```
procedurePonerUnaDeCada(){
 /* PROPOSITO: Poner una bolita de cada color en la celda actual.PRECONDICION: Ninguna.*/

       Poner(Azul)
       Poner(Negro)
       Poner(Rojo)
       Poner(Verde)
 }
```

```
procedurePoner_DeColor_(cantidadAPoner, colorAPoner) {
 /* PROPOSITO: Poner tantas bolitas como se indica del color dado de la celda
 actual.PRECONDICION: Ninguna. PARAMETROS: *cantidadAPoner* : Numero. Indica la
 cantidad de bolitas a poner.
              *colorAPoner* : Color. Indica el color de las bolitas.*/

       repeat (cantidadAPoner) {
             Poner(colorAPoner)
       }
 }
```

```
 procedure Mover_VecesAl_(cantidadAMover, direcciónAMover) {
     /*
         PROPÓSITO: Mueve el cabezal una cantidad de veces **cantidadAMover** de celdas en
 la dirección
                    **direcciónAMover**.
         PARÁMETROS:
             * cantidadAMover  : Número     - La cantidad de celdas a mover el cabezal
             * direcciónAMover : Direcciòn  - La dirección en la cual mover el cabezal
         PRECONDICIÓN: Debe haber al menos **cantidadAMover** celdas en la dirección
 **direcciónAMover**.
     */

     repeat(cantidadAMover) {
         Mover(direcciónAMover)
     }
 }
```

```
 procedure Sacar_DeColor_(cantidadASacar, colorASacar) {
 /* PROPOSITO: sacar *cantidadASacar* bolitas de color *colorASacar*. PRECONDICION:
 Debe haber al menos *cantidadASacar* bolitas de color *colorASacar*.PARAMETROS:
 *cantidadASacar*: Numero. Indica el número de bolitas a sacar.
              *colorASacar*: Color. Indica el color de bolitas a sacar.*/

       repeat(cantidadASacar) {
             Sacar(colorASacar)
       }
 }
```

```
 procedure SacarTodasLasDeColor_(colorASacar) {
     /*
         PROPÓSITO: Saca todas las bolitas del color **colorASacar** de la celda actual.
         PARÁMETROS:
             * colorASacar : Color - El color de las bolitas a sacar.
         PRECONDICIÓN: Ninguna.
     */

     repeat(nroBolitas(colorASacar)) {
         Sacar(colorASacar)
     }
 }
```

```
procedure Poner_Si_ (color, condicion){
/* PROPOSITO: poner en la celda actual una bolita de color *color* si el valor de
lacondición *condicion* es verdadero. Si el valor es falso, no pone bolitas.
PRECONDICION: Ninguna?
PARAMETROS: *color*: Color. Indica el color de las bolitas a poner.
            *condicion*: Condición. Valor booleano que puede ser verdadero o falso.*/

      if (condicion){
            Poner(color)
      }
}
```

```
procedure Sacar_Si_ (color, condicion){
/* PROPOSITO: saca una bolita de color *color* de la celda actual si el valor de
lacondición *condicion* es verdadero. Si el valor es falso, no saca bolitas.
PRECONDICION: Ninguna?
PARAMETROS: *color*: Color. Indica el color de las bolitas a poner.
            *condicion*: Condición. Valor booleano que puede ser verdadero o falso. */

      if (condicion){
            Sacar(color)
      }
}
```

```
procedure Mover_Si_ (direccion, condicion) {
/* PROPOSITO: mueve la celda en la dirección *direccion* si el valor de la condición
*condicion* es verdadero. Si el valor es falso, no mueve el cabezal.PRECONDICION: Ninguna?
PARAMETROS: *direccion*: Dirección. Indica la dirección en la que se moverá el cabezal.
            *condicion*: Valor booleano que puede ser verdadero o falso.*/

      if (condicion){
            Mover(direccion)
      }
}
```

```
procedureIrAPrimeraCeldaEnUnRecorridoAl_Y_(dirPrincipal,dirSecundaria){
/* PROPOSITO: va a la esquina "dirPrincipal" y "dirSecundaria".
