1. Se requiere desarrollar el software de un GPS para mapas de ciudades conformadas por manzanas cuadradas de 100 metros de lado y cuyas calles son todas de doble mano. El GPS dará indicaciones al finalizar cada cuadra, con las siguientes posibilidades: **continúe recto**, **gire a la izquierda**, **gire a la derecha**, **llegó al destino**.

   Para ello se debe definir un TAD Recorrido que contendrá las indicaciones del GPS para ir de una esquina de la ciudad hasta otra. Un recorrido trivial será el que consta de ir desde una esquina hasta ella misma, por lo cual simplemente el GPS dirá **llegó al destino**. Un ejemplo un poco más interesante podría ser: **continúe recto**, **continúe recto**, **gire a la izquierda**, **gire a la derecha**, **llegó al destino**. Un Recorrido entonces consistirá de una secuencia de indicaciones.

   Se necesita también que una vez realizada una acción el recorrido restante se actualice. Para ello, se introducen las operaciones **actualizar continuando recto**, **actualizar girando a la izquierda** y **actualizar girando a la derecha**, que en caso de coincidir con lo indicado por el recorrido se limitará a eliminar del mismo la indicación ejecutada, pero en caso de no coincidir deberá recalcular el recorrido para llegar al destino original.

   Por ejemplo, si en el último ejemplo el vehículo continuó recto, el recorrido resultante será: **continúe recto**, **gire a la izquierda**, **gire a la derecha**, **llegó al destino**. En cambio, si el vehículo en vez de continuar recto, giró a la izquierda, el recorrido resultante podrá ser entonces: **gire a la derecha**, **gire a la derecha**, **gire a la izquierda**, **gire a la izquierda**, **gire a la derecha**, **llegó al destino**.

   También se requieren operaciones que devuelvan un valor booleano para saber si la siguiente indicación de un recorrido es continuar recto, doblar a la derecha, doblar a la izquierda o si ya se llegó al destino.

   (a) Especificá el TAD Recorrido con los constructores correspondientes, las tres operaciones de actualización, las cuatro operaciones de consulta, y una operación **longitud**, que devuelve la longitud medida en metros de un recorrido.

   (b) Implementá el TAD Recorrido utilizando una lista de elementos de tipo **Indicación**, donde el tipo Indicación deberá definirse como un enumerado con 3 valores: Straight, Left, Right.

   (c) Utilizando el tipo de datos abstracto Recorrido, implementá una función que dado un Recorrido, devuelva un natural indicando cuántas veces se debe doblar a la izquierda.

Enunciado:

- El gps da las indicaciones continúe recto, gire a la izquierda, gire a la derecha y llegó a destino
- El Tad Recorrido contendrá las indicaciones del gps
- Recorrido trivial es ir desde una esquina hasta ella misma (llegó a destino)
- Un Recorrido es una secuencia de indicaciones

Operaciones:

- actualizar continuando recto
- actualizar girando a la izquierda
- actualizar girando a la derecha
    - Si se respeta el recorrido se elimina la indicación
    - Si no se respeta deberá recalcular el recorrido
- Operación que diga si la siguiente indicación es continuar recto
- Operación que diga si la siguiente indicación es doblar a la derecha
- Operación que diga si la siguiente indicación es doblar a la izquierda
- Operación que diga si se llego al destino

Constructores:

- Agregar derecho
- Agregar izquierdo
- Agregar recto
- Crear recorrido vacío
- Destroy
- Copy

a)

Spec Recorrido **Where**

Constructors

```
fun recorrido_vacio() ret r: Recorrido
{- Devuelve un recorrido vacío -}

proc agregar_recto(in/out r: Recorrido )
{- Agrega la indicación ir recto -}

proc agregar_derecha(in/out r: Recorrido )
{- Agrega la indicación ir por derecha -}

proc agregar_izquierda(in/out r: Recorrido )
{- Agrega la indicación ir por izquierda-}
```

Destroy

```
proc destroy(in/out r: Recorrido)
{- Libera memoria en caso de ser necesario -}

fun copiar_recorrido (r: Recorrido) ret c: Recorrido
{- Copia el recorrido r en el recorrido c -}
```

