Pedro Villar

Notas de Clase
Algoritmos y Estructuras de Datos 2
Teórico Práctico

Primer Cuatrimestre 2024

Información de la materia

Profesores

- **Teórico:** Emanuel Gunther, Luque Franco, Bustos Facundo, Vilela Demetrio y Zigarán Gonzalo.
- **Laboratorio:** Avalos Santiago, Cabral Juan, Canchi Sergio, Gadea Alejandro, Molina Matias, Peralta Gonzalo, Ramos Leandro y Rocchietti Marco.

Régimen de regularidad y promoción

- **Regularidad:** Para regularizar la materia, se tienen que aprobar con nota mayor a 4 ambos parciales.
- **Promoción:** Para promocionar la materia, se tienen que aprobar con nota mayor o igual a 6 ambos parciales y tener un promedio mayor o igual a 7.

No hay recuperatorios para promoción.

Bibliografía

Se utilizará el material de la cátedra que está aca FAMAF Wiki.

Índice de Contenido

1 Introducción al lenguaje de programación Pascal-like

El lenguaje a utilizar está inspirado en el lenguaje imperativo *Pascal*, al cual se le han ido realizando modificaciones de acuerdo a las necesidades did´acticas de la materia. Se utilizará principalmente para la incorporación de conceptos tales como análisis de algoritmos, definición de tipos abstractos de datos, o la comprensión de distintas técnicas de programación.

1.1 Tipos de datos

En este lenguaje, existen varios tipos de datos, incluyendo:

- `int`: para valores enteros,
- `real`: para valores decimales,
- `bool`: para valores de verdad (verdadero o falso),
- `char`: para caracteres.

Además, se pueden crear estructuras de datos más complejas como arreglos para agrupar elementos del mismo tipo.

1.2 Declaración de variables

Las variables se declaran utilizando la palabra clave `var`, seguida de una lista de variables separadas por comas. Cada variable debe tener un tipo asociado. Por ejemplo:

```
{- Variable entera -}
var i : int
i := 10
{- Variable real -}
var x : real
x := 3.14
{- Variable booleana -}
var b : bool
b := true
{- Variable caracter -}
var c : char
c := 'a'
```

1.3 Operadores

Los operadores aritméticos básicos son +, -, * y /. Además, se pueden utilizar los operadores de comparación <, >, <=, >=, == y !=.

```
{- Operadores aritmeticos -}
var a : int
a := 10 + 5
a := 10 - 5
a := 10 * 5
a := 10 / 5
{- Operadores de comparacion -}
var b : bool
b := 10 < 5
b := 10 > 5
b := 10 <= 5
b := 10 >= 5
```

```
13      b := 10 == 5
14      b := 10 != 5
```

Junto con las constantes lógicas `true` y `false`, se pueden utilizar los operadores lógicos `&&`, `||` y `!`.

```
1       {- Operadores logicos -}
2       var b : bool
3       b := true && false
4       b := true || false
5       b := !true
```

No hay operadores definidos para el tipo `char`.

1.4 Tipos de datos estructurados

Un tipo estructurado permite representar colecciones de otros tipos de datos. De manera similar a los tipos básicos, se definen operaciones específicas para acceder a los elementos que conforman al tipo. En el lenguaje solo tenemos definidos de forma nativa a los arreglos.

Para definir un arreglo es necesario detallar el tipo de sus componentes y los tamaños para cada una de sus dimensiones, los cuales deberán ser mayores a cero.

```
1       {- Arreglo de enteros -}
2       var a : array [1..10] of int
3       a[1] := 10
4       {- Arreglo de reales -}
5       var b : array [1..10] of real
6       b[1] := 3.14
7       {- Arreglo de booleanos -}
8       var c : array [1..10] of bool
9       c[1] := true
10      {- Arreglo de caracteres -}
11      var d : array [1..10] of char
12      d[1] := 'a'
```

En este ejemplo, se declaran arreglos de 10 elementos de distintos tipos. Luego, se asigna un valor a la primera posición de cada arreglo.