PRECONDICION: "dirPrincipal" y "dirSecundaria" no pueden ser opuestas o
iguales.PARAMETROS: **dirPrincipal: dirección: es una dirección.
            **dirSecundaria: dirección: es una dirección.*/

    IrAlBorde_(dirPrincipal)
    IrAlBorde_(dirSecundaria)
}
```

```
procedureIrAlBorde_(dirección){
/* PROPOSITO: va al borde "dirección" del tablero.PRECONDICION:
Ninguna.
PARAMETROS: **dirección: dirección: dirección a la cual irá al borde.*/

    while ( puedeMover(dirección) ) {
        Mover(dirección)
    }
}
```

```
functionhaySiguienteCeldaEnUnRecorridoAl_Y_(dirPrincipal,dirSecundaria){
/* PROPOSITO: indica si hay una celda al "dirPrincipal" o al
"dirSecundaria".PRECONDICION: Ninguna. PARAMETROS: **dirPrincipal: dirección:
es una dirección.
            **dirSecundaria: dirección: es una dirección.*/

    return( puedeMover(dirPrincipal) || puedeMover(dirSecundaria) )
}
```

```
procedureIrASiguienteCeldaEnUnRecorridoAl_Y_(dirPrincipal,dirSecundaria){
/* PROPOSITO: va a la siguiente celda del recorrido hacia "dirPrincipal" o bien
hacia"dirSecundaria". PRECONDICION: "dirPrincipal" y "dirSecundaria" no pueden ser
iguales u opuestas. PARAMETROS:**dirPrincipal: dirección: dirección principal en la
que se mueve el cabezal.
        **dirSecundaria: dirección: dirección secundaria en la que se mueve el cabezal.*/

    if (puedeMover(dirSecundaria)) {
        Mover(dirSecundaria)
    }
    else { IrAlBorde_(opuesto(dirSecundaria))
    Mover(dirPrincipal)
    }
}
```

```
functionesCeldaVacía(){
/*PROPOSITO: Indica si en la celda actual no hay ninguna bolita.PRECONDICION: Ninguna
TIPO: booleano.*/

    return(
        not hayBolitas(Azul) && not
        hayBolitas(Negro) && not
        hayBolitas(Rojo) && not
        hayBolitas(Verde)
          )
}
```

```
functiontieneUnaDeCada(){
/* PROPOSITO: Indica si la celda tiene al menos una bolita de cada
color.PRECONDICION: Ninguna. TIPO: booleano.*/

    return(
            hayBolitas(Azul) &&
            hayBolitas(Negro) &&
            hayBolitas(Rojo) &&
            hayBolitas(Verde)
          )
}
```

```
functionesCeldaConBolitas(){
/* PROPOSITO: Indica si la celda actual tiene al menos una bolita, de cualquier color.
PRECONDICION:Ninguna.*/

    return( not esCeldaVacía() )
}
```

```
functionhayBolitas_Al_(color, dirección){
/* PROPOSITO: Indica si hay una celda lindante en la dirección "dirección" y la
mismatiene bolitas de color "color". PRECONDICION: Ninguna.
PARAMETROS: **color: color: es el color de las bolitas.
**dirección: dirección: dirección de la celda lindante donde se desea ver si
hay bolitas.TIPO: booleano OBS: Si no hay una celda lindante hacia
"dirección", describe Falso.*/

    return( puedeMover(dirección) && tieneBolitas_Al_(color, dirección) )
}
```

```
functiontieneBolitas_Al_(color,dirección){
/* PROPOSITO: Indica si hay bolitas de color "color" en la dirección
"dirección".PRECONDICION: Ninguna. PARAMETROS: **color: color: es el color de
las bolitas.
**dirección: dirección: dirección de la celda lindante donde se desea ver si
hay bolitas.TIPO: booleano OBS: Si no hay una celda lindante hacia
"dirección", hace BOOM.*/

    Mover(dirección)
    return(hayBolitas(color) )
}
```

```
functionmaximoEntre_Y_(num1, num2){
/*PROPOSITO: describe el valor más grande según los valores
dados.PRECONDICION: Ninguna PARAMETROS: *num1*: Número. Primer número a
comparar.
