    ppiv:= j
end proc
```

Procedimiento `partition` modificado:

```
proc partition(in/out a: array[1..n] of T, in lft,rgt: nat, out ppiv: nat)
    var i,j: nat
    {-Modificacion al tomar el pivot-}
    if a[lft] <= a[rgt] ->
    if a[lft] <= a[mid] ->
        if a[mid] <= a[rgt] ->
        ppiv := mid
        [] ->
        ppiv := rgt
        fi
        [] ->
        ppiv := lft
        fi
    [] ->
    if a[rgt] <= a[mid] ->
        ppiv := mid
    [] ->
        ppiv := rgt
    fi
    fi
    {-------------------------------------}
    i:= lft+1
    j:= rgt
    do i <= j -> if a[i] <= a[ppiv] -> i:= i+1
                    a[j] >= a[ppiv] -> j:= j-1
                    a[i] > a[ppiv] ∧ a[j] < a[ppiv] -> swap(a,i,j)
                fi
    od
    swap(a,ppiv,j)
    ppiv:= j
end proc
```

2.5 EJERCICIO 5

Escribí un algoritmo que dado un arreglo `a : array[1..n] of int` y un número natural $k \leq n$ devuelve el elemento de a que quedaría en la celda `a[k]` si a estuviera ordenado. Está permitido realizar intercambios en a, pero no ordenarlo totalmente. La idea es explotar el hecho de que el procedimiento partition del quick sort deja al pivot en su lugar correcto.

2.5.1 Solución

```
fun encontrarElemento(a: array[1..n] of int, k: nat): ret r : int
    var lft, rgt, ppiv: nat

    lft := 1
    rgt := n

    do lft < rgt -->
        partition(a, lft, rgt, ppiv)

        if ppiv = k -->
            r := a[ppiv]
        [] ppiv < k -->
            lft := ppiv + 1
        [] -->
            rgt := ppiv - 1
        fi
    od

    r := a[k]
end proc
```

2.6 EJERCICIO 6

El procedimiento `partition` que se dio en clase separa un fragmento de arreglo principalmente en dos segmentos: menores o iguales al pivot por un lado y mayores o iguales al pivot por el otro. Modificá ese algoritmo para que separe en tres segmentos: los menores al pivot, los iguales al pivot y los mayores al pivot. En vez de devolver solamente la variable `pivot`, deberá devolver `pivot izq` y `pivot der` que informan al algoritmo `quick_sort_rec` las posiciones inicial y final del segmento de repeticiones del `pivot`. Modificá el algoritmo `quick_sort_rec` para adecuarlo al nuevo procedimiento `partition`.

2.6.1 Solución

Procedimiento `partition` modificado:

```
proc partition(in/out a: array[1..n] of T, in lft, rgt: nat,
out pivotIzq, pivotDer: nat)
    var i, j, pivotPos: nat
    {-pivotPos rastrea la posicion actual del pivote-}
    pivotPos := lft
    i := lft + 1
    j := rgt

    do i <= j -->
        {-Casos para manejar elementos menores, mayores e iguales al pivote-}
        if a[i] < a[pivotPos] -->
            i := i + 1
        [] a[j] > a[pivotPos] -->
            j := j - 1
        [] a[i] > a[pivotPos] -->
            swap(a, i, j)
        {-Caso para manejar repeticiones del pivote-}
        [] -->
            swap(a, i, pivotPos)
            pivotPos := i
            i := i + 1
            j := j - 1
        fi
    od
    {-pivotIzq es la posicion inicial del segmento de repeticiones del pivote-}
    pivotIzq := lft
    {-pivotDer es la posicion final del segmento de repeticiones del pivote-}
    pivotDer := j
    {-Mover todas las repeticiones del pivote a posiciones contiguas despues de pivotIzq-}
    do j < rgt -->
        swap(a, j + 1, rgt)
        j := j + 1
    od
end proc

proc quick_sort_rec(in/out a: array[1..n] of T, in lft, rgt: nat)
    var pivotIzq, pivotDer: nat

    if lft < rgt -->
        partition(a, lft, rgt, pivotIzq, pivotDer)
        quick_sort_rec(a, lft, pivotIzq - 1)
        quick_sort_rec(a, pivotDer + 1, rgt)
    fi
end proc
```

CAPÍTULO

3 PRÁCTICO 1 - PARTE 3

3.1 EJERCICIO 1

Calculá el orden de complejidad de los siguientes algoritmos:

```
proc f1(in n : nat)
    if n ≤ 1 then skip
    else
        for i := 1 to 8 do
            f1(n div 2)
        od
        for i := 1 to n^3 do
            t := 1
        od
    fi
end proc
```

