1. (a) Ordena 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.

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

7	1	10	3	4	9	5
7	1	10	3	4	9	5
7	1	10	3	4	9	5
7	1	10	3	4	9	5
7	1	10	3	4	9	5
7	1	10	3	4	9	5
1	7	10	3	4	9	5
1	7	10	3	4	9	5
1	7	10	3	4	9	5
1	7	10	3	4	9	5
1	7	3	10	4	9	5
1	3	7	10	4	9	5
1	3	7	10	4	9	5
1	3	7	10	4	9	5
1	3	7	10	4	9	5
1	3	7	10	4	9	5
1	3	7	10	4	9	5
1	3	7	10	4	9	5
1	3	7	10	4	5	9
1	3	4	5	7	9	10

[5, 4, 3, 2, 1]

5	4	3	2	1
5	4	3	2	1
5	4	3	2	1
5	4	3	2	1
5	4	3	2	1
5	4	3	2	1
4	5	3	2	1
4	5	3	2	1
4	5	3	2	1
3	4	5	2	1
3	4	5	2	1
3	4	5	2	1
3	4	5	2	1
3	4	5	2	1
3	4	5	1	2
1	2	3	4	5

[1, 2, 3, 4, 5]

1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5

1c)

[7,1,10,3,4,9,5] , n=7
primera llamada a merge_sort_rec(a,1,7)
lft = 1, rgt = 7, mid = 4
1.1 merge_sort_rec(a,1,4)
lft = 1, rgt = 4, mid = 2
1.1.1 merge_sort_rec(a,1,2)
lft = 1, rgt = 2, mid = 1
1.1.1.1 merge_sort_rec(a,1,1)
lft = 1, rgt = 1, mid = —
1.1.1.2 merge_sort_rec(a,2,2)
lft = 2, rgt = 2, mid = —
1.1.1.3 merge(a,1,1,2)
[1,7,10,3,4,9,5]
1.1.2 merge_sort_rec(a,3,4)
lft = 3, rgt = 4, mid = 3
1.1.2.1 merge_sort_rec(a,3,3)
lft = 3, rgt = 3, mid = —
1.1.2.2 merge_sort_rec(a,4,4)
lft = 4, rgt = 4, mid = —

1.1.2.3 merge(a,3,3,4)

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

1.1.3 merge_sort_rec(a,1,2,4)

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

1.2 merge_sort_rec(a,5,7)

lft = 5, rgt = 7, mid = 6

1.2.1 merge_sort_rec(a,5,6)

lft = 5, rgt = 6, mid = 5

1.1.2.1 merge_sort_rec(a,5,5)

lft = 5, rgt = 5, mid = —

1.1.2.2 merge_sort_rec(a,6,6)

lft = 6, rgt = 6, mid = —

1.1.2.3 merge(a,5,5,6)

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

1.2.2 merge_sort_rec(a,7,7)

lft = 7, rgt = 7, mid = —

1.2.3 merge(a,5,6,7)

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

1.3 merge(a,1,4,7)

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

2.

(a) Escribı 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[2i + 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.

```
proc intercalar_cada(in/out a:array[1..2^n] of int, in i : nat)
        var mid: nat
        var rgt: nat
        var lft: nat
        rgt := 2^(i+1)        //tamaño del arreglo en ese intervalo
        lft := 1
        mid := (rgt + lft) / 2
        while rgt ≤ 2^n do
                merge(a,lft,mid,rgt)
                lft := rgt + 1
                rgt := rgt + 2^(i+1)
                mid := (rgt + lft) / 2
        od
end proc
```

(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.

```
proc merge_sort_iterativo (in/out a: array[1..n] of T)
        for i := 1 to log_2(length(a)) do
                intercalar_cada(a,i)
        od
end proc
```

3.
(a) Ordena 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.

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

7	1	10	3	4	9	5
7	1	10	3	4	9	5
7	1	5	3	4	9	10
4	1	5	3	7	9	10
4	1	5	3	7	9	10
4	1	3	5	7	9	10
3	1	4	5	7	9	10
3	1	4	5	7	9	10
3	1	4	5	7	9	10
3	1	4	5	7	9	10
1	3	4	5	7	9	10
1	3	4	5	7	9	10
1	3	4	5	7	9	10
1	3	4	5	7	9	10
1	3	4	5	7	9	10
1	3	4	5	7	9	10
1	3	4	5	7	9	10
1	3	4	5	7	9	10

[5, 4, 3, 2, 1]

5	4	3	2	1
5	4	3	2	1
5	4	3	2	1
1	4	3	2	5
1	4	3	2	5
1	4	3	2	5
1	4	3	2	5
1	4	3	2	5
1	4	3	2	5
1	4	3	2	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5

[1, 2, 3, 4, 5]

