# Práctico 1.2

1)

1a)

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

2) 🤔

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.

2a)

```
proc intercalar_cada(in/out a : array[1..2^n] of int in i : nat )
      var j, tam : nat
      j ≔ 0
      while (j+1) * 2^j ≤ 2^n do
            intercar(a, j * 2^i + 1, (j+1) * 2^i, j * 2^i)
            j ≔ j + 1
      od
end proc

proc intercalar(in/out a : array[1..n] of T, in lft, mid, rgt : nat)
      var tmp : array[1..n] of T
      var j, k : nat
      for i ≔ lft to mid do
            tmp[i] ≔ a[i]
      od
      j ≔ lft
      k ≔ mid + 1
      for i ≔ lft to rgt do
            if j ≤ mid ∧ (k > rgt ∨ tmp[j] ≤ a[k]) →
                  a[i] ≔ tmp[j]
                  j ≔ j + 1
            else
                  a[i] ≔ a[k]
                  k ≔ k + 1
            fi
      od
end proc
```

2b)

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

3)

```
[7, 1, 10, 3, 4, 9, 5] partition(a, 1, 7, ppiv)
[7, 1, 10, 3, 4, 9, 5]
[7, 1, 5, 3, 4, 9, 10]
[4, 1, 5, 3, 7, 9, 10] partition(a, 1, 4, ppiv)
[4, 1, 5, 3, 7, 9, 10]
[4, 1, 3, 5, 7, 9, 10]
[3, 1, 4, 5, 7, 9, 10] partition(a, 1, 2, ppiv)
[3, 1, 4, 5, 7, 9, 10]
[1, 3, 4, 5, 7, 9, 10] partition(a, 6, 7, ppiv)
[1, 3, 4, 5, 7, 9, 10]
```

4)

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

      if a[lft] ≤ a[rgt] →
            if a[rgt] ≤ a[(lft + rgt) `div` 2] →
                  ppiv ≔ rgt
            else
                  if a[lft] ≤ a[(lft + rgt) `div` 2] →
                        ppiv ≔ (lft + rgt) `div` 2
                  else
                        ppiv ≔ lft
                  fi
            fi
      else
            if a[(lft + rgt) `div` 2] ≤ a[rgt] →
                  ppiv ≔ rgt
            else
                  if a[lft] ≤ a[(lft + rgt) `div` 2] →
                        ppiv ≔ (lft + rgt) `div` 2
                  else
                        ppiv ≔ lft
                  fi
            fi
      fi

      i ≔ lft+1
      j ≔ rgt
      while i ≤ j do
            if a[i] ≤ a[ppiv] →
                  i ≔ i+1
            else if a[j] ≥ a[ppiv] →
                        j ≔ j-1
                  else if a[i] > a[ppiv] ∧ a[j] < a[ppiv] →
                              swap(a, i, j)
                              i ≔ i + 1
                              j ≔ j + 1
                        fi
                  fi
            fi
      od
      swap(a, ppiv, j)
      ppiv ≔ j
end proc
```

5)

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.

```
proc posDeOrdenado(in k : nat in/out a : array[1..n] of int out res : int)
      gen_posDeOrdenado(k, 1, n, a, res)
end proc

proc gen_posDeOrdenado(in k, lft, rgt : nat in/out a : array[1..n] of int out res : int)
      var ppiv : nat
      partition(a, lft, rgt, ppiv)
      if ppiv = k →
            res ≔ a[k]
      else if k < ppiv →
                  gen_posDeOrdenado(k, lft, ppiv - 1, a, res)
            else
                  gen_posDeOrdenado(k, ppiv + 1, a, res)
            fi
      fi
end proc
```

6)

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 quick_sort(in/out a : array[1..n] of T)
      quick_sort_rec(a,1,n)
end proc
```

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

proc partition(in/out a : array[1..n] of T in izq, der : nat out ppiv_izq, ppiv_der : nat)
      var ppiv, i, j, cpivs_izq, cpivs_der: Nat
      ppiv ≔ izq
      i ≔ izq + 1  {- Posicionador desde la izquierda -}
      j ≔ der      {- Posicionador desde la derecha -}
      cpivs_izq ≔ 0  {- Para almacenar la cantidad de elementos iguales a a[ppiv] que hay a voy poniendo a la izquierda -}
      cpivs_der ≔ 0  {- Lo mismo pero para la derecha -}
      while i ≤ j do
            if a[i] < a[ppiv] →
                  i ≔ i+1
            else if a[i] = a[ppiv] →  {- En este caso, swapeo a[i] con la posición más cercana a la izquierda que tiene -}
                        swap(a, izq + cpivs_izq + 1, i)  {- un elemento distinto de a[ppiv]-}
                        i ≔ i+1  {- El nuevo elemento que hay en a[i], ya se sabía que eran menor al pivote -}
                        cpivs_izq ≔ cpivs_izq + 1  {- La cantidad de pivs a la izquierda aumentó en uno-}
```

```
                else if a[j] > a[ppiv] →
                        j ≔ j - 1
                    else if a[j] = a[ppiv] →  {- Lo mismo que cuando a[i] = a[ppiv], solo que moviendo a la derecha -}
                            swap(a, der - cpivs_der, j)
                            j ≔ j-1
                            cpivs_der ≔ cpivs_der + 1
                        else
                            swap(a, i, j)
                        fi
                    fi
                fi
            fi
        od
        for k ≔ izq to izq + cpivs_izq + 1 do  {- Muevo todos los pivs que hay a la izquierda al centro -}
            swap(a, k, i - 1)
            i ≔ i - 1
        od
        for k ≔ der downto der - cpivs_der do  {- Muevo todos los pivs que hay a la derecha al centro -}
            swap(a, j, k)
            j ≔ j + 1
        od
        ppivs_izq ≔ i
        ppivs_der ≔ j
end proc
```