1.5 Definición de tipos

El lenguaje permite definir tipos de datos personalizados. Esto es útil para abstraer la representación de un concepto y facilitar la comprensión del código. Por ejemplo, se puede definir un tipo `punto` para representar un punto en el plano cartesiano.

```
1   type punto = tuple
2                   x : real
3                   y : real
4                   end tuple
```

En este ejemplo, se define un tipo `punto` que contiene dos campos `x` e `y` de tipo `real`. Luego, se pueden declarar variables de tipo `punto` y asignarles valores.

1.5.1 Tipos enumerados

Un tipo enumerado representa un conjunto finito de valores. Cada valor está definido mediante un identificador único. Para declarar un tipo enumerado se emplean las palabras claves **enumerate** y **end enumerate**. Por ejemplo, definamos un tipo enumerado para los días de la semana.

```
type day = enumerate
            Lunes
            Martes
            Miercoles
            Jueves
            Viernes
            Sabado
            Domingo
            end enumerate
```

Ya una vez definido el tipo enumerado, se pueden declarar variables de este tipo y asignarles valores.

```
var d : day
d := Lunes
```

1.5.2 Sinónimos de tipos

Un sinónimo de tipo es una forma de referirse a un tipo de datos con un nombre diferente. Por ejemplo, se puede definir un sinónimo de tipo `real` para representar la temperatura en grados Celsius.

```
type celsius = real
```

No necesariamente los sinónimos de tipo deben ser de tipos básicos, también se pueden definir sinónimos de tipos estructurados.

```
type matrizdereales = array [1..10] of real
```

Una expresión de este tipo, es útil cuando se utiliza una función donde se espera un valor de uno de los sinónimos de su tipo. En el siguiente ejemplo se declara una variable `mR` del tipo `matrizdereales`, y se opera de manera transparente como si fuese un arreglo tradicional.

```
var mR : matrizdereales
for i := 0 to 9 do
    for j := 0 to 9 do
        mR[i, j] := 0.0
    od
od
```

1.5.3 Tuplas

Una tupla es un tipo de dato estructurado que permite agrupar un número finito de elementos de distintos tipos. Para definir una tupla se emplean las palabras claves **tuple** y **end tuple**. Por ejemplo, definamos una tupla para representar los datos de una persona.

```
type persona = tuple
                nombre : string
                edad : int
                altura : real
                end tuple
```

Y para darle valor a una variable de tipo `persona` se hace de la siguiente manera.

```
var p : persona
p.nombre := "Juan"
p.edad := 20
p.altura := 1.80
```

1.6 Funciones y procedimientos

Las funciones y procedimientos son bloques de código que pueden ser invocados desde otros bloques de código. La diferencia entre ambos radica en que las funciones devuelven un valor, mientras que los procedimientos no. La sintaxis para definir funciones y procedimientos es la siguiente:

```
fun nombre (p1 : T1, p2 : T2, ... , pn : Tn) ret r : T
    {- Cuerpo de la funcion -}
end fun
```

Esta función toma los parametros `p1`, `p2`, ..., `pn` de tipos `T1`, `T2`, ..., `Tn` respectivamente, y devuelve un valor de tipo `T`. Por ejemplo, definamos una función que sume dos números enteros.

```
fun suma (a : int, b : int) ret r : int
    ret := a + b
end fun
```

En el siguiente ejemplo se muestra la implementación de la función `factorial`, que calcula el factorial de un número entero positivo `n`. La variable de retorno fact almacena la productoria de números, y al finalizar la ejecución de la función, se retorna su valor al contexto donde se efectuó la llamada. El comentario simplemente indica la precondición ha satisfacer para garantizar el comportamiento esperado de la función.

```
{- PRE : n >= 0 -}
fun factorial (n : int) ret fact : int
    fact := 1
    for i := 2 to n do
        fact := fact * i
    od
end fun
```

Un procedimiento realiza una computación de acuerdo a un conjunto de parámetros de lectura, para modificar un conjunto de parámetros de escritura. Su comportamiento es determinado solamente por los parámetros que recibe donde cada uno lleva un decorado que indica si es de lectura `in`, de escritura `out`, o ambas `in/out`. Un procedimiento no modifica el estado de los parámetros de lectura, y tampoco consulta el estado de los parámetros de escritura.