1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5

b)

[7, 1, 10, 3, 4, 9, 5], n=7
primera llamada a quick_sort_rect(a,1,7)
lft = 1, rgt = 7, piv = 1
1.1 partition(a,1,7,1)
[4,1,5,3,7,9,10]
1.2 quick_sort_rec(a,1,4)
lft = 1, rgt = 4, piv = 1
1.1.1 partition(a,1,4,1)
[3,1,4,5,7,9,10]
1.1.2 quick_sort_rec(a,1,2)
lft = 1, rgt = 2, piv = 1
1.1.1.1 partition(a,1,2,1)
[1,3,4,5,7,9,10]
1.1.1.2 quick_sort_rec(a,1,1)
lft = 1, rgt = 1, piv = —
1.1.1.3 quick_sort_rec(a,3,2)
lft = 3, rgt = 2, piv = —
1.1.3 quick_sort_rec(a,4,4)
lft = 4, rgt = 4, piv = —
1.3 quick_sort_rec(a,6,7)
lft = 6, rgt = 7, piv = 6
1.2.1 partition(a,5,7,5)
[1,3,4,5,6,7,9,10]
1.2.2 quick_sort_rec(a,6,6)
lft = 6, rgt = 6, piv = —
1.2.3 quick_sort_rec(a,8,7)
lft = 6, rgt = 6, piv = —

[1,3,4,5,6,7,9,10]

4. Escribı 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 deber´a elegir como pivot al último

proc partition (in/out a: array[1..n] of T, in lft, rgt: nat, out ppiv: nat)

```
    var i,j,mid: nat
    mid := (rgt + lft)/2
    if(a[lft] ≤ a[mid] ≤ a[rgt] || a[rgt] ≤ a[mid] ≤ a[lft]) →
        swap(a,lft,mid)
    else(a[lft] ≤ a[rgt] ≤ a[mid] || a[mid] ≤ a[rgt] ≤ a[lft]) →
        swap(a,lft,rgt)
    fi
    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)
                                                    i:= i+1
                                                    j:= j-1
               fi
    od
    swap(a,ppiv,j)        {dejando el pivot en una posición más central}
    ppiv:= j              {señalando la nueva posición del pivot }
end proc
```

5. Escribí un algoritmo que ha dado un arreglo a : array[1..n] of int y un número natural k ≤ n devuelve el elemento de a que quedará 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 quicksort deja al pivot en su lugar correcto.

```
proc partition_k (in a:array[1..n] of T in k: nat, out result : nat)
    quick_sort_k (a,1,n,k)
end proc

proc quick_sort_k(in/out a : array[1..n] of T, in lft, rgt : nat, in/out k : nat, out result)
    var ppiv : nat
    if (rgt ≥ lft) →
        partition(a,lft,rgt,ppiv)
        if (ppiv = k) then
            result := a[ppiv]
        elseif (ppiv < k) then
            quick_sort_k(a,ppiv+1,rgt,k)
        else
            quick_sort_k(a,lft,ppiv-1,k)
        fi
    fi
end proc
```

testeo con [7, 1, 10, 3, 4, 9, 5] k = 4

[1,3,4,5,7,9,10] nos tiene que dar el número 5.

1.1 quick_sort_k(a,1,7,4)
lft = 1 rgt = 7
partition(a,1,7,ppiv)
[**4,1,5,3**,**7**,**9,10**]
ppiv = 5
1.1.1 quick_sort_k(a,1,4,4)
lft = 1, rgt = 4
partition(a,1,4,ppiv)
[1 3 4 5]
ppiv = 3
1.1.1.1 quick_sort_k(a,4,4,4) // (rgt > lft) = false
si ponemos rgt ≥ lft
partition(a,4,4,ppiv)
[5]
ppiv = 4
1.1.1.1 (piv = k)
rgt = 4, lft = 4, ppiv = 4
result = a[4]
consultar

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

```
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)
                                                    i:= i+1
                                                    j:= j-1
                fi
      od
      swap(a,ppiv,j) {dejando el pivot en una posición más central}
      ppiv:= j              {señalando la nueva posición del pivot }
end proc
```

Nuevo partition:

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

      od
      rgt_ppiv := lft_ppiv
      lft_ppiv := j
end proc
```

Nuevo quick_sort_rec

```
proc my_quick_sort_rec (in/out a: array[1..n] of T, in lft,rgt: nat)
      var lft_ppiv: nat
      var rgt_ppiv: nat
      if rgt > lft → my_partition(a,lft,rgt,lft_ppiv,rgt_ppiv)
                     my_quick_sort_rec(a,lft,ltf_ppiv-1)
                     my_quick_sort_rec(a,rgt_ppiv+1,rgt)
      fi
end proc
```