            *num2*: Número. Segundo número a comparar.
TIPO: Numero.*/

    return(
            choose
                num1 when (num1 > num2)
                num2 otherwise
    )
}
```

```
functionminimoEntre_Y_(num1, num2){
/*PROPOSITO: describe el valor más chico según los valores dados.PRECONDICION: Ninguna
PARAMETROS: *num1*: Número. Primer número a comparar.
            *num2*: Número. Segundo número a comparar.
TIPO: Numero./*

    return(
            choose
                num1 when (num1 < num2)
                num2 otherwise
           )
}
```

```
functiondistanciaAlBorde_(direccion) {
/* PROPOSITO: Describe la cantidad de celdas que hay entre la celda
actual y el bordeindicado. PRECONDICION: Ninguna.
PARAMETRO: *direccion*. Dirección. Indica el borde.
TIPO: Numero.*/

    contadorDeCeldas := 0
    while (puedeMover(direccion)){
        Mover(direccion)
        contadorDeCeldas := contadorDeCeldas +1
    }
    return(contadorDeCeldas)
}
```

```
functioncoordenadaX(){
/* PROPOSITO: describe la coordenada de la fila actual.PRECONDICION: Ninguna.
TIPO: Numero.
OBS: La coordenada es la cantidad de celdas desde el borde oeste hasta la posicion actualdel
cabezal.*/

    return(distanciaAlBorde_(Oeste))
}

functioncoordenadaY(){
/* PROPOSITO: describe la coordenada de la columna actual.PRECONDICION: Ninguna.
TIPO: Numero.
OBS: La coordenada es la cantidad de celdas desde el borde sur hasta la posicion actualdel
cabezal.*/

    return(distanciaAlBorde_(Sur))
}
```

```
function nroFilas(){
/*PROPOSITO: cuenta la cantidad de filas del tablero.PRECONDICION: Ninguna
TIPO: Número*/
    IrAlBorde(Sur)
    contadorDeFila := 1
    while (puedeMover(Norte)){
        Mover(Norte)
        contadorDeFila := contadorDeFila + 1
    }
    return(contadorDeFila)
}

function nroColumnas(){
/*PROPOSITO: cuenta la cantidad de columnas del tablero.PRECONDICION: Ninguna
TIPO: Número*/ IrAlBorde(Oeste)
    contadorDeColumnas := 1 while
    (puedeMover(Este)){
        Mover(Este)
        contadorDeColumnas := contadorDeColumnas + 1
    }
    return(contadorDeColumnas)
}
```

```
function nroVacías(){
/*PROPOSITO: describe la cantidad de celdas vacías en el tablero.PRECONDICION: Ninguna.
TIPO: Número.*/ IrAPrimeraCeldaEnUnRecorridoAl_Y_(Sur, Oeste)
    cantidadVacias :=unoSi_CeroSino(esCeldaVacía())
    while(haySiguienteCeldaEnUnRecorridoAl_Y_(Este,
        Norte)){IrASiguienteCeldaEnUnRecorridoAl_Y_(Este, Norte) cantidadVacias :=
        cantidadVacias +unoSi_CeroSino(esCeldaVacía())
    }
    return(cantidadVacias)
}
```

```
functioncantidadDeCeldasConBolitasDeColor_(color){
/*PROPOSITO: describe la cantidad de celdas con al menos una bolita de color
*color*.PRECONDICION: Ninguna. TIPO: Número.*/
    IrAPrimeraCeldaEnUnRecorridoAl_Y_(Sur, Oeste) cantidadCeldasConColor :=
    unoSi_CeroSino(hayBolitas(color))while
    (haySiguienteCeldaEnUnRecorridoAl_Y_(Este, Norte)){
        IrASiguienteCeldaEnUnRecorridoAl_Y_(Este, Norte) cantidadCeldasConColor :=
        cantidadCeldasConColor +
unoSi_CeroSino(hayBolitas(color))
    }
    return(cantidadCeldasConColor)
}
```

```
functionnroBolitasTotalDeColor_(color){
/*PROPOSITO: describe la cantidad de bolitas de color *color* que hay en
total en todo eltablero. PRECONDICION: Ninguna.