```
proc f2(in n : nat)
    for i := 1 to n do
        for j := 1 to i do
            t := 1
        od
    od
    if n > 0 then
        for i := 1 to 4 do
            f2(n div 2)
        od
    fi
end proc
```

3.1.1 Solución (a)

En este caso notar que:

- Tamaño de la entrada: n,
- Operación a contar: $t := 1$.

Entonces, se puede definir una funcion $r(n)$, que representará la cantidad de asignaciones a la variable t que ocurren al llamar a la función $f1$ con el dato de entrada n.
Podemos observar que la función $f1$ esta dividida en dos casos, si $n \leq 1$ entonces no se realiza ninguna asignación a la variable t, por lo que $r(n) = 0$.

$$r(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \dots & \text{en caso contrario} \end{cases}$$

Como hay dos ciclos for en una secuencia, se puede analizar cada uno por separado y sumarlos, por ahora los puedo expresar como sumatoria:

$$r(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \sum_{i=1}^{8} \ldots + \sum_{i=1}^{n^3} \ldots & \text{en caso contrario} \end{cases}$$

En el primer for, queremos contar la cantidad de asignaciones que se realizan al llamar a la función $f1$ con el dato de entrada $n/2$, por lo que se puede expresar como:

$$r(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \sum_{i=1}^{8} r(n/2) + \sum_{i=1}^{n^3} \ldots & \text{en caso contrario} \end{cases}$$

En el segundo for, se realiza una asignación a la variable t por cada iteración, por lo que se puede expresar como:

$$r(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \sum_{i=1}^{8} r(n/2) + \sum_{i=1}^{n^3} 1 & \text{en caso contrario} \end{cases}$$

Resolviendo las sumatorias, se obtiene:

$$r(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ 8 \cdot r(n/2) + n^3 & \text{en caso contrario} \end{cases}$$

Por lo que se puede observar que:

- $a = 8$,
- $b = 2$,
- $g(n) = n^3$.

Como $a = 8 = 2^3 = b^3$, se puede decir que el orden de complejidad es $O(n^3 \log n)$.

3.1.2 Solución (b)

En este caso notar que:

- Tamaño de la entrada: n,
- Operación a contar: $t := 1$.

Entonces, se puede definir una funcion $r(n)$, que representará la cantidad de asignaciones a la variable t que ocurren al llamar a la función $f2$ con el dato de entrada n.
En este caso la función $f2$ está dividida en n casos (cada uno representado por el valor de i):

$$r(n) = \begin{cases} \sum_{i=1}^{n} \sum_{j=1}^{i} 1 & \text{casos simples} \\ \ldots & \text{en caso contrario} \end{cases}$$

En el for dentro del if, se realiza una llamada recursiva a la función $f2$ con el dato de entrada $n/2$, por lo que se puede expresar como:

$$r(n) = \begin{cases} \sum_{i=1}^{n} \sum_{j=1}^{i} 1 & \text{casos simples} \\ \sum_{i=1}^{n} \sum_{j=1}^{i} 1 + \sum_{i=1}^{4} r(n/2) & \text{en caso contrario} \end{cases}$$

Resolviendo las sumatorias, se obtiene:

$$r(n) = \begin{cases} \sum_{i=1}^{n} i & \text{casos simples} \\ \sum_{i=1}^{n} i + \sum_{i=1}^{4} r(n/2) & \text{en caso contrario} \end{cases}$$

$$r(n) = \begin{cases} \frac{n \cdot (n+1)}{2} & \text{casos simples} \\ \frac{n \cdot (n+1)}{2} + 4 \cdot r(n/2) & \text{en caso contrario} \end{cases}$$

Por lo que se puede observar que:

- $a = 4$,
- $b = 2$,
- $g(n) = \frac{n \cdot (n+1)}{2}$.

Como $a = 4 = 2^2 = b^2$, se puede decir que el orden de complejidad es $O(n^2 \log n)$.

3.2 EJERCICIO 2

Dado un arreglo `a : array[1..n] of nat` se define una cima de a como un valor k en el intervalo $1,...,n$ tal que $a[1..k]$ está ordenado crecientemente y $a[k..n]$ está ordenado decrecientemente.

(a) Escribá un algoritmo que determine si un arreglo dado tiene cima.

(b) Escribí un algoritmo que encuentre la cima de un arreglo dado (asumiendo que efectivamente tiene una cima) utilizando una búsqueda secuencial, desde el comienzo del arreglo hacia el final.

(c) Escribí un algoritmo que resuelva el mismo problema del inciso anterior utilizando la idea de `búsqueda binaria`.

(d) Calculá y compará el orden de complejidad de ambos algoritmos.

3.2.1 Solución (a)

Para determinar si un arreglo tiene cima, se puede recorrer el arreglo y buscar el máximo que va a ser el probable punto cima.

```
fun tieneCima (a : array[1..n] of nat) ret r : bool
    r := false
    var tempos : nat
    {-Busco la posible cima y lo guardo en tempos-}
    for i := 2 to n do
        if a[i-1] < a[i] then
            tempos := i
            r := true
        fi
    od
    ...
end fun
```

Luego de encontrar un elemento, habria que verificar si el resto del arreglo cumple con la condición de cima.

```
fun tieneCima (a : array[1..n] of nat) ret r : bool
    r := false
    var tempos : nat
    {-Busco la posible cima y lo guardo en tempos-}
    for i := 2 to n do
        if a[i-1] < a[i] then
            tempos := i
            r := true
        fi
    od
    {-Si hay alguna posible cima, verifico que el resto arreglo cumpla-}
    for i := 1 to tempos do
        if a[i] < a[i+1] then
            r := false
        fi
    od
    for i := tempos to n do
        if a[i] > a[i+1] then
            r := false
        fi
    od
end fun
```

3.2.2 Solución (b)

Para encontrar la cima y devolverla, el algoritmo es el mismo que el anterior con la diferencia de que en vez de devolver el valor de r, devuelvo el valor del arreglo en la posición tempos (y la verificación de si hay cima o no se puede omitir ya que se asume que hay cima).

