1

a

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

a

1	2	3	4	5	6	7
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	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
1	3	4	5	7	9	10

b

1	2	3	4	5
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
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

c

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

```
// a = [7, 1, 10, 3, 4, 9, 5]
// lft = 1
// rgt = 7
merge_sort_rec(a, 1, 7)
    merge_sort_rec(a, 1, 4)
        merge_sort_rec(a, 1, 2)
            merge_sort_rec(a, 1, 1)
            merge_sort_rec(a, 2, 2)
            merge(a, 1, 1, 2)
//          a = [1, 7, 10, 3, 4, 9, 5]
        merge_sort_rec(a, 3, 4)
            merge_sort_rec(a, 3, 3)
            merge_sort_rec(a, 4, 4)
            merge(a, 3, 3, 4)
//          a = [1, 7, 3, 10, 4, 9, 5]
        merge(a, 1, 2, 4)
//      a = [1, 3, 7, 10, 4, 9, 5]
    merge_sort_rec(a, 5, 7)
        merge_sort_rec(a, 5, 6)
            merge_sort_rec(a, 5, 5)
            merge_sort_rec(a, 6, 6)
            merge(a, 5, 5, 6)
//          a = [1, 3, 7, 10, 4, 9, 5]
```

```
        merge_sort_rec(a, 7, 7)
            merge_sort_rec(a, 7, 7)
            merge_sort_rec(a, 8, 7)
            merge(a, 7, 7, 7)
//          a = [1, 7, 3, 10, 4, 9, 5]
        merge(a, 5, 6, 7)
//      a = [1, 3, 7, 10, 4, 5, 9]
    merge(a, 1, 4, 7)
//  a = [1, 3, 4, 5, 7, 9, 10]
```

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.

- $a = [3, 7, 1, 6, 1, 5, 3, 4]$ e $i = 1$, entonces $[1, 3, 6, 7, 1, 3, 4, 5]$.
- $a = [1, 3, 6, 7, 1, 3, 4, 5]$ e $i = 2$, entonces $[1, 1, 3, 3, 4, 5, 6, 7]$.

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)

    for j := 0 to n - (i + 1) do
        merge_sort_rec(a, 1 + (2*j)*2**i, (2*j + 2)*2**i)
    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..2^n] of int)

    for i := 0 to n - 1 do
        intercalar_cada(a, i)
    od

end proc
```

3)

a

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

a

1	2	3	4	5	6	7
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
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

b

1	2	3	4	5
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

c

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

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]
// lft = 1
// rgt = 7
quick_sort_rec(a, 1, 7)
    partition(a, 1, 7, ppiv)
//  a = [4, 1, 5, 3, 7, 9, 10]
    quick_sort_rec(a, 1, 4)
        partition(a, 1, 4, ppiv)
//      a = [3, 1, 4, 5, 7, 9, 10]
        quick_sort_rec(a, 1, 2)
            partition(a, 1, 2, ppiv)
//          a = [1, 3, 4, 5, 7, 9, 10]
            quick_sort_rec(a, 1, 1)
            quick_sort_rec(a, 3, 2)
        quick_sort_rec(a, 4, 4)
    quick_sort_rec(a, 5, 7)
        partition(a, 5, 7, ppiv)
//      a = [1, 3, 4, 5, 7, 9, 10]
        quick_sort_rec(a, 5, 4)
        quick_sort_rec(a, 6, 7)
            partition(a, 6, 7, ppiv)
//          a = [1, 3, 4, 5, 7, 9, 10]
            quick_sort_rec(a, 6, 5)
            quick_sort_rec(a, 7, 7)
//  a = [1, 3, 4, 5, 7, 9, 10]
```

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.

```
proc partition(in/out a: array[1..n] of int, in lft, rgt: nat, out ppiv: nat)
    var prim, mid, ult: int
    var i,j: nat

    prim := a[left]
    mid := a[(lft + rgt) / 2]
    ult := a[rgt]

    if (prim <= mid && mid <= ult) then
        i := lft
        j := rgt
        ppiv := (lft + rgt) / 2
    else if (mid <= prim && prim <= ult) then
        i := lft + 1
        j := rgt
        ppiv := lft
    else
        i := lft
        j := rgt - 1
        ppiv := rgt
    fi

    while (i <= j) do
        if (a[i] <= a[ppiv] || i = ppiv) then
            i := i + 1
        else if (a[j] >= a[ppiv] || j = 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

    swap(a,ppiv,j)
    ppiv:= j

end proc
```

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.

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

    partition(a, 1, n, ppiv)

    if (ppiv < k) then
        lft := ppiv
        rgt := n
    else
        lft := 1
        rgt := ppiv
    fi

    while (ppiv != k) do
        partition(a, lft, rgt, ppiv)

        if (ppiv < k) then
            lft := ppiv + 1
            rgt := rgt
        else
            lft := lft
            rgt := ppiv - 1
        fi
    od

    r := a[k]

end fun
```

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 $partition$.

```
proc partition(in/out a: array[1..n] of int, in lft, rgt: nat, out ppivL, ppivR:
nat)
    var i, j, L, R: nat
    var ppivS : array[lft..rgt] of int

    i := lft + 1
    j := rgt
    L := 0
    R := 0
    ppivL := lft

    while (i <= j) do
        if (a[i] <= a[ppivL]) then
            if (a[i] = a[ppiVL]) then
                L := L + 1
```

```
            fi
            i := i + 1
        else if (a[j] => a[ppivL]) then
            if (a[j] = a[ppiVL]) then
                R := R + 1
            fi
            j := j - 1
        else if (a[i] > a[ppivL] && a[j] < a[ppivL]) then
            swap(a, i, j)
            i := i + 1
            j := j - 1
        fi
    od

    i := lft + 1

    swap(a, ppivL, j)
    ppivL := j
    ppivR := j

    while (L > 0) {
        if (a[i] = a[ppivL]) {
            ppivL := ppivL - 1
            L := L - 1
            swap(a, ppivL, i)
        }

        i := i + 1
    }

    i := rgt

    while (R > 0) {
        if (a[i] = a[ppivR]) {
            ppivR := ppivR + 1
            R := R - 1
            swap(a, ppivR, i)
        }

        i := i - 1
    }

end proc

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

    if rgt > lft then
        partition(a, lft, rgt, ppivL, ppivR)

        quick_sort_rec(a, lft, ppivL - 1)
        quick_sort_rec(a, ppivR + 1, rgt)
    fi

end proc
```