TIPO: Número.*/ IrAPrimeraCeldaEnUnRecorridoAl_Y_(Sur, Oeste)
    cantidadDeColor := nroBolitas(color)
    while (haySiguienteCeldaEnUnRecorridoAl_Y_(Este, Norte)){
        IrASiguienteCeldaEnUnRecorridoAl_Y_(Este, Norte) cantidadDeColor :=
        cantidadDeColor + nroBolitas(color)
    }
    return(cantidadDeColor)
}
```

```
functionunoSi_CeroSino(expresionBooleana) {
/*PROPOSITO: Describe un 1 si se verifica *expresionBooleana*.PRECONDICION: Ninguna
PARAMETRO: *expresionBooleana*: Booleano. La condición a cumplir.
TIPO: Booleano. */
    return (
      choose
            1 when (expresionBooleana)
            0 otherwise
    )
}
```

```
 function longitudDe_(unaLista){
  /*
  Propósito: Describe la cantidad de elementos de la lista **unaLista**.
  Parámetros:
  * unaLista: Lista de elementos - La lista a saber su cantidad de elementos.
  Precondición: Ninguna.
  Tipo: Número.
  */

  restoLista := unaLista
  cantidad := 0
  while(not esVacía(restoLista)){
  cantidad := cantidad + 1
  restoLista := resto(restoLista)
  }

  return(cantidad)
}
```

```
function reversoDe_(unaLista){
 /*
 Propósito: Describe el reverso de la lista dada, es decir, la lista dada vuelta.
 Parámetros:
 * unaLista: Lista de elementos - La lista a describir el reverso.
 Precondición: Ninguna.
 Tipo: Lista de elementos.
 */

 listaResto := unaLista
 listaReversa := [primero(listaResto)]
```

```
 listaResto := resto(listaResto)
 while(not esVacía(listaResto)){
 listaReversa := [primero(listaResto)] ++ listaReversa
 listaResto := resto(listaResto)
 }

 return(listaReversa)
 }
```

```
function sumatoriaDe_(unaListaDeNúmeros){
 /*
 Propósito: Describe la suma de todos los elementos de la lista **unaListaDeNúmeros**.
 Parámetros:
 * unaListaDeNúmeros: Lista de números - La lista de números a hacerle la sumatoria.
 Precondición: Ninguna.
 Tipo: Número.
 */

 sumatoria := 0
 restoLista := unaListaDeNúmeros
 while(not esVacía(restoLista)){
 sumatoria := sumatoria + primero(restoLista)
 restoLista := resto(restoLista)
 }

 return(sumatoria)
}
```

```
function productoriaDe_(unaListaDeNúmeros){
 /*
 Propósito: Describe el producto de todos los elementos de la lista
**unaListaDeNúmeros**.  Parámetros:
 * unaListaDeNúmeros: Lista de números - La lista de números a hacerle la productoria.
 Precondición: Ninguna.
 Tipo: Número.
 */

 productoria := 1
 restoLista := unaListaDeNúmeros
 while(not esVacía(restoLista)){
 productoria := productoria * primero(restoLista)
 restoLista := resto(restoLista)
 }

 return(productoria)
}
```

```
function direccionesOpuestasDe_(unaListaDeDirecciones){
 /*
 Propósito: Describe una lista de direcciones donde cada elemento es el opuesto
al de la posición original.  Parámetros:
 * unaListaDeDirecciones: Lista de direcciones - La lista de direcciones a
hacerle lista de opuestos.  Precondición: Ninguna.
 Tipo: Lista de direcciones.
 */

 listaDireccionesOpuestas := []
 restoLista := unaListaDeDirecciones
 while(not esVacía(restoLista)){
 listaDireccionesOpuestas := listaDireccionesOpuestas ++
[opuesto(primero(restoLista))]  restoLista := resto(restoLista)
 }

 return(listaDireccionesOpuestas)
}
```

```
function siguientesDe_(unaListaDeColores){
 /*
 Propósito: Describe una lista de colores donde cada elemento es el siguiente
del original de la lista  **unaListaDeColores**.