```
fun cima (a : array[1..n] of nat) ret c : nat
    var tempos : nat
    {-Busco la cima y lo guardo en tempos-}
    for i := 2 to n do
        if a[i-1] < a[i] then
            tempos := i
        fi
    od
    c := a[tempos]
end fun
```

3.2.3 Solución (c)

Para encontrar la cima y devolverla, se puede utilizar la idea de búsqueda binaria. Se puede buscar el punto cima de la siguiente manera:

- Se toma el punto medio del arreglo y se compara con el siguiente elemento.
- Si el punto medio es menor al siguiente elemento, entonces la cima se encuentra en la mitad derecha del arreglo.
- Si el punto medio es mayor al siguiente elemento, entonces la cima se encuentra en la mitad izquierda del arreglo.

```
fun busqueda_binaria_rec (a : array[1..n] of nat, lft, rgt: nat) ret i : nat
var mid : nat
if lft < rgt then
    i := 0
else
    mid := (lft + rgt) div 2
    if a[mid−1] ≥ a[mid] ∧ a[mid+1] ≥ a[mid] then
        i := busqueda_binaria_rec(a, lft, mid−1)
    fi
    if a[mid−1] ≤ a[mid] ∧ a[mid+1] ≤ a[mid] then
        i := mid
    fi
    if a[mid−1] ≤ a[mid] ∧ a[mid+1] ≥ a[mid] then
        i := busqueda_binaria_rec(a, mid+1, rgt)
    fi
fi
end fun

fun cima (a : array[1..n] of nat) ret c : nat
    c := busqueda_binaria_rec(a, 2, n−1)
end fun
```

3.2.4 Solución (d)

Para el algoritmo de búsqueda secuencial, se puede observar que el peor caso es cuando la cima se encuentra en la última posición del arreglo, por lo que se recorre todo el arreglo. Por lo que el orden de complejidad es $O(n)$.
Para el algoritmo de búsqueda binaria, se puede observar que el peor caso es cuando la cima se encuentra en la mitad del arreglo, por lo que se divide el arreglo en dos partes y se busca en una de ellas. Por lo que el orden de complejidad es $O(\log n)$.

3.3 EJERCICIO 3

El siguiente algoritmo calcula el mínimo elemento de un arreglo `a : array[1..n]` of nat mediante la técnica de programación divide y vencerás. Analizá la eficiencia de $minimo(1,n)$.

```
fun minimo(a : array[1..n] of nat, i, k : nat) ret m : nat
    if i = k then m := a[i]
    else
        j := (i + k) div 2
        m := min(minimo(a, i, j), minimo(a,j+1,k))
    fi
end fun
```

3.3.1 Solución

Para analizar la eficiencia de $minimo(1,n)$, se puede observar que:

- Tamaño de la entrada: n,
- Operación a contar: $m := a[i]$.

Entonces, se puede definir una función $r(n)$, que representará la cantidad de asignaciones a la variable m que ocurren al llamar a la función $minimo$ con el dato de entrada n.

Se puede observar que la función $minimo$ está dividida en dos casos, si $i = 1$ se realiza una sola asignación a m, por lo que $r(n) = 1$. Si se toma el largo del segmento como p:

$$r(n) = \begin{cases} 1 & \text{si } p = 1 \\ 1 + r(p/2) + r(p/2) & \text{en caso contrario} \end{cases}$$

Resolviendo la ecuación, se obtiene:

$$r(n) = \begin{cases} 1 & \text{si } p = 1 \\ 1 + 2r(p/2) & \text{en caso contrario} \end{cases}$$

Por lo que se puede observar que:

- $a = 2$,
- $b = 2$,
- $g(n) = 1$.

Como $a = 2 > b^k$ con $k = 0$, se puede decir que el orden de complejidad es $O(n^{\log_2 2}) = O(n)$.

3.4 EJERCICIO 4

Ordena usando $\sqsubset$ e $\approx$ los órdenes de las siguientes funciones. No calcules límites, utilizá las propiedades algebraicas.

(a) $n \log 2^n$, $2^n \log n$, $n! \log n$, 2^n,

(b) $n^4 + 2 \log n$, $\log(n^{n4})$, $2^{4 \log n}$, 4^n, $n^3 \log n$,

(c) $\log n!$, $n \log n$, $log(n^n)$.

3.4.1 Solución (a)

Para ordenar los órdenes de las funciones, se puede observar que:

$$1 < \log n$$
$$2^n < 2^n \log n$$

Se puede tomar cualquier base, entonces:

$$\log_2 2^n \approx n$$
$$\Rightarrow \log 2^n \approx \log_2 2^n \approx n$$

Habría que comparar 2^n con n^2:

$$\log 2^n \approx n$$
$$\Rightarrow n \cdot \log 2^n \approx n^2 < 2^n$$

Se sabe que $n! > 2^n$, entonces:

$$n! \log n > 2^n \log n$$

Y entonces se puede ordenar las funciones como:

$$n \log 2^n < 2^n < 2^n \log n < n! \log n$$

3.4.2 Solución (b)

Para ordenar los órdenes de las funciones, se puede observar que:

CAPÍTULO 4

PRÁCTICO 2 - PARTE 1

4.1 EJERCICIO 1

Escribir un algoritmo que dada una matriz `a: array[1..n,1..m] of int` calcule el elemento mínimo. Escribir otro algoritmo que devuelva un arreglo `array[1..n]` con el mínimo de cada fila de la matriz `a`.

4.1.1 Solución

```
fun min_elem_array(a : array[1..n,1..m] of int) ret min_elem int
    min_elem := a[1][1];
    for fil:= 1 to n do
        for col:= 1 to m do
            if a[fil][col] < min_elem then
                min_elem := a[col][fil];
            fi
        od
    od
end fun
```