Operations

```
proc actualizar_recto(in/out r: Recorrido)
{- Se actualiza el recorrido con la indicación ir recto -}

proc actualizar_derecha(in/out r: Recorrido)
{- Se actualiza el recorrido con la indicación ir por derecha -}

proc actualizar_izquierda(in/out r: Recorrido)
{- Se actualiza el recorrido con la indicación ir por izquierda -}

fun es_recto(r: Recorrido) ret b: bool
{- Nos dice si la siguiente indicación es ir recto -}

fun es_derecha(r: Recorrido) ret b: bool
{- Nos dice si la siguiente indicación es ir por derecha -}

fun es_izquierda(r: Recorrido) ret b: bool
{- Nos dice si la siguiente indicación es ir por izquierda -}

fun es_llegada(r: Recorrido) ret b: bool
{- Nos dice si se llego al destino -}
```

```
fun length_recorrido(r: Recorrido) ret l: nat
{- Nos dice el largo del recorrido -}

end spec
```

b)

```
Implement Recorrido where
type Inidcacion=enumerate
                    Recto
                    Izquierda
                    Derecha
                end enumerate

type Recorrido = List of Indicación

Constructors

fun recorrido_vacio() ret r: Recorrido
     r := empty_list
end fun

proc agregar_recto(in/out r: Recorrido )
     addl(r, Recto)
end proc

proc agregar_derecha(in/out r: Recorrido )
     addl(r,Derecha)
end proc

proc agregar_izquierda(in/out r: Recorrido )
     addl(r,Izquierda)
end proc

proc destroy(in/out r: Recorrido)
     destroy_list(r)
end proc

fun copiar_recorrido (r: Recorrido) ret c: Recorrido
     c := copy_list(r)
end fun

Operations

fun es_recto(r: Recorrido) ret b: bool
     b := head(r) == Recto
end fun
```

```
fun es_derecha(r: Recorrido) ret b: bool
     b := head(r) == Derecha
end fun

fun es_izquierda(r: Recorrido) ret b: bool
     b := head(r) == Izquierda
end fun

fun es_llegada(r: Recorrido) ret b: bool
     b := is_empty_list(r)
end fun

proc actualizar_recto(in/out r: Recorrido)
if(es_recto(r)) then
     tail(r)
else if(es_derecha(r)) then
     addl(r, Derecha)
     addl(r, Derecha)
     addl(r, Recto)
     addl(r, Derecha)
     addl(r, Derecha)
else if(es_izquierda(r)) then
     addl(r, Izquierda)
     addl(r, Izquierda)
     addl(r, Recto)
     addl(r, Izquierda)
     addl(r, Izquierda)
fi
end proc

proc actualizar_derecha(in/out r: Recorrido)
if(es_derecha(r)) then
     tail(r)
else if(es_recto(r)) then
     addl(r, Derecha)
     addl(r, Derecha)
     addl(r, Derecha)
else if(es_izquierda(r)) then
     addl(r, Derecha)
     addl(r, Derecha)
     addl(r, Derecha)
fi
end proc

proc actualizar_izquierda(in/out r: Recorrido)
     if(es_izquierda(r)) then
          tail(r)
     else if(es_recto(r)) then
```

```
            addl(r, Izquierda)
            addl(r, Izquierda)
            addl(r, Izquierda)
      else if(es_derecha(r)) then
            addl(r, Izquierda)
            addl(r, Izquierda)
            addl(r, Izquierda)
      fi
end proc

fun length_recorrido(r: Recorrido) ret l: nat
l := length(r) * 100
end fun
```

c)

```
fun izquierda (r: Recorrido) ret res: nat
      var r_aux: Recorrido
      res := 0
      r_aux := copiar_recorrido(r)
      while not es_llegada(r_aux) do
            if es_izquierda(r_aux)then
                  res++
                  actualizar_izquierda(r_aux)
            fi
            if es_derecha(r_aux) then
                  actualizar_derecha(r_aux)
            fi
            if es_recto(r_aux) then
                  actualizar_recto(r_aux)
            fi
      od
      destroy(r_aux)
end fun
```

2. Utilizando la siguiente implementación del tipo *list* con punteros, se quiere implementar el procedimiento *lshift*. Este procedimiento toma una lista y efectua una rotación circular a izquierda de los elementos. Por ejemplo, dada la lista de números $[0, 0, 1, 1, 0, 1]$, al aplicar el procedimiento *lshift*, el resultado será $[1, 0, 0, 1, 1, 0]$.

```
type node = tuple
              value: elem
              next: pointer to node
            end
type list = pointer to node
```