 Parámetros:
 * unaListaDeColores: Lista de colores - La lista de colores a hacerle una
lista de los siguientes.  Precondición: Ninguna.
 Tipo: Lista de colores.
 */

 listaColoresSiguientes := []
 restoLista := unaListaDeColores
```

```
 while(not esVacía(restoLista)){
 listaColoresSiguientes := listaColoresSiguientes ++ [siguiente(primero(restoLista))]
 restoLista := resto(restoLista)
 }

 return(listaColoresSiguientes)
}
```

```
function elementosDe_multiplicadosPor_(unaListaDeNúmeros, númeroMultiplicador){
 /*
 Propósito: Describe una lista de números donde cada número de la lista
**unaListaDeNúmeros** fue multiplicado por  **númeroMultiplicador**.
 Parámetros:
 * unaListaDeNúmeros : Lista de números - La lista de números a multiplicar.
 * númeroMultiplicador: Número - El número para multiplicar los números
de la lista.  Precondición: Ninguna.
 Tipo: Lista de números.
 */

 listaNúmerosMultiplicados := []
 restoLista := unaListaDeNúmeros
 while(not esVacía(restoLista)){
 listaNúmerosMultiplicados := listaNúmerosMultiplicados ++
[((primero(restoLista))*númeroMultiplicador)]  restoLista := resto(restoLista)
 }

 return(listaNúmerosMultiplicados)
}
```

```
function númerosParesDe_(unaListaDeNúmeros){
 /*
 Propósito: Describe una lista de números de los números pares que aparezcan en la lista
**unaListaDeNúmeros**.  Parámetros:
 * unaListaDeNúmeros: Lista de números - La lista de números a saber los pares.
 Precondición: Ninguna.
 Tipo: Lista de números.
 */

 listaNúmerosPares := []
 restoLista := unaListaDeNúmeros
 while(not esVacía(restoLista)){
 listaNúmerosPares := listaNúmerosPares ++ listaSiEsPar_(primero(restoLista))
 restoLista := resto(restoLista)
 }

 return(listaNúmerosPares)
}
```

```
function listaSiEsPar_(número){
 /*
 Propósito: Describe una lista con el elemento **número** si es par, si no lo
es describe una lista vacía.  Parámetros:
 * número: Número - El número a saber si es par.
 Precondición: Ninguna.
 Tipo: Lista de números.
 */

 return(
 choose
 [número] when ((número mod 2) == 0)
 [] otherwise
 )
}
function laLista_SinElElemento_(unaLista, elementoAQuitar){
 /*
 Propósito: Describe la lista que resulta de quitar todas las apariciones del elemento
**elementoAQuitar** de la lista  **unaLista**.
 Parámetros:
 * unaLista : Lista de elementos - La lista de elementos.
 * elementoAQuitar: Elemento - El elemento a quitar de la lista.
 Precondición: Ninguna.
 Tipo: Lista de elementos.
 */

 listaDeElementosFinal := []
 restoLista := unaLista
```

```
 while(not esVacía(restoLista)){
 listaDeElementosFinal := listaDeElementosFinal ++
listaSiElElemento_NoEs_(primero(restoLista), elementoAQuitar)  restoLista :=
resto(restoLista)
 }

 return(listaDeElementosFinal)
}
```

```
function listaSiElElemento_NoEs_(elementoDeLista, elementoDeComparación){
 /*
 Propósito: Describe una lista con el elemento **elementoDeLista** si este no es igual a
**elementoDeComparación**, si es igual describe una lista vacía.
 Parámetros:
 * elementoDeLista : Elemento - El elemento de la lista.
 * elementoDeComparación: Elemento - El elemento a comparar.
 Precondición: Ninguna.
 Tipo: Lista de elementos.
 */

 return(
 choose
 [elementoDeLista] when (elementoDeLista /= elementoDeComparación)
 [] otherwise
 )
}
```

```
function losMayoresA_De_(umbral, unaLista){
 /*
 Propósito: Describe una lista con los elementos de la lista **unaLista** que
sean mayores a **umbral**.  Parámetros:
 * umbral : Elemento - El elemento de umbral.
 * unaLista: Lista de elementos - La lista de elementos.
