Ordenación avanzada

Práctico 1.2

Ejercicio 1

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

[7, 1, 10, 3, 4, 9, 5]
{divido el arreglo por la mitad}
[**7, 1, 10, 3,** 4, 9, 5]
{vuelvo a dividir la primera mitad del arreglo}
[**7, 1,** 10, 3, 4, 9, 5]
{divido el primer cuarto de arreglo}
[**7,** 1, 10, 3, 4, 9, 5]
{ordeno [7]}
[**7,** 1, 10, 3, 4, 9, 5]
{ordeno [1]}
[7, **1,** 10, 3, 4, 9, 5]
{ordeno [7,1]}
[**1, 7,** 10, 3, 4, 9, 5]
{divido el segundo cuarto de arreglo}
[1, 7, **10, 3,** 4, 9, 5]
{ordeno [10]}
[1, 7, **10,** 3, 4, 9, 5]
{ordeno [3]}
[1, 7, 10, **3,** 4, 9, 5]
{ordeno [10, 3]}
[1, 7, **3, 10,** 4, 9, 5]
{ordeno la primera mitad del arreglo([1, 7, 3, 10])}
[**1, 3, 7, 10,** 4, 9, 5]
{divido la segunda mitad del arreglo}
[1, 3, 7, 10, **4, 9, 5**]
{divido el tercer cuarto de arreglo}
[1, 3, 7, 10, **4, 9,** 5]
{ordeno [4]}
[1, 3, 7, 10, **4,** 9, 5]
{ordeno [9]}
[1, 3, 7, 10, 4, **9,** 5]
{ordeno [4, 9]}
[1, 3, 7, 10, **4, 9,** 5]
{ordeno el último cuarto del arreglo ([5])}
[1, 3, 7, 10, 4, 9, **5**]
{ordeno la segunda mitad del arreglo([4, 9, 5])}
[1, 3, 7, 10, **4, 5, 9**]
{ordeno la totalidad del arreglo}
[**1, 3, 4, 5, 7, 9, 10**]

[5, 4, 3, 2, 1]
{divido el arreglo por la mitad}
[**5, 4, 3,** 2, 1]
{vuelvo a dividir la primera mitad del arreglo}

[**5, 4,** 3, 2, 1]
{divido el primer tercio de arreglo}
[**5,** 4, 3, 2, 1]
{ordeno [5]}
[**5,** 4, 3, 2, 1]
{ordeno [4]}
[5, **4,** 3, 2, 1]
{ordeno [5, 4]}
[**4, 5,** 3, 2, 1]
{ordeno [3]}
[4, 5, **3,** 2, 1]
{ordeno la primera mitad del arreglo}
[**3, 4, 5,** 2, 1]
{divido la segunda mitad del arreglo}
[3, 4, 5, **2, 1**]
{ordeno [2]}
[3, 4, 5, **2,** 1]
{ordeno [1]}
[3, 4, 5, 2, **1**]
{ordeno la segunda mitad del arreglo}
[3, 4, 5, **1, 2**]
{ordeno el arreglo en su totalidad}
[**1, 2, 3, 4, 5**]

[1, 2, 3, 4, 5]
{divido el arreglo por la mitad}
[**1, 2, 3,** 4, 5]
{vuelvo a dividir la primera mitad del arreglo}
[**1, 2,** 3, 4, 5]
{divido el primer tercio de arreglo}
[**1,** 2, 3, 4, 5]
{ordeno [1]}
[**1,** 2, 3, 4, 5]
{ordeno [2]}
[1, **2,** 3, 4, 5]
{ordeno [1, 2]}
[**1, 2,** 3, 4, 5]
{ordeno [3]}
[1, 2, **3,** 4, 5]
{ordeno la primera mitad del arreglo}
[**1, 2, 3,** 4, 5]
{divido la segunda mitad del arreglo}
[1, 2, 3, **4, 5**]
{ordeno [4]}
[1, 2, 3, **4,** 5]
{ordeno [5]}
[1, 2, 3, 4, **5**]
{ordeno la segunda mitad del arreglo}
[1, 2, 3, **4, 5**]
{ordeno el arreglo en su totalidad}
[**1, 2, 3, 4, 5**]