```
fun array_of_mins(a: array[1..n,1..m] of int) ret array_min array[1..n] of int
for fil:= 1 to n do
    array_min[fil] := min_elem_array(a[fil, 1..m])
od
end fun
```

4.2 EJERCICIO 2

Dados los tipos enumerados

```
type mes = enumerate
            enero
            febrero
            ...
            diciembre
           end enumerate
```

```
type clima = enumerate
                Temp
                TempMax
                TempMin
                Pres
                Hum
                Prec
             end enumerate
```

El arreglo `med:array[1980..2016,enero..diciembre,1..28,Temp..Prec] of nat` es un arreglo multidimensional que contiene todas las mediciones estadísticas del clima para la ciudad de Córdoba desde el 1/1/1980 hasta el 28/12/2016. Por ejemplo, `med[2014,febrero,3,Pres]` indica la presión atmosférica que se registró el día 3 de febrero de 2014. Todas las mediciones están expresadas con números enteros. Por simplicidad asumiremos que todos los meses tienen 28 días.

(a) Dar un algoritmo que obtenga la menor temperatura mínima (TempMin) histórica registrada en la ciudad de Córdoba según los datos del arreglo.

(b) Dar un algoritmo que devuelva un arreglo que registre para cada año entre 1980 y 2016 la mayor temperatura máxima (TempMax) registrada durante ese año.

(c) Dar un algoritmo que devuelva un arreglo que registre para cada año entre 1980 y 2016 el mes de ese año en que se registró la mayor cantidad mensual de precipitaciones (Prec).

(d) Dar un algoritmo que utilice el arreglo devuelto en el inciso anterior (además de med) para obtener el año en que ese máximo mensual de precipitaciones fue mínimo (comparado con los de otros años).

(e) Dar un algoritmo que obtenga el mismo resultado sin utilizar el del inciso (c)

4.2.1 Solución (a)

```
fun min_tempMin (a:array[1980..2016,enero..diciembre,1..28,Temp..Prec] of nat)
    ret min_temp int
temp_min:= a[1980,enero,1,TempMin]
for a:= 1980 to 2016 do
    for m:= enero to diciembre do
        for d:= 1 to 28 do
            if (a[a,m,d,TempMin] < temp_min) then
                temp_min:= a[a,m,d,TempMin]
            fi
        od
    od
od
end fun
```

4.2.2 Solución (b)

```
fun temp_max_a(a:array[1980..2016,enero..diciembre,1..28,Temp..Prec] of nat)
    ret res:array[1980..2016] of int
var max_a_temp: int
for a:= 1980 to 2016 do
    max_a_temp:= a[a,1,1,TempMax]
    for mes:= enero to diciembre do
        for dia:= 1 to 28 do
        if(max_a_temp < a[a,m,d,TempMax]) then
            res[a]:= a[a,m,d,TempMax]
        fi
        od
    od
od
end fun
```

4.2.3 Solución (c)

```
fun mes_max_prec(a:array[1980..2016,enero..diciembre,1..28,Temp..Prec] of nat)
    ret res:array[1980..2016] of mes
var max_mes : mes
var max_mes_prec, mas_prec : nat

for a := 1980 to 2016 do
    {-calcular max_mes para cada anio-}
    max_mes_prec := 0
    for mes := enero to diciembre do
        {-calcular la suma prec_mes para cada mes-}
        prec_mes := 0
        for dia := 1 to 28 do
            prec_mes := prec_mes + a[a,mes,dia,Prec]
        od
        if prec_mes >= max_mes_prec then
            max_mes_prec := prec_mes
            max_mes := mes
        fi
    od
    res[a] := max_mes
od
end fun
```

4.2.4 Solución (d)

```
fun min_prec_mes(a:array[1980..2016,enero..diciembre,1..28,Temp..Prec] of nat)
ret res_a: int
var meses: array[1980..2016] of string
var prec_meses: array[1980..2016] of string
meses:= mes_may_prec(a)
for mes := enero to diciembre do
        {-calcular la suma prec_mes para cada mes-}
        prec_mes := 0
        for dia := 1 to 28 do
            prec_mes := prec_mes + a[a,mes,dia,Prec]
        od
        if prec_mes >= max_mes_prec then
            max_mes_prec := prec_mes
            max_mes := mes
        fi
    od
var min_prec: int
min_prec:= prec_meses[1980]
res_a:= 1980
for a:= 1981 to 2016 do
    if(prec_meses[a] < min_prec) then
        min_prec:= prec_meses[a]
        res_a:= a
    fi
od
end fun
```

4.2.5 Solución (e)

```
fun min_prec_mes_2(a:array[1980..2016,enero..diciembre,1..28,Temp..Prec] of nat)
    ret res:array[1980..2016] of int
{- primer algoritmo-}
var res_tmp, prec_mes: int
for an:= 1980 to 2016 do
    res_tmp:= 0
    for mes:= enero to diciembre do
        prec_mes:= 0
        for dia:= 1 to 28 do
        prec_mes:= prec_mes + a[an,mes,dia,Prec]
        od
        if(res_tmp < prec_mes) then
        res_parte1[an]:= mes
        res_tmp:= prec_mes
    od
od
{-segundo algoritmo-}
var meses: array[1980..2016] of string
var prec_meses: array[1980..2016] of string
for an:= 1980 to 2016 do
    for dia:= 1 to 28 do
        prec_mes:= prec_mes + a[an,res_parte1[n],dia,Prec]
    od
    prec_meses[an]:= prec_mes
od
var min_prec: int
min_prec:= prec_meses[1980]
res_an:= 1980
for an:= 1981 to 2016 do
    if(prec_meses[an] < min_prec) then
        min_prec:= prec_meses[an]
        res_an:= an
    fi
od
end proc
```

4.3 EJERCICIO 3

Dado el tipo