 Precondición: Ninguna.
 Tipo: Lista de elementos.
 */

 listaElementosFinal := []
 restoLista := unaLista
 while(not esVacía(restoLista)){
 listaElementosFinal := listaElementosFinal ++
listaSi_EsMayorA_(primero(restoLista), umbral)  restoLista :=
resto(restoLista)
 }

 return(listaElementosFinal)
}
```

```
function listaSi_EsMayorA_(elemento, umbral){
 /*
 Propósito: Describe una lista con el elemento **elemento** si este es mayor a **umbral**,
si no lo es describe una lista  vacía.
 Parámetros:
 * elemento : Elemento - El elemento a comparar.
 * umbral : Elemento - El umbral.
 Precondición: Ninguna.
 Tipo: Lista de elementos.
 */

 return(
 choose
 [elemento] when (elemento > umbral)
 [] otherwise
 )
}
```

```
function contiene_A_(unaLista, unElemento){
 /*
 Propósito: Indica si la lista **unaLista** contiene al elemento **unElemento**.
 Parámetros:
 * unaLista : Lista de elementos - La lista a saber si está el elemento.
 * unElemento: Elemento - El elemento a saber si está en la lista.
 Precondición: Ninguna.
 Tipo: Booleano.
 */

 restoLista := unaLista
 while(not esVacía(restoLista) && primero(restoLista) /= unElemento){
 restoLista := resto(restoLista)
 }
 return(not esVacía(restoLista) && primero(restoLista) == unElemento)
}
```

```
function algunoMayorQué_En_(unElemento, unaLista){
 /*
 Propósito: Indica si la lista **unaLista** contiene algún elemento que sea
mayor a **unElemento**.  Parámetros:
 * unElemento: Elemento - El elemento a saber si hay uno mayor.
 * unaLista : Lista de elementos - La lista de elementos.
 Precondición: Ninguna.
 Tipo: Booleano.
 */

 return(not esVacía(losMayoresA_De_(unElemento, unaLista)))
}
```

```
function hayAlgunoDe_Entre_Y_(unaListaDeNúmeros, nroDesde, nroHasta){
 /*
 Propósito: Indica si la lista **unaListaDeNúmeros** contiene algún elemento que sea mayor
a **nroDesde** y menor  que **nroHasta**.
 Parámetros:
 * unaListaDeNúmeros: Lista de números - La lista de elementos a analizar.
 * nroDesde : Número - El número desde.
 * nroHasta : Número - El número hasta.
 Precondición: Ninguna.
 Tipo: Booleano.
 */

 restoLista := unaListaDeNúmeros
 while(not esVacía(restoLista) && ((primero(restoLista) < nroDesde) ||
(primero(restoLista) > nroHasta))){  restoLista := resto(restoLista)
 }

 return(not esVacía(restoLista) && (primero(restoLista) > nroDesde) &&
(primero(restoLista) < nroHasta)) }
```

```
function hayAlgúnElementoImparDe_(unaListaDeNúmeros){
 /*
 Propósito: Indica si la lista **unaListaDeNúmeros** contiene algún
elemento que sea impar.  Parámetros:
 * unaListaDeNúmeros: Lista de números - La lista a ver si hay algún impar.
 Precondición: Ninguna.
 Tipo: Booleano.
 */

 restoLista := unaListaDeNúmeros
 while(not esVacía(restoLista) && ((primero(restoLista) mod 2) == 0)){
 restoLista := resto(restoLista)
 }

 return(not esVacía(restoLista) && ((primero(restoLista) mod 2) /= 0))
}
```

```
function sinDuplicados_(unaLista){
 /*
 Propósito: Describe una lista de elementos que contenga los elementos de la lista
**unaLista** sin repetir.  Parámetros:
 * unaLista: Lista de elementos - La lista a sacarle los duplicados que haya.
 Precondición: Ninguna.
 Tipo: Lista de elementos.