En el siguiente ejemplo se implementa el procedimiento initialize, el cual se encarga de inicializar un arreglo de enteros según un valor determinado. Lo interesante a destacar en este ejemplo es la forma en que se manejan los parámetros de lectura y escritura. El parámetro de lectura solo ocurre del lado derecho de la asignación, mientras que el parámetro de escritura solo ocurre del lado izquierdo. Esto significa que al llamar a la función initialize, se pasarán los parámetros como argumentos de lectura (para ser utilizados dentro de la función) y como argumentos de escritura (para almacenar el resultado de la función).

```
proc initialize ( in e : int , out a : array [ 10 ] of int )
    for i := 9 downto 0 do
        a [ i ] := e
    od
end proc
```

1.6.1 Recursión

El lenguaje soporta la recursión, es decir, una función o procedimiento puede llamarse a sí mismo. Por ejemplo, definamos una función que calcule el factorial de un número de manera recursiva. Por ejemplo definamos la función factorial de la siguiente manera.

```
{- PRE : n >= 0 -}
fun factorial (n : int) ret fact : int
    if n >= 2 then
        fact := n * factorial(n - 1)
    else
        fact := 1
    fi
end fun
```

1.6.2 Polimorfismo paramétrico

El lenguaje soporta el polimorfismo paramétrico, es decir, la capacidad de definir funciones y procedimientos que operan sobre un rango de tipos. Por ejemplo, definamos una función que intercambie los valores de dos variables de cualquier tipo.

```
proc swap ( in / out a : array [ n ] of T , in i , j : int )
    var temp : T
    temp := a[i]
    a[i] := a[j]
    a[j] := temp
end fun
```

1.6.3 Polimorfismo *Ad Hoc*

El lenguaje soporta el polimorfismo *Ad Hoc*, es decir, la capacidad de definir funciones y procedimientos que podrían ser implementados de manera genérica pero no para cualquier tipo de datos, sino para ciertos tipos que comparten alguna característica, en consecuencia no es posible utilizar polimorfismo paramétrico. Consideremos las siguientes implementaciones de `belongs` y `selectionSort`. La primera decide si un valor determinado pertenece a un arreglo y la segunda permite ordenar un arreglo de menor a mayor.

```
fun belongs ( e : int , a : array [ n ] of int ) ret b : bool
var i : int
i := 0
b := false
    while ! b && i < n do
        b := a [ i ] == e
        i := i + 1
    od
end fun
```

```
proc selectionSort ( in / out a : array [ n ] of int )
var minPos : int
    for i := 0 to n - 1 do
        minPos := i
        for j := i + 1 to n - 1 do
            if a [ j ] < a [ minPos ] then minPos := j fi
        od
        swap ( a , i , minPos )
    od
end proc
```

En ambas implementaciones los elementos son de tipo entero. Sin embargo podríamos definir las mismas operaciones para otros tipos de datos, como por ejemplo, caracteres. Se tendrían que redefinir las anteriores con los nombres `belongsInt` y `selectionSortInt`, y declarar de manera idéntica las operaciones `belongsChar` y `selectionSortChar` donde solo cambiaríamos int por char en los tipos de los parámetros. Con un trabajo tediosamente repetitivo se podrían dar declaraciones para todos los tipos que tengan definidas las operaciones de comparación; aunque no sería posible para aquellos que no las tengan definidas. El *polimorfismo ad hoc* nos permite escribir de manera genérica una función o un procedimiento donde la tarea que realiza sólo está bien definida para algunos tipos. Además esta tarea puede no ser la misma dependiendo del tipo.
En el lenguaje se definen una serie de clases las cuales pueden ser pensadas como una especie de interfaz que caracteriza algún comportamiento. Un tipo es una instancia de una clase, cuando implementa el comportamiento que la clase describe. El lenguaje sólo incorpora de forma nativa las clases **Eq** y **Ord**, y no existe posibilidad de declarar nuevas clases. La primera representa a los tipos que tienen alguna noción de igualdad, y sus operaciones definidas comprenden al == y !=. La segunda representa a los tipos que poseen alguna relación de orden, y sus operaciones definidas comprenden al <, <=, >=, >.