```
type person = tuple
     name: string
     age: nat
     weight: nat
    end
```

(a) escribí un algoritmo que calcule la edad y peso promedio de un arreglo `a : array[1..n] of person`.

(b) escribí un algoritmo que ordene alfabéticamente dicho arreglo.

4.3.1 Solución (a)

```
proc prom_edad_peso(in a:array [1..n] of person, out prom_edad: float,
    out prom_peso: float)
prom_edad := 0
prom_peso := 0
for i := 1 to n do
    prom_edad := prom_edad + a[i].age
    prom_peso := prom_peso + a[i].weight
od
prom_edad := prom_edad / n
prom_peso := prom_peso / n
end proc
```

4.3.2 Solución (b)

Voy a necesitar una función que compare dos strings y devuelva verdadero si el primero es menor alfabéticamente que el segundo.

```
fun menor(a:string, b:string) ret res:bool
if(s1[1] < s2[1]) then res:= true
else if(s1[1] = s2[1]) then
    res:= menor(s1[2..length(s1)], s2[2..length(s2)])
else res:= false
fi
end fun
```

Luego el algoritmo de ordenación selection sort aplicado a este caso sería:

```
proc ordenar(inout a:array[1..n] of person)
var min: int
for i:= 1 to n do
    min:= i
    for j:= i+1 to n do
        if(menor(a[j].name, a[min].name)) then
            min:= j
        fi
    od
    if(min != i) then
        swap(a[i], a[min])
    fi
od
end proc
```

4.4 EJERCICIO 4

Dados dos punteros `p, q : pointer to int`

(a) escribí un algoritmo que intercambie los valores referidos sin modificar los valores de p y q.

(b) escribí otro algoritmo que intercambie los valores de los punteros.

4.4.1 Solución (a)

```
proc swap_valores(p: pointer to int, q: pointer to int)
var aux: int
aux:= *p
*p:= *q
*q:= aux
end proc
```

$\star p$ es una expresión que me da el valor guardado en el lugar de memoria apuntado por p. si p es de tipo `pointer to T`, entonces el tipo de $\star p$ es `T`.

4.4.2 Solución (b)

```
proc swap_punteros(p: pointer to int, q: pointer to int)
var aux: pointer to int
aux:= p
p:= q
q:= aux
end proc
```

4.5 EJERCICIO 5

Dados dos arreglos `a, b : array[1..n] of nat` se dice que a es "`lexicográficamente menor`" que b si existe $k \in 1...n$ tal que `a[k] <b[k]`, y para todo $i \in 1...k-1$ se cumple `a[i] = b[i]`. En otras palabras, si en la primera posición en que a y b difierente, el valor de a es menor que el de b. También se dice que a es *"lexicográficamente menor o igual"* a b si a es lexicográficamente menor que b o a es igual a b.

(a) Escribir un algoritmo lex less que recibe ambos arreglos y determina si a es lexicográficamente menor que b.

(b) Escribir un algoritmo lex less or equal que recibe ambos arreglos y determina si a es lexicográficamente menor o igual a b.

(c) Dado el tipo enumerado

```
type ord = enumerate
             igual
             menor
             mayor
           end enumerate
```

Escribir un algoritmo lex compare que recibe ambos arreglos y devuelve valores en el tipo ord. ¿Cuál es el interés de escribir este algoritmo?

4.5.1 Solución (a)

```
fun lex_less(a,b : array[1..n] of nat) ret res: bool
res := false
if a[1] < b[1] then
    res := true
else
    for i:= 1 to n do
        if a[i] < b[i] then
            res := true
            break
        fi
    od
fi
end fun
```

Inicialización: Se declara una variable res y se inicializa a false. Esta variable almacenará el resultado final que indica si a es lexicográficamente menor que b. **Comparando los primeros elementos:**

- El código verifica si el primer elemento de a (indicado por a[1]) es menor que el primer elemento de b (indicado por b[1]).
- Si lo es, entonces res se establece inmediatamente a true. Esto se debe a que en la comparación lexicográfica, el arreglo con el primer elemento más pequeño se considera menor.

Iterando por los arreglos:

- Si los primeros elementos no son diferentes, el código entra en un ciclo que itera desde el índice 1 hasta n (la longitud de los arreglos).
- Dentro del ciclo, compara los elementos en el índice actual (i) de ambos arreglos (a[i] y b[i]).
  - Si se encuentra que un elemento en a es menor que el elemento correspondiente en b, entonces res se establece en true y el ciclo se detiene (break) usando la instrucción break. Esto se debe a que una vez que se encuentra una diferencia donde a tiene un elemento más pequeño que b, sabemos que a es lexicográficamente menor y no hay necesidad de seguir iterando.
  - Si se encuentra que un elemento en a es mayor que el elemento correspondiente en b, el ciclo termina inmediatamente devolviendo false. Esto se debe a que si a tiene un elemento mayor en cualquier punto, no puede ser lexicográficamente menor que b.

Resultado: Una vez que el ciclo termina de iterar por todos los elementos, si no se encuentra ninguna diferencia (a y b tienen los mismos elementos en todo), entonces res seguirá siendo false.

4.5.2 Solución (b)