Implementá el procedimiento *lshift* dando correctamente su encabezado y definición.

```
proc lshift (in/out L: List of T)
```

```
    if not is_empty_list(L) then
        if length(l) > 1 then
            var current_node: pointer to (Node of T)
            var prev_node: pointer to (Node of T)
            current_node := L
            prev_node := null
            while (current_node->next != NULL) do
                prev_node :=current_node
                current_node := current_node->next
            od
            current_node->next := L
            L := current_node
            prev_node->next := null
        fi
    fi
end proc
```

3. Para cada uno de los siguientes algoritmos determinar por separado cada uno de los sig incisos.

   (a) ¿Qué hace? ¿Cuáles son las precondiciones necesarias para que haga eso?

   (b) ¿Cómo lo hace?

   (c) El orden del algoritmo, analizando los distintos casos posibles.

   (d) Proponer nombres más adecuados para los identificadores (de variables, funciones y proced tos).

```
proc p (in/out a: array[1..N] of T)
    var i: nat
    var x: nat
    i:= 2
    do i <= N
        x:= f(a,i)
        swap(a,i,x)
        i:= i+2
    od
end proc
```

```
fun f (a: array[1..N] of T, i: nat) ret x:
    var j: nat
    j:= i+2
    x:= i
    do j <= N
        if a[j] < a[x] then x:= j fi
        j:= j+2
    od
end fun
```

Algoritmo 1:

```
proc p (in/out a: array[1..N] of T)
      var i: nat
      var x: nat
      i := 2
      do i ≤ N
            x := f(a,i)
            swap(a,i,x)
            i := i + 2
      od
end proc
```

¿Qué hace?

Ordena las posiciones pares de una secuencia.

¿Cuáles son las precondiciones necesarias?

Ninguna

¿Cómo lo hace?

Se recorre la secuencia desde el índice 2, luego se obtiene el índice par del mínimo elemento en adelante a través de comparaciones llamando a f, se intercambian los elementos y se repite esto para el siguiente índice par hasta que sea mayor al tamaño de la secuencia.

¿Qué orden tiene?

Debemos contar cuantas veces se compara i ≤ N, como i = 2 en un primer momento y además incrementa en 2 por cada iteración, el bucle se ejecutara N div 2 veces. Por lo tanto el orden de complejidad es o(N/2). Ahora, por cada iteración del bucle, en f tenemos una complejidad o(N). Entonces o(N/2) * o(N) = o(N/2 * N) = o(N^2)

¿Nombre?

ordenar_pares

Algoritmo 2:

```
fun f (a: array[1..N] of T, i: nat) ret x: nat
      var j: nat
      j := i + 2
      x := i
      do j ≤ N
            if a[j] < a[x] then
                  x := j
            fi
            j := j + 2
```

```
    od
end fun
```

¿Qué hace?

La función f toma un índice, un arreglo y devuelve otro índice en donde se encuentra el elemento mínimo.

¿Cuáles son las precondiciones necesarias?

Ninguna

¿Cómo lo hace?

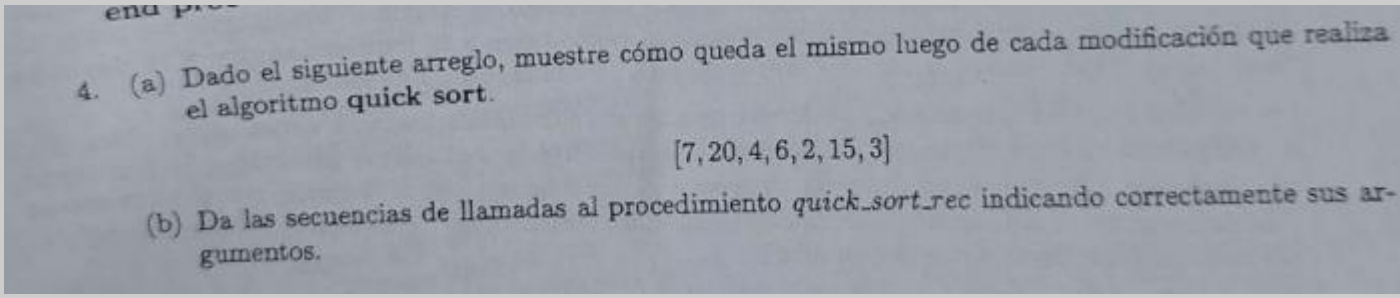
La función f inicialmente guarda el valor del índice de entrada en la variable a retornar, luego de forma cíclica para las posiciones válidas, pares y consiguientes del arreglo si se encuentra un elemento menor se actualiza la variable a retornar con el índice del mismo.