1.7 Memoria dinámica

El lenguaje permite manipular explícitamente la memoria dinámica mediante un tipo de datos especial, que llamaremos puntero. Supongamos que deseamos definir el tipo correspondiente a las listas. Una lista permite representar una colección ordenada de elementos de algún tipo de datos, cuyo tamaño es variable; lo cual significa que su tamaño crece tanto como sea necesario, de acuerdo a la cantidad de elementos almacenados. Todos los tipos presentados hasta el momento utilizan una cantidad fija de memoria, la cual no puede ser modificada en tiempo de ejecución. Recordemos una vez más que los arreglos implementados en el lenguaje tienen un tamaño fijo al momento de la ejecución. En este aspecto el uso de punteros resulta fundamental, ya que permiten reservar y liberar memoria en la medida que sea necesario durante la ejecución del programa.

1.7.1 Punteros

Un puntero es una variable que almacena la dirección de memoria de otra variable. En el lenguaje, los punteros se representan con el tipo **pointer**, indica el lugar de memoria donde se encuentra almacenado el valor de la variable. La sintaxis para declarar un puntero es la siguiente:

```
var p : pointer of int
```

1.7.2 Ejemplo

```
type node of ( T ) = tuple
                       elem : T ,
                       next : pointer of node of ( T )
                       end tuple

type list of ( T ) = pointer of node of ( T )
```

En el ejemplo anterior se declara una lista enlazada denominada list, la cual se compone de una sucesión de nodos node que se integran por los campos elem de cierto tipo paramétrico T, y next el cual referencia al siguiente nodo en la lista.

1.7.3 Operaciones con punteros

En el lenguaje se definen tres operaciones para manipular punteros. El procedimiento nativo `alloc` toma una variable de tipo puntero, y le asigna la dirección de un nuevo bloque de memoria, cuyo tamaño estará determinado por el tipo de la variable. El operador # permite acceder al bloque de memoria apuntado por el puntero. El procedimiento nativo **free** toma una variable de tipo puntero, y libera el respectivo bloque de memoria referenciado. Retomando el ejemplo de la lista enlazada, los procedimientos `empty` y `addL` son utilizados para construir valores del tipo en cuestión.

```
proc empty ( out l : list of ( T ) )
    l := null
end proc
```

El procedimiento empty construye una lista vacía. La constante null representa un puntero que no referencia un lugar de memoria válido. En el ejemplo, la constante representa una lista que no posee ningín nodo.

```
proc addL ( in e : T , in / out l : list of ( T ) )
    var p : pointer of node of ( T )
    alloc ( p )
    p -> elem := e
    p -> next := l
    l := p
end proc
```

2 Ordenación elemental

El problema de ordenar un conjunto de elementos es uno de los problemas más estudiados en la historia de la computación. Existen numerosos algoritmos para resolver este problema, cada uno con sus propias características y desventajas. En esta sección estudiaremos el algoritmo de ordenación más sencillo (pero no el más rápido), cuyo procedimiento es el siguiente:

1. **Selecciona** el menor de todos, lo intercambia con el elemento que se encuentra en la primera posición.
2. **Selecciona** el menor de los restantes, lo intercambia con el elemento que se encuentra en la segunda posición.
3. **Selecciona** el menor de los restantes, lo intercambia con el elemento que se encuentra en la tercera posición.
4. ... y así sucesivamente hasta que el conjunto esté ordenado.

2.1 Recurso gráfico

Para ilustrar el funcionamiento del algoritmo de ordenación por selección, se ha desarrollado un recurso gráfico que permite visualizar el estado del conjunto en cada iteración. Supongamos que se tiene la lista:

$$[\ 9,3,1,3,5,27\ ]$$

y lo que se busca es aplicar el algoritmo de ordenación por selección.

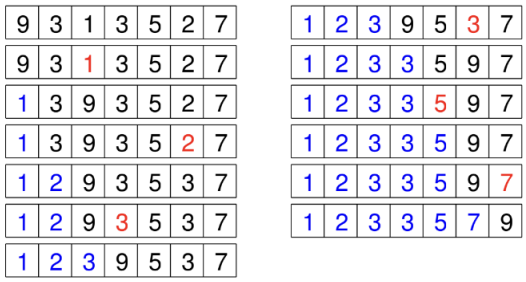

Figure 1: Paso a paso de la ordenación por selección.