```
fun lex_less(a,b : array[1..n] of nat) ret res: bool
res := false
if a[1] < b[1] then
    res := true
else
    for i:= 1 to n do
        if a[i] < b[i] then
            res := true
            break
        fi
    od
    if not res then
    res := true
    fi
fi
end fun
```

4.5.3 Solución (c)

```
fun lex_less(a,b : array[1..n] of nat) ret res: ord
var comp: ord
res := igual
if a[1] < b[1] then
comp := menor
else if a[1] > b[1] then
comp := mayor
else
for i:= 2 to n do
    if a[i] < b[i] then
        comp := menor
        break
    fi
    if comp = mayor then
        break
    fi
od
fi
res := comp
end fun
```

Comparación de primeros elementos: Se compara el primer elemento de a y b. Si a[1] <b[1], se establece comp en menor. Si a[1] >b[1], se establece comp en mayor. En caso de igualdad, se mantiene comp en igual.
Ciclo de comparación: Se recorren los elementos restantes de los arreglos. Si se encuentra un elemento en a menor que el correspondiente en b, se establece comp en menor y se sale del ciclo usando break. Si se encuentra un elemento en a mayor que el correspondiente en b, se establece comp en mayor y se sale del ciclo usando break. Si se recorre todo el ciclo sin encontrar diferencias (los elementos son iguales), comp mantiene el valor que le haya sido asignado en la comparación de los primeros elementos.
Asignación final: Al final, se asigna el valor de comp a la variable res, lo que garantiza que el resultado devuelto sea de tipo ord y represente correctamente la comparación lexicográfica entre a y b.
la función lex_less devolverá un valor de tipo ord que indica si a es lexicográficamente menor que b (menor), mayor que b (mayor), o igual a b (igual).

4.6 EJERCICIO 6

Escribir un algoritmo que dadas dos matrices `a, b: array[1..n,1..m]` of nat devuelva su suma.

4.6.1 Solución

```
fun suma_matric(a,b: array[1..n,1..m] of nat) ret res: array[1..n,1..m] of nat
for i:= 1 to n do
    for j:= 1 to m do
        res[i,j]:= a[i,j] + b[i,j]
    od
od
end fun
```

4.7 EJERCICIO 7

Escribir un algoritmo que dadas dos matrices `a: array[1..n,1..m] of nat` y `b: array[1..m,1..p]` of nat devuelva su producto.

4.7.1 Solución

```
fun producto_matric(a: array[1..n,1..m] of nat, b: array[1..m,1..p] of nat)
    ret res: array[1..n,1..p] of nat
for i:= 1 to n do
    for j:= 1 to m do
        suma := 0
        for k:= 1 to m do
            suma := suma + a[i,k] * b[k,j]
        od
        res[i,j] := suma
    od
od
end fun
```

CAPÍTULO 5

PRÁCTICO 2 - PARTE 2

5.1 EJERCICIO 1

Completá la implementación de listas dada en el teórico usando punteros.

5.1.1 Solución

```
implement List of T where
type Node of T = tuple
                   elem : T
                   next : pointer to (Node of T)
                 end tuple

type List of T = pointer to (Node of T)
```

```
fun empty() ret l : List of T
    l := null
end fun

proc add(in e : T, in/out l : List of T)
    var p : pointer to (Node of T)
    alloc(p)
    p->elem := e
    p->next := l
    l := p
end proc
```

```
fun is_empty (l : List of T) ret b : bool
    b := l = null
end fun

fun head (l : List of T) ret e : T
    e := l->elem
end fun

proc tail(in/out l : List of T)
    var p : pointer to (Node of T)
    p := l
    l := l->next
    free(p)
end proc

proc addr (in/out l : List of T, in e : T)
    var p : pointer to (Node of T)
    alloc(p)
    p->elem := e
    p->next := null
    var q : pointer to (Node of T)
    q := l
    while q->next != null do
        q := q->next
    end while
    q->next := p
end proc

fun length(l : List of T) ret n : nat
    var p : pointer to (Node of T)
    p := l
    n := 0
    while p != null do
        n := n + 1
        p := p->next
    end while
end fun

fun concat (in/out l : List of T, in l0 : List of T)
    var p : pointer to (Node of T)
    p := l
    while p->next != null do
        p := p->next
    end while
    p->next := l0
end fun

fun index (l : List of T, n : nat) ret e : T
    var p : pointer to (Node of T)
    p := l
    for i := 1 to n do
        p := p->next
    end for
    e := p->elem
end fun
```

```
proc take(in/out l : List of T, in n : nat)
    var p : pointer to (Node of T)
    p := l
    for i := 1 to n do
        p := p->next
    end for
    var q : pointer to (Node of T)
    q := p->next
    p->next := null
    while q != null do
        var r : pointer to (Node of T)
        r := q
        q := q->next
        free(r)
    end while
end proc

proc drop(in/out l : List of T, in n : nat)
    var p : pointer to (Node of T)
    p := l
    for i := 1 to n do
        p := p->next
    end for
    l := p
end proc

fun copy_list(l1 : List of T) ret l2 : List of T
    var p : pointer to (Node of T)
    var q : pointer to (Node of T)
    if l1 = null then
        l2 := null
    else
        alloc(q)
        q->elem := l1->elem
        q->next := null
        l2 := q
        p := l1->next
        while p != null do
            alloc(q->next)
            q := q->next
            q->elem := p->elem
            q->next := null
            p := p->next
        end while
    end if
end fun
```