 */

 listaSinDuplicados := []
 restoLista := unaLista
 while(not esVacía(restoLista)){
 listaSinDuplicados := listaSinDuplicados ++
listaDeElemento_SiNoEstáEnLista_(primero(restoLista),  listaSinDuplicados)
 restoLista := resto(restoLista)
 }

 return(listaSinDuplicados)
}
```

```
function listaDeElemento_SiNoEstáEnLista_(unElemento, unaLista){
 /*
 Propósito: Describe una lista de elementos con el elemento **unElemento** si no está en
**unaLista**, si ya está en la  lista describe una lista vacía.
 Parámetros: * unElemento: Elemento - El elemento a ver si está en la lista.
 * unaLista : Lista de elementos - La lista a ver si está el elemento.
 Precondición: Ninguna.
 Tipo: Lista de elementos.
 */

 return(
 choose
 [unElemento] when (not contiene_A_(unaLista, unElemento))
 [] otherwise
 )
}
```

```
function posiciónDe_enLaQueAparece_(unaLista, unElemento){
 /*
 Propósito: Describe la posición en **unaLista** donde esté el elemento **unElemento**.
 Parámetros:
 * unaLista : Lista de elementos - La lista de elementos a saber la posición del elemento.
 * unElemento: Elemento - El elemento a saber su posición.
 Precondición: Debe estar **unElemento** en la lista **unaLista**.
 Tipo: Número.
 */

 posición := 0
 restoLista := unaLista
 while(primero(restoLista) /= unElemento){
 posición := posición + 1
 restoLista := resto(restoLista)
 }

 return(posición)
}
```

```
function lista_estáIncluidaEn_(primerLista, segundaLista){
 /*
 Propósito: Indica si la lista **primerLista** se encuentra contenida en
la lista **segundaLista**.  Parámetros:
 * primerLista : Lista de elementos - La lista a saber si está contenida en la otra.
 * segundaLista: Lista de elementos - La lista a saber si contiene a la otra.
 Precondiciones:
 * Las dos listas no contienen elementos repetidos.
 * La lista **primerLista*** no puede estar vacía.

 Tipo: Booleano.
 */

 restoLista := primerLista
 while(not esVacía(restoLista) && contiene_A_(segundaLista, primero(restoLista))){
 restoLista := resto(restoLista)
 }

 return(esVacía(restoLista) || contiene_A_(segundaLista, primero(restoLista)))
}
```

```
function intersecciónDe_Con_(primerLista, segundaLista){
 /*
 Propósito: Describe una lista de todos los elementos que se encuentren a la vez
en **primerLista** y en  **segundaLista**.
 Parámetros:
 * primerLista : Lista de elementos - La primer lista.
 * segundaLista: Lista de elementos - La segunda lista.
 Precondiciones:
 * Las dos listas no contienen elementos repetidos.
 Tipo: Listas de elementos.
 */

 listaIntersección := []
 restoLista := primerLista
 while(not esVacía(restoLista)){
 listaIntersección := listaIntersección ++ listaSiContiene_A_(segundaLista,
primero(restoLista))  restoLista := resto(restoLista)
 }

 return(listaIntersección)
}
```

```
function listaSiContiene_A_(unaLista, unElemento){
 /*
Propósito: Describe una lista con el elemento **unElemento** si este está
en la lista **unaLista**.  Parámetros:
 * unaLista : Lista de elementos - La lista de elementos.
 * unElemento: Elemento - El elemento a saber si está en la lista.
 Precondiciones: Ninguna.
 Tipo: Listas de elementos.
 */

 return(
 choose
 [unElemento] when (contiene_A_(unaLista, unElemento))
 [] otherwise
 )
}
```

```
function uniónDe_Con_(primerLista, segundaLista){
 /*
 Propósito: Describe una lista sin repetidos que contenga todos los elementos que
aparezcan en las listas  **primerLista** y en **segundaLista**.
 Parámetros:
 * primerLista : Lista de elementos - La primer lista.
 * segundaLista: Lista de elementos - La segunda lista.
 Precondiciones:
 * Las dos listas no contienen elementos repetidos.
 Tipo: Listas de elementos.