2.2 Código

```
proc selection_sort (in/out a: array[1..n] of T)
    var minp: nat
    for i := 1 to n do
        minp := min_pos_from(a, i)
        swap(a, i, minp)
    od
end proc

fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
    minp := i
    for j:= i+1 to n do
        if a[j] < a[minp] then
            minp:= j
        fi
    od
end fun
```

```
17
18 proc swap (in/out a: array[1..n] of T, in i, j: nat)
19     var temp: T
20     temp := a[i]
21     a[i] := a[j]
22     a[j] := temp
23 end proc
```

Para el algoritmo de ordenación por selección, se ha desarrollado un procedimiento `selection_sort` que recibe un arreglo de tamaño n y lo ordena. El procedimiento `selection_sort` utiliza dos funciones auxiliares: `min_pos_from` y `swap`. La función `min_pos_from` recibe un arreglo y una posición, y retorna la posición del menor elemento a partir de la posición dada. La función `swap` recibe un arreglo y dos posiciones, e intercambia los elementos en dichas posiciones. Se pueden marcar las siguientes observaciones:

- `selection_sort` utiliza un ciclo for que recorre el arreglo desde la primera posición hasta la última. En cada iteración, se busca el menor elemento a partir de la posición actual y se intercambia con el elemento en la posición actual,
- se encuentra una llamada a la función `min_pos_from` que recibe el arreglo y la posición actual, y retorna la posición del menor elemento a partir de la posición actual,
- y se recibe tambien una llamada al procedimiento `swap` que recibe el arreglo y las posiciones actual y la posición del menor elemento, e intercambia los elementos en dichas posiciones,
- encontramos una **comparación** entre elementos de un arreglo, y una **asignación** de elementos de un arreglo,
- la operación que mas se repite es la comparación de elementos de un arreglo, y toda operación se repite a lo sumo de manera proporcional a esa,
- como la celda de un arreglo es constante, su costo no depende de cual es la celda o del tamaño del arreglo, por lo que el costo de la operación es constante.

3 Ordenación por inserción

El algoritmo de ordenación por inserción es un algoritmo sencillo que funciona de la siguiente manera:

1. **Selecciona** el segundo elemento de la lista,
2. **verifica** si es menor que el primer elemento, si es así, lo intercambia con el primer elemento,
3. **Selecciona** el tercer elemento de la lista,
4. **verifica** si es menor que el segundo elemento, si es así, lo intercambia con el segundo elemento, y **verifica** si es menor que el primer elemento, si es así, lo intercambia con el primer elemento,
5. y así sucesivamente hasta que el conjunto esté ordenado.

3.1 Recurso gráfico

Para ilustrar el funcionamiento del algoritmo de ordenación por inserción, se ha desarrollado un recurso gráfico que permite visualizar el estado del conjunto en cada iteración. Supongamos que se tiene la lista:

$$[\,9,3,1,3,5,27\,]$$

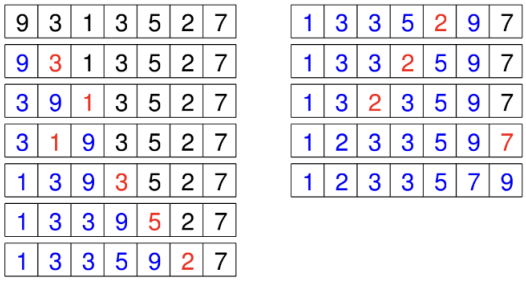

Figure 2: Paso a paso de la ordenación por inserción.

3.2 Código

```
proc insertion_sort (in/out a: array[1..n] of T)
    for i:= 2 to n do
        insert(a,i)
    od
end proc
proc insert (in/out a: array[1..n] of T, in i: nat)
    var j: nat
    j:= i
    do j > 1 && a[j] < a[j - 1] ->
        swap(a,j-1,j)
        j:= j-1
    od
end proc
```

Para el algoritmo de ordenación por inserción, se ha desarrollado un procedimiento `insertion_sort` que recibe un arreglo de tamaño n y lo ordena. El procedimiento `insertion_sort` utiliza un procedimiento auxiliar `insert`. El procedimiento `insert` recibe un arreglo y una posición, y mueve el elemento en la posición dada a la posición correcta. Se puede definir de una forma mas abreviada el algoritmo de ordenación por inserción, como se muestra a continuación:

```
proc insertion_sort (in/out a: array[1..n] of T)
    for i:= 2 to n do
        j:= i
        do j > 1 && a[j] < a[j - 1] ->
            swap(a,j-1,j)
            j:= j-1
        od
    od
end proc
```

4 Algoritmo de ordenación merge_sort

Este algoritmo de ordenación es un ejemplo de algoritmo de ordenación recursivo. La idea es dividir el vector en dos partes iguales, ordenar cada una de las partes y luego combinarlas en un solo vector ordenado.