```
proc destroy(in/out l : List of T)
    var p : pointer to (Node of T)
    while l != null do
        p := l
        l := l->next
        free(p)
    end while
end proc
```

5.2 EJERCICIO 2

Dada una constante natural N, implementá el TAD Lista de elementos de tipo T, usando un arreglo de tamaño N y un natural que indica cuántos elementos del arreglo son ocupados por elementos de la lista. ¿Esta implementación impone nuevas restricciones? ¿En qué función o procedimiento tenemos una nueva precondición?

5.2.1 Solución

```
implement List of T where
type List of T = tuple
                   elems : array 1..N of T
                   n : nat
                 end tuple
```

```
fun empty() ret l : List of T
    l.elems := []
    l.n := 0
end fun

proc addl(in e : T, in/out l : List of T)
    l.n := l.n + 1
    l.elems[l.n] := e
end proc

fun is_empty (l : List of T) ret b : bool
    b := l.n = 0
end fun

fun head (l : List of T) ret e : T
    e := l.elems[1]
end fun

proc tail(in/out l : List of T)
    for i := 1 to l.n - 1 do
        l.elems[i] := l.elems[i + 1]
    end for
    l.n := l.n - 1
end proc

proc addr (in/out l : List of T, in e : T)
    l.n := l.n + 1
    l.elems[l.n] := e
end proc

fun length(l : List of T) ret n : nat
    n := l.n
end fun
```

```
fun concat (in/out l : List of T, in l0 : List of T)
    for i := 1 to l0.n do
        l.n := l.n + 1
        l.elems[l.n] := l0.elems[i]
    end for
end fun

fun index (l : List of T, n : nat) ret e : T
    e := l.elems[n]
end fun

proc take(in/out l : List of T, in n : nat)
    for i := n + 1 to l.n do
        l.elems[i - n] := l.elems[i]
    end for
    l.n := l.n - n
end proc

proc drop(in/out l : List of T, in n : nat)
    for i := 1 to l.n - n do
        l.elems[i] := l.elems[i + n]
    end for
    l.n := l.n - n
end proc

fun copy_list(l1 : List of T) ret l2 : List of T
    l2.n := l1.n
    for i := 1 to l1.n do
        l2.elems[i] := l1.elems[i]
    end for
end fun

proc destroy(in/out l : List of T)
    l.n := 0
end proc
```

Esta implementación impone la restricción de que la cantidad de elementos de la lista no puede superar a N. La nueva precondición se encuentra en la operación `addl`, que no puede agregar un elemento si la lista ya está llena.

5.3 EJERCICIO 3

Implementá el procedimiento *add_at* que toma una lista de tipo T, un natural n, un elemento e de tipo T, y agrega el elemento e en la posición n, quedando todos los elementos siguientes a continuación. Esta operación tiene como precondición que n sea menor al largo de la lista.

5.3.1 Solución

```
proc add_at(in/out l : List of T, in n : nat, in e : T)
    for i := l.n downto n + 1 do
        l.elems[i + 1] := l.elems[i]
    end for
    l.elems[n] := e
    l.n := l.n + 1
end proc
```

1. **Argumentos:**
   - l (in/out): La lista que se modificará. La palabra clave in/out indica que se puede leer y escribir en l.
   - n (in): El índice en el que se insertará el nuevo elemento. Los valores válidos van de 0 a l.n.
   - e (in): El elemento que se insertará.
2. **Desplazamiento de elementos:**
   - La lógica central consiste en desplazar elementos en el arreglo elems para hacer espacio para el nuevo elemento en la posición n.
   - El ciclo for itera hacia atrás desde l.n hasta n + 1. En cada iteración:
     - l.elems[i + 1] := l.elems[i]: El elemento en el índice actual i se copia a la siguiente posición i + 1
   - Este ciclo esencialmente crea un espacio en la posición n + 1 al mover los elementos una posición hacia la derecha.
3. **Inserción del nuevo elemento:** Después del ciclo, l.elems[n] se asigna el nuevo elemento e. Esto coloca e en el índice de inserción deseado n.
4. **Actualización de la longitud de la lista:** Finalmente, l.n se incrementa en 1 para reflejar el nuevo tamaño de la lista.

5.4 EJERCICIO 4

(a) Especificá un TAD tablero para mantener el tanteador en contiendas deportivas entre dos equipos (equipo A y equipo B). Deberá tener un constructor para el comienzo del partido (tanteador inicial), un constructor para registrar un nuevo tanto del equipo A y uno para registrar un nuevo tanto del equipo B. El tablero sólo registra el estado actual del tanteador, por lo tanto el orden en que se fueron anotando los tantos es irrelevante.

Además se requiere operaciones para comprobar si el tanteador está en cero, si el equipo A ha anotado algún tanto, si el equipo B ha anotado algún tanto, una que devuelva verdadero si y sólo si el equipo A va ganando, otra que devuelva verdadero si y sólo si el equipo B va ganando, y una que devuelva verdadero si y sólo si se registra un empate.

Finalmente habrá una operación que permita anotarle un número n de tantos a un equipo y otra que permita "castigarlo" restándole un número n de tantos. En este último caso, si se le restan más tantos de los acumulados equivaldrá a no haber anotado ninguno desde el comienzo del partido.

(b) Implementá el TAD Tablero utilizando una tupla con dos contadores: uno que indique los tantos del equipo A, y otro que indique los tantos del equipo B.

(c) Implementá el TAD Tablero utilizando una tupla con dos naturales: uno que indique los tantos del equipo A, y otro que indique los tantos del equipo B. ¿Hay alguna diferencia con la implementación del inciso anterior? ¿Alguna operación puede resolverse más eficientemente?

5.4.1 Solución (a)

- **Nombre:** Tablero
- **Constructores:**
  - `tanteador_inicial()`: comienzo del partido.