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]
{merge_sort_rec(a, 1, 4)
 merge_sort_rec(a, 5, 7)}
[7, 1, 10, 3, 4, 9, 5]
{merge_sort_rec(a, 1, 2)
 merge_sort_rec(a, 3, 4)}
[7, 1, 10, 3, 4, 9, 5]
{merge_sort_rec(a, 1, 1)
 merge_sort_rec(a, 2, 2)}
[1, 7, 10, 3, 4, 9, 5]
{merge_sort_rec(a, 3, 3)
 merge_sort_rec(a, 4, 4)}
[1, 3, 7, 10, 4, 9, 5]
{merge_sort_rec(a, 5, 7)
 merge_sort_rec(a, 7, 7)}
[1, 3, 7, 10, 4, 9, 5]
{merge_sort_rec(a, 5, 6)
 merge_sort_rec(a, 6, 6)}
[1, 3, 4, 5, 7, 9, 10]
```

Ejercicio 2

a) Escribir 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.

```
proc intercalar_cada(in/out a: array[1..2^n] of T,
                     in i: nat)
    var izq, med, der, tam, largo_a: nat
    largo_a:= 2^n
    tam:= 2^(i+1)                    {- tamaño del intervalo a mergear -}
    izq:= 1
    der:= tam
    med:= der div 2
    while der ≤ largo_a do
        merge(izq, med, der)
        izq:= izq + tam
        der:= der + tam
        med:= der div 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 (in/out a: array[1..n] of T)
    var potencia: nat
    potencia:= log2n - 1
    intercalar_cada(a, potencia)
end proc
```

Ejercicio 3

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

[7, 1, 10, 3, 4, 9, 5]
{elijo el pivot (7)}
[**7**, 1, 10, 3, 4, 9, 5]
{ordeno entre mayores y menores}
[7, 1, 5, 3, 4, 9, 10]
{swap entre el pivot y su posición final}
[**4**, 1, 5, 3, **7**, 9, 10]
{elijo mi pivot (4)}
[**4**, 1, 5, 3, **7**, 9, 10]
{ordeno entre mayores y menores}
[**4**, 1, 3, 5, **7**, 9, 10]
{swap entre el pivot y su posición final}
[**3**, 1, **4**, 5, **7**, 9, 10]
{elijo mi pivot (3)}
[**3**, 1, **4**, 5, **7**, 9, 10]
{ordeno entre mayores y menores}
[**3**, 1, **4**, 5, **7**, 9, 10]
{swap entre el pivot y su posición final}
[**1**, **3**, **4**, 5, **7**, 9, 10]
{elijo mi pivot (5)}
[**1**, **3**, **4**, **5**, **7**, 9, 10]
{ordeno entre mayores y menores}
[**1**, **3**, **4**, 5, **7**, 9, 10]
{swap entre el pivot y su posición final}
[**1**, **3**, **4**, **5**, **7**, 9, 10]
{elijo mi pivot (9)}
[**1**, **3**, **4**, **5**, **7**, **9**, 10]
{ordeno entre mayores y menores}
[**1**, **3**, **4**, **5**, **7**, 9, 10]
{swap entre el pivot y su posición final}
[**1**, **3**, **4**, **5**, **7**, **9**, 10]
{el arreglo está ordenado}

[5, 4, 3, 2, 1]
{elijo mi pivot (5)}
[**5**, 4, 3, 2, 1]
{ordeno entre mayores y menores}

[**5**, 4, 3, 2, 1]
{swap entre el pivot y su posición final}
[**1**, 4, 3, 2, **5**]
{elijo mi pivot (1)}
[**1**, 4, 3, 2, **5**]
{ordeno entre mayores y menores}
[**1**, 4, 3, 2, **5**]
{swap entre el pivot y su posición final}
[**1**, 4, 3, 2, **5**]
{elijo mi pivot (4)}
[**1**, **4**, 3, 2, **5**]
{ordeno entre mayores y menores}
[**1**, **4**, 3, 2, **5**]
{swap entre el pivot y su posición final}
[**1**, **2**, 3, **4**, **5**]
{elijo mi pivot (2)}
[**1**, **2**, 3, **4**, **5**]
{ordeno entre mayores y menores}
[**1**, **2**, 3, **4**, **5**]
{swap entre el pivot y su posición final}
[**1**, **2**, 3, **4**, **5**]
{el arreglo está ordenado}

[1, 2, 3, 4, 5]
{elijo mi pivot (1)}
[**1**, 2, 3, 4, 5]
{ordeno entre mayores y menores}
[**1**, 2, 3, 4, 5]
{swap entre el pivot y su posición final}
[**1**, 2, 3, 4, 5]
{elijo mi pivot (2)}
[**1**, **2**, 3, 4, 5]
{ordeno entre mayores y menores}
[**1**, **2**, 3, 4, 5]
{swap entre el pivot y su posición final}
[**1**, **2**, 3, 4, 5]
{elijo mi pivot (3)}
[**1**, **2**, **3**, 4, 5]
{ordeno entre mayores y menores}
[**1**, **2**, **3**, 4, 5]
{swap entre el pivot y su posición final}
[**1**, **2**, **3**, 4, 5]
{elijo mi pivot (4)}
[**1**, **2**, **3**, **4**, 5]
{ordeno entre mayores y menores}
[**1**, **2**, **3**, **4**, 5]
{swap entre el pivot y su posición final}
[**1**, **2**, **3**, **4**, 5]
{el arreglo está ordenado}

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]