4.1 Descripción del algoritmo por partes

La idea es definir un procedimiento al que le pasamos en qué parte del arreglo queremos hacer lo fundamental del mergesort: dividirlo en dos, ordenar cada mitad y luego intercalar las dos mitades. Este procedimiento es merge_sort_rec, que toma el arreglo a y las posiciones inicial y final del pedazo de arreglo que vamos a ordenar. El procedimiento principal llama al recursivo con los índices 1 y n (el arreglo completo).

```
proc merge_sort(in/out a: array[1..n] of T)
    merge_sort_rec(a,1,n)
end proc
proc merg_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
...
end proc
```

El procedimiento merge_sort_rec toma el arreglo a, y los índices **lft** y **rgt**, que corresponden con el comienzo y el final del pedazo de arreglo que queremos ordenar. Recordando la idea del algoritmo, el caso más simple es cuando el pedazo de arreglo tiene un solo elemento. En nuestra implementación eso corresponde a que lft sea igual a rgt. En ese caso el procedimiento no debe hacer nada, ya que el pedazo está trivialmente ordenado. En caso que no se dé esa situación, debemos:

1. Dividir el pedazo de arreglo en dos,
2. ordenar cada una de esas mitades utilizando el mismo algoritmo, e
3. intercalar cada mitad ordenada.

Para dividir el pedazo de arreglo, se define una variable **mid** de tipo nat a la que se le asigna el índice correspondiente a la posición del medio.

```
proc merg_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var mid: nat
    if rgt > lft --> mid := (rgt+lft) `div` 2
    ...
end proc
```

Ahora entonces hay que llamar recursivamente al procedmiento dos veces: una para la primera mitad que irá desde la posición **lft** hasta **mid**, y otra para la segunda mitad, que irá desde la posición **mid+1** hasta **rgt**.

```
proc merg_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var mid: nat
    if rgt > lft --> mid := (rgt+lft) `div` 2
    merge_sort_rec(a,lft,mid)
    merge_sort_rec(a,mid+1,rgt)
    ...
end proc
```

y por último, hay que intercalar. Esta tarea la implementaremos con un procedimiento llamado merge, que se define mas adelante.

```
proc merg_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var mid: nat
    if rgt > lft --> mid := (rgt+lft) `div` 2
    merge_sort_rec(a,lft,mid)
    merge_sort_rec(a,mid+1,rgt)
    merge(a,lft,mid,rgt)
end proc
```

Ahora para implementar el procedimiento de **intercalación**, se necesita un arreglo auxiliar, en donde se van a guardar los valores de la primera mitad a intercalar. Se define entonces una variable de tipo array, dos variables en las que luego almacenaremos índices `j` y `k`, y se copia la primera mitad del arreglo en el arreglo auxiliar:

```
proc merge(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
    ...
end proc
```

Los índices `j` y `k` indicarán respectivamente el elemento de la primera mitad que estoy analizando para insertar en el pedazo de arreglo que quedará ordenado, y el índice de la segunda mitad que estoy analizando. Inicialmente observo el primero de cada mitad, es decir **lft** y **mid+1**.

```
proc merge(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
    ...
end proc
```

Ahora hay que rellenar el pedazo completo de arreglo que contendrá las dos mitades intercaladas ordenadamente. Lo recorro con un for desde lft hasta rgt. Y se puede ver que en cada paso si el elemento que estoy observando de la primera mitad es menor o igual que el de la segunda mitad, de acuerdo a esa comparación sabré qué elemento va a ubicarse en el arreglo ordenado.

```
proc merge(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])
            then a[i] := tmp[j]
                 j:= j+1
            else a[i] := a[k]
                 k := k+1
        fi
    od
end proc
```

En la guarda del if hay que considerar también el caso en que ya haya agotado todos los elementos de la segunda mitad, lo que sucederá cuando `k > rgt`, y entonces en ese caso también completo con los elementos de la primera mitad (es decir los que están en el arreglo auxiliar).