¿Qué orden tiene?

Como j incrementa en pasos de a dos, debemos ver cuántas veces podemos sumarle 2 a j tq $j \leq N$, restamos j en ambos lados $0 \leq N - j$, cuántas veces entra el 2? $(N - j) / 2$. En términos de complejidad lo único que nos interesa es el valor de N, entonces el algoritmo f tiene un orden de o(N) que crece linealmente dependiendo el valor de N

¿Nombre?

minima_pos



4. (a) Dado el siguiente arreglo, muestre cómo queda el mismo luego de cada modificación que realiza el algoritmo **quick sort**.

$$[7, 20, 4, 6, 2, 15, 3]$$

(b) Da las secuencias de llamadas al procedimiento *quick_sort_rec* indicando correctamente sus argumentos.

```
[7,20,4,6,2,15,3]

quick_sort_rec(a,1,7)
     lft = 1, rgt = 7
     partition(a,1,7,ppiv)
     [2,3,4,6,7,15,20]
     ppiv = 5
     1.1 quick_sort_rec(a,1,4)
          lft = 1, rgt = 4
          partition(a,1,4,ppiv)
          ppiv = 1
          [2,3,4,6,7,15,20]
          1.1.1 quick_sort_rec(a,1,0)
               lft = 1, rgt = 0 rgt < lft
```

```
                muere la rama
          1.1.2 quick_sort_rec(a,2,4)
                lft = 2, rgt = 4
                partition(a,2,4,ppiv)
                ppiv = 2
                [2,3,4,6,7,15,20]
                1.1.1.1 quick_sort_rec(a,2,1)
                      lft = 2, rgt = 1 rgt < lft
                      muere la rama
                1.1.1.2 quick_sort_rec(a,3,4)
                      lft = 3, rgt = 4
                      partition(a,3,4,ppiv)
                      ppiv = 3
                      [2,3,4,6,7,15,20]
                      1.1.1.1.1 quick_sort_rec(a,3,2)
                            muere la rama
                      1.1.1.1.2 quick_sort_rec(a,4,4)
                            muere la rama
     1.2 quick_sort_rec(a,6,7)
           lft = 6, rgt = 7
           partition(a,6,7,ppiv)
           ppiv = 6
           [2,3,4,6,7,15,20]
           1.2.1 quick_sort_rec(a,6,5)
                muere la rama
           1.2.2 quick_sort_rec(a,7,7)
                muere la rama

     Finalmente: a = [2,3,4,6,7,15,20]
```

Testeando:

```
proc p (in/out a: array[1..N] of T)
     var i: nat
     var x: nat
     i := 2
     do i ≤ N
          x := f(a,i)
          swap(a,i,x)
          i := i + 2
     od
end proc


fun f (a: array[1..N] of T, i: nat) ret x: nat
     var j: nat
     j := i + 2
```

```
      x := i
      do j ≤ N
            if a[j] < a[x] then
                  x := j
            fi
            j := j + 2
      od
end fun

a[1,5,2,0,0,1,8]
i = 2, N = 7
      x = f(a,2) = 4
      swap(a,2,4)
      i = i + 2
      a[1,0,2,5,0,1,8]
i = 4, N = 7
      x = f(a,4) = 6
      swap(a,4,6)
      i = i + 2
      a[1,0,2,1,0,5,8]
i = 6, N = 7
      x = f(a,6) = 6
      swap(a,6,6)
      i = i + 2
      a[1,0,2,1,0,5,8]
i = 8 > N = 7
      fin

f(a,2) =
a[1,5,2,0,0,1,8]
j = 4, x = 2
      a[j] < a[x] ? si, x = 4
j = 6, x = 4
      a[j] < a[x] ? no, x = 4
j = 8 > N = 7

f(a,2) =
a[1,0,2,5,0,1,8]
j = 6, x = 4
      a[j] < a[x] ? si, x = 6
j = 8 > N = 7

x = f(a,6) =
a[1,0,2,1,0,5,8]
j = 8, x = 6
      No entra al bucle j > N
      x = 6
```

N puede ser par e impar.