 */

 listaUnión := primerLista
 restoLista := segundaLista
 while(not esVacía(restoLista)){
 listaUnión := listaUnión ++
listaDeElemento_SiNoEstáEnLista_(primero(restoLista), listaUnión)
restoLista := resto(restoLista)
 }

 return(listaUnión)
}
```

```
function mínimoElementoDe_(unaListaDeNúmeros){
 /*
 Propósito: Describe el elemento más chico que se encuentra en la lista
**unaListaDeNúmeros**.  Parámetros:
 * unaListaDeNúmeros : Lista de números - La lista de números a buscar el más chico.
 Precondiciones:
 * Debe haber al menos un elemento en la lista.
 Tipo: Número.
 */

 mínimoElemento := primero(unaListaDeNúmeros)
 restoLista := resto(unaListaDeNúmeros)
 while(not esVacía(restoLista)){
 mínimoElemento := mínimoEntre_Y_(mínimoElemento, primero(restoLista))
 restoLista := resto(restoLista)
 }

 return(mínimoElemento)
}
```

```
function sinElMínimoElemento_(unaListaDeNúmeros){
 /*
 Propósito: Describe la lista que se obtiene después de eliminar una única vez el
elemento más chico de la lista  **unaListaDeNúmeros**.
 Parámetros:
 * unaListaDeNúmeros : Lista de números - La lista de números a eliminar una
única vez el más chico.  Precondiciones:
 * Debe haber al menos un elemento en la lista.
 Tipo: Lista de números.
 */

 listaSinElMínimo := []
 restoLista := unaListaDeNúmeros
 while(primero(restoLista) /= mínimoElementoDe_(unaListaDeNúmeros)){
 listaSinElMínimo := listaSinElMínimo ++ [primero(restoLista)]
 restoLista := resto(restoLista)
 }
 listaSinElMínimo := listaSinElMínimo ++ resto(restoLista)

 return(listaSinElMínimo)
}
```

```
function lista_ordenada(unaListaDeNúmeros){
 /*
 Propósito: Describe la lista con los mismos elementos de **unaListaDeNúmeros**, pero
ordenada de menor a mayor.  Parámetros:
 * unaListaDeNúmeros : Lista de números - La lista de números a ordenar
de menor a mayor.  Precondiciones:
 * Debe haber al menos un elemento en la lista.
 Tipo: Lista de números.
 */
 listaOrdenada := [mínimoElementoDe_(unaListaDeNúmeros)]
 restoLista := sinElMínimoElemento_(unaListaDeNúmeros)
 while(not esVacía(restoLista)){
 listaOrdenada := listaOrdenada ++ [mínimoElementoDe_(restoLista)]
 restoLista := sinElMínimoElemento_(restoLista)
 }

 return(listaOrdenada)
}
```

```
function máximoElementoDe_(unaListaDeNúmeros){
 /*
 Propósito: Describe el elemento más grande que se encuentra en la lista
**unaListaDeNúmeros**.  Parámetros:
 * unaListaDeNúmeros : Lista de números - La lista de números a buscar el más grande.
 Precondiciones:
 * Debe haber al menos un elemento en la lista.
 Tipo: Número.
 */

 máximoElemento := primero(unaListaDeNúmeros)
 restoLista := resto(unaListaDeNúmeros)
 while(not esVacía(restoLista)){
 máximoElemento := máximoEntre_Y_(máximoElemento, primero(restoLista))
 restoLista := resto(restoLista)
 }

 return(máximoElemento)
}
```

```
function sinElMáximoElemento_(unaListaDeNúmeros){
 /*
 Propósito: Describe la lista que se obtiene después de eliminar una única vez el
elemento más grande de la lista  **unaListaDeNúmeros**.
 Parámetros:
 * unaListaDeNúmeros : Lista de números - La lista de números a eliminar una
única vez el más grande.  Precondiciones:
 * Debe haber al menos un elemento en la lista.
 Tipo: Lista de números.
 */

 listaSinElMáximo := []
 restoLista := unaListaDeNúmeros
 while(primero(restoLista) /= máximoElementoDe_(unaListaDeNúmeros)){
 listaSinElMáximo := listaSinElMáximo ++ [primero(restoLista)]
 restoLista := resto(restoLista)
 }
 listaSinElMáximo := listaSinElMáximo ++ resto(restoLista)

 return(listaSinElMáximo)
}
```