El código completo del algoritmo de ordenación merge_sort es el siguiente:

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

proc merge_sort_rec(in/out a: array[1..n] of T, in lft,rgt: nat)
    var mid: nat
    if rgt > lft --> mid := (rgt+lft) `div` 2
        merge_sort_rec(a,lft,mid)
        merge_sort_rec(a,mid+1,rgt)
        merge(a,lft,mid,rgt)
    fi
end proc

proc merge(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])
            then a[i] := tmp[j]
                 j:= j+1
            else a[i] := a[k]
                 k := k+1
        fi
    od
end proc
```

5 Algoritmo de ordenación `quick_sort`

Este algoritmo consiste en separar dos mitades: por un lado los que irían al principio y por otro los que irían al final. Luego se ordenan las dos mitades de forma recursiva.

5.1 Descripción del algoritmo por partes

De manera similar al algoritmo de ordenación `merge_sort`, se define un procedimiento recursivo que tomará el arreglo de elementos y dos índices correspondientes al pedazo de arreglo que se ordenará. El algoritmo principal llama a este procedimiento con los índices `1` y `n`, correspondiendo con la ordenación del arreglo completo.

```
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 lft,rgt: nat)
...
end proc
```

Este procedimiento recursivo tiene su caso más simple cuando **lft** y **rgt** son iguales, lo que significa que estoy ordenando un arreglo de un solo elemento. En el caso interesante, al procedimiento se lo llama partition que será el encargado de acomodar los elementos del pedazo de arreglo utilizando el elemento de más a la izquierda como **pivot**.

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

El procedimiento `partition` modifica el arreglo desde `lft` hasta `rgt` dejando al comienzo todos los elementos que son menores o iguales al que se encontraba originalmente en la posición `lft`, y al final a todos los que son mayores o iguales. También modifica la variable `ppiv` asignándole el índice correspondiente al lugar donde queda ubicado definitivamente el elemento que se usó como **pivot**. Luego lo único que queda por hacer es llamar recursivamente al procedimiento, una vez para los elementos que quedaron acomodados a la izquierda del pivot, y otra vez para los elementos que quedaron acomodados a la derecha.

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

Queda por ver el procedimiento partition. Toma el arreglo, los dos índices que indican qué fragmento estamos ordenando, y una variable de solo escritura, en la cual indicaremos el índice en donde queda el pivot una vez que finalice el procedimiento.

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

Lo que hay que hacer en este procedimiento es ir mirando con un índice los elementos que están a la izquierda y con otro los que están a la derecha. El índice `i` indicará el elemento que estoy mirando desde la izquierda, y

respectivamente j indicará el de la derecha. El elemento tomado como **pivot** será el que está más a la izquierda.

```
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
    ...
end proc
```

Luego hay que ir viendo si el elemento indicado con el índice i y el indicado con j están bien ubicados, es decir, si a[i] es menor o igual al **pivot**, y si a[j] es mayor o igual. En caso que sea así, "avanzo" el índice. Este avance corresponde a sumar uno para i, y a restar uno para j. En caso que el elemento de la izquierda esté mal ubicado, debemos encontrar un elemento de la derecha que también esté mal ubicado, y los intercambiamos.

```
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)
        fi
    od
...
end proc
```

Esto se repite hasta que los índices i y j se hayan cruzado. En ese momento se termina el ciclo y solo queda ubicar correctamente al elemento **pivot**, y asignar la variable ppiv para que indique la posición final en donde queda el mismo.

```
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)
                 fi
    od
    swap(a,ppiv,j)
    ppiv:= j
end proc
```

El código completo del algoritmo de ordenación quick_sort es el siguiente:

```
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 lft,rgt: nat)
    var ppiv: nat
    if rgt > lft -->
        partition(a,lft,rgt,ppiv)
        quick_sort_rec(a,lft,ppv-1)
        quick_sort_rec(a,ppiv+1,rgt)
    fi
end proc

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)
                  fi
    od
    swap(a,ppiv,j)
    ppiv:= j
end proc
```