```
{quick_sort_rec(a, 1, 7)}
[7, 1, 10, 3, 4, 9, 5]
{ordeno entre mayores y menores}
[7, 1, 5, 3, 4, 9, 10]
{swap entre el pivot y su posición final}
[4, 1, 5, 3, 7, 9, 10]
{quick_sort_rec(a,1,4)
 quick_sort_rec(a,6,7)}
[3, 1, 4, 5, 7, 9, 10]
{quick_sort_rec(a,1,2)
 quick_sort_rec(a,4,4)}
[1, 3, 4, 5, 7, 9, 10]
{el arreglo está ordenado}
```

Ejercicio 4

Escribir 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 algoritmo deberá 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: nat
    ppiv:= (lft + rgt) div 2
    i:= lft
    j:= rgt
    while i ≤ j do
        if a[i] ≤ a[ppiv] then
            i:= i+1
        else if a[j] ≥ a[ppiv] then
            j:= j-1
        else if a[i] > a[ppiv] ∧ a[j] < a[ppiv] then
            swap(a,i,j)
            i:= i+1
            j:= j-1
        fi
    od
end proc
```

Ejercicio 5

Escribir un algoritmo que dado un arreglo a: array[1…n] of int y un número natural k ≤ 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.

```
proc k_esimo(in/out a: array[1..n] of T, in k: nat, out elem: T)
    var izq, der, piv: nat
```

```
    izq:= 1
    der:= n
    piv:= partition(a, izq, der)
    while piv ≠ k do
        if piv < k then
            izq:= piv + 1
        else
            der:= piv - 1
        fi
    partition(a, izq, der)
    od
    elem:= a[k]
end proc
```

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. Modificar 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. Modificar el algoritmo quick_sort_rec para adecuarlo al nuevo procedimiento partition.

```
proc quick_sort_rec (in/out a: array[1..n] of T, in lft,rgt: nat)
    var ppiv: nat
    if rgt > lft then
        partition(a,lft,rgt,ppiv_izq, ppiv_der)
        quick_sort_rec(a,lft,ppiv_izq)
        quick_sort_rec(a,ppiv_der,rgt)
    fi
end proc

proc partition (in/out a: array[1..n] of T, in lft, rgt: nat,
                out piv_izq, piv_der: nat)
    var piv_val: T
    var i, j, k: nat

    piv_val:= a[lft]
    i:= lft
    j:= rgt
    k:= lft
    while k+1 ≤ j do
        if a[k+1] = piv_val then
            k:= k + 1
        else if piv_val < a[k+1] then
            swap(a, k+1, j)
            j:= j - 1
        else
            swap(a, k+1, j)
            i:= i + 1
            k:= k + 1
```

```
            fi
        od
        piv_lft:= i
        piv_rgt:= k
    end proc
```