```
proc tanteador_inicial(in/out t : Tablero)
```

- `anotar_A()`: registrar un nuevo tanto del equipo A.

```
proc anotar_A(in/out t : Tablero)
```

- `anotar_B()`: registrar un nuevo tanto del equipo B.

```
proc anotar_B(in/out t : Tablero)
```

- **Operaciones:**
  - Comprobar si el tanteador está en cero.
  - Comprobar si el equipo A ha anotado algún tanto.
  - Comprobar si el equipo B ha anotado algún tanto.
  - Devolver verdadero si y sólo si el equipo A va ganando.
  - Devolver verdadero si y sólo si el equipo B va ganando.
  - Devolver verdadero si y sólo si se registra un empate.
  - Anotarle un número n de tantos a un equipo.
  - "Castigarlo" restándole un número n de tantos.

5.4.2 Solución (b)

```
implement Tablero where
type Tablero = tuple
                  tantos_A : Counter
                  tantos_B : Counter
               end tuple
```

```
proc tanteador_inicial(in/out t : Tablero)
    init(t.tantos_A)
    init(t.tantos_B)
end proc

proc anotar_A(in/out t : Tablero)
    incr(t.tantos_A)
end proc

proc anotar_B(in/out t : Tablero)
    incr(t.tantos_B)
end proc
```

```
fun esta_en_cero(t : Tablero) ret b : bool
    b := is_init(t.tantos_A) ∧ is_init(t.tantos_B)
end fun

fun anoto_A(t : Tablero) ret b : bool
    b := !is_init(t.tantos_A)
end fun

fun anoto_B(t : Tablero) ret b : bool
    b := !is_init(t.tantos_B)
end fun

fun va_ganando_A(t : Tablero) ret b : bool
    b := t.tantos_A > t.tantos_B {-revisar-}
end fun

fun va_ganando_B(t : Tablero) ret b : bool
    b := t.tantos_B > t.tantos_A {-revisar-}
end fun

fun empate(t : Tablero) ret b : bool
    b := t.tantos_A = t.tantos_B {-revisar-}
end fun

proc anotar_n_tantos(in/out t : Tablero, in n : nat, in equipo : char)
    if equipo = 'A' then
        for i := 1 to n do
            inc(t.tantos_A)
        end for
    else
        for i := 1 to n do
            inc(t.tantos_B)
        end for
    end if
end proc

proc castigar(in/out t : Tablero, in n : nat, in equipo : char)
    if equipo = 'A' then
        for i := 1 to n do
            if not is_zero(t.tantos_A) then
                dec(t.tantos_A)
            end if
        end for
    else
        for i := 1 to n do
            if not is_zero(t.tantos_B) then
                dec(t.tantos_B)
            end if
        end for
    end if
end proc
```

5.4.3 Solución (c)

```
implement Tablero where
type Tablero = tuple
                    tantos_A : nat
                    tantos_B : nat
               end tuple
```

La diferencia es que ahora los tantos son naturales en lugar de contadores. La operación `inc` y `dec` no son necesarias, ya que se puede incrementar o decrementar directamente los valores.

5.5 EJERCICIO 5

Especificá el TAD Conjunto finito de elementos de tipo T. Como constructores considerá el conjunto vacío y el que agrega un elemento a un conjunto. Como operaciones: una que chequee si un elemento e pertenece a un conjunto c, una que chequee si un conjunto es vacío, la operación de unir un conjunto a otro , insersectar un conjunto con otro y obtener la diferencia. Estas últimas tres operaciones deberían especificarse como procedimientos que toman dos conjuntos y modifican el primero de ellos.

5.5.1 Solución

- **Nombre:** Conjunto
- **Constructores:**
  - conjunto vacío.

    ```
    fun vacio() ret c : Conjunto
    ```

  - agrega un elemento a un conjunto.

    ```
    proc agregar(in e : T, in/out c : Conjunto)
    ```

- **Operaciones:**
  - chequea si un elemento e pertenece a un conjunto c.

    ```
    fun pertenece(in e : T, in c : Conjunto) ret b : bool
    ```

  - chequea si un conjunto es vacío.

    ```
    fun es_vacio(in c : Conjunto) ret b : bool
    ```

  - unir un conjunto a otro.

```
proc unir(in c1 : Conjunto, in c2 : Conjunto,
    in/out c3 : Conjunto)
```

- intersectar un conjunto con otro.

```
proc interseccion(in c1 : Conjunto, in c2 : Conjunto,
    in/out c3 : Conjunto)
```

- obtener la diferencia.

```
proc diferencia(in c1 : Conjunto, in c2 : Conjunto,
    in/out c3 : Conjunto)
```

La especificación completa queda:

```
spec Conjunto where

constructors
    fun vacio() ret c : Conjunto
    proc agregar(in e : T, in/out c : Conjunto)

destroy
    proc destroy(in/out c : Conjunto)

operations
    fun pertenece(in e : T, in c : Conjunto) ret b : bool
    fun es_vacio(in c : Conjunto) ret b : bool
    proc unir(in c1 : Conjunto, in c2 : Conjunto, in/out c3 : Conjunto)
    proc interseccion(in c1 : Conjunto, in c2 : Conjunto, in/out c3 : Conjunto)
    proc diferencia(in c1 : Conjunto, in c2 : Conjunto, in/out c3 : Conjunto)
```