Ejercicio 3 práctico 3.1:

Datos:

- Se desea realizar un viaje en un automóvil con autonomía A.
- Desde la localidad l0 hasta la localidad ln pasando por las localidades l1, … , ln-1 (en ese orden).
- Se conoce cada distancia di <= A entre la localidad li-1 y la localidad li.
- Existe una estación de combustible en cada una de las localidades.

Escribir un algoritmo que compute el menor número de veces que es necesario cargar combustible para realizar el viaje, y las localidades donde se realizaría la carga.

Entonces si voy de l0 a ln tengo que pasar por n+1 localidades, donde la distancia es conocida. Y mi auto tiene autonomía de A km, entonces si tengo una lista con las localidades, voy a ir recorriendo esa lista de izquierda a derecha, y viendo el primer elemento. Si no tengo autonomía para ir a esa localidad la eliminaré de la lista. Pero en caso de tener autonomía tengo que añadirla a una lista donde voy guardando los resultados y luego se elimina.

El criterio de selección: **Cargó combustible en las localidades cuando no me alcanza para llegar a destino.**

La estructura:

```
type Localidad = tuple
                     id: nat
                     distancia: nat
               end tuple

fun viaje (l: List of Localidad, in autonomia: nat) ret loc_carga:
                                                          List of Localidad
      var l_copy: List of Localidad
      var loc: Localidad
      var a: nat
      var nafta: nat

      nafta := 0
      a := autonomia
      l_copy = copy_list(l)
      loc_carga := empty_list()

      while not is_empty(l_copy) do
            loc := head(l_copy)
            if a < loc.distancia then
                  tail(l_copy)
            else
                  nafta := nafta + 1
                  a := a + 1
                  addr(loc_carga, loc)
                  tail(l_copy)
            if
```

```
      od
      destroy(l_copy)
end fun
```

Ejercicio 4 (Ballena)

Datos:

- n ballenas varadas en una playa.
- se conocen los tiempos s1,...,sn que cada ballena sea capaz de sobrevivir hasta que la asista un equipo de rescate.

Dar un algoritmo voraz que determine el orden en que deben ser rescatadas para salvar el mayor número posible de ellas, asumiendo que llevar una ballena mar adentro toma tiempo constante t, que hay un único equipo de rescatee y que una ballena no muere mientras está siendo regresada mar adentro.

El criterio de selección: **el salvar a la ballena que tiene menor si (tiempo que es capaz de sobrevivir).**

La estructura:

```
type Ballena = tuple
                  id: nat
                  tiempoRestante: nat
            end tuple

fun salvarBallena(l: Set of Ballena, t: nat) ret rescatadas: List
                                                              of Ballena
      var ballenasAunVivas: Set of Ballena
      var hora: nat
      var ballena: Ballena

      hora := 0 {- para llevar el contador de la hora -}
      ballenasAunVivas := set_copy(l)
      rescatadas := empty_list()

      while not is_empty_set(ballenasAunVivas) do
            ballena := elegirBallena(ballenasAunVivas)
            addR(rescatadas, ballena)
            elim_set(ballenasAunVivas, ballena)
            hora := hora + t
            quitarMuertas(ballenasAunVivas, hora)
      od
      destroy(ballenasAunVivas)
end fun

fun elegirBallena(l: Set of Ballena) ret b: Ballena
      var b_aux: Ballena
      var min: nat
      var l_aux: Set of Ballena
```

```
    var min := +infinito
    l_aux := set_copy(l)

    while not is_empty(l_aux) do
          b_aux := get(B_aux)
          if b_aux.tiempoRestante < min then
                min := b_aux.tiempoRestante
                b := b_aux
          fi
          elim_set(Baux, b_aux)
    od
    destroy_set(l_aux)
end fun

proc quitarMuertas(in/out l: Set of Ballenas, hora: nat)
    var d: Set of Ballena
    var b: Ballena

    d := copy_set(l)
    while not is_empty(d) do
          b := get(d)
          if b.tiempoRestate < hora then
                elim(l, b)
          fi
          elim(d, b)
    od
    destroy_set(d)
end proc
```

Ejercicio 5 (teléfono satelital)

Datos:

- n amigos para que se lo llevan de vacaciones
- Cada uno va a un lugar diferente
- Se conocen los dias de partida y regreso de cada uno

El criterio: **le doy el teléfono al que tenga menor fecha de regreso**

```
type Amigo = tuple
                  nombre: string
                  partida: nat
                  regreso: nat
             end tuple

fun prestarCel(l: Set of Amigo) ret ls: List of Amigo
    var t: nat
    var aux_l: Set of Amigo
    var a: Amigo

    t := 0
    ls := list_empty()
```

```
    aux_l := set_copy(l)

    while not is_empty(aux_l) do
          a := select_amigo(aux_as)
          addr(ls, a)
          elim_set(aux_as, a)
          t := t + a.regreso
          elim_no_prestamos(aux_l, t)
    od
    set_destroy(aux_as)
end fun

proc elim_no_prestamos(in/out l: Set of Amigo, in t: nat)
    var aux_l: Set of Amigo
    var a: Amigo

    aux_l := set_copy(l)

    while not is_empty(aux_l) do
          a := get(aux_as)
          if a.regreso <= t then
                elim_set(l, a)
          fi
          elim_set(aux_l, a)
    od
    set_destroy(aux_l)
end proc

fun select_amigo(l: Set of Amigo) ret a: Amigo
    var aux_l: Set of Amigo
    var min: nat
    var a: Amigo

    min := +infinito
    aux_l := set_copy(l)
    while not is_empty(aux_l) do
          b := get(aux_l)
          if b.regreso < min then
                min := b.regreso
                a := b
          fi
          elim_set(aux_l, b)
    od
    set_destroy(aux_l, b)
end fun
```

Ejercicio 6 (panadería)

Datos:

- Abrir el horno el menor número de veces
- Hay n piezas de panadería
- Cada pieza tiene un tiempo mínimo y un tiempo máximo, si se la extrae antes del min queda cruda y si es después del max se quema

¿Qué criterio utiliza un algoritmo voraz para extraer todas las piezas del horno piezas del horno en perfecto estado?

Datos:
    n facturas
    conocemos tiempo mínimo (ti) y tiempo máximo (Ti) de cocción
    Queremos abrir el horno la menor cantidad de veces

Criterio:
    Tomamos la factura que menor tiempo máximo tenga y sacamos todas las facturas que ya estén listas en ese tiempo

Representación:

```
    type Bun = tuple
                     id: nat
                    min_time: nat
                    max_time: nat
               end tuple

fun open(buns: Set of Bun) ret t: nat
    var in_oven: Set of Bun
    var next_bun: Bun
    var hour: nat

    {- Comenzamos con todas las facturas en el horno -}
    in_oven := copy_set(buns)
    t := 0
    hour := 0

    while (!is_empty_set(in_oven)) do
        {- Elegimos la factura a sacar de acuerdo al criterio -}
        next_bun := choose_bun(in_oven)
        {- Establecemos la hora en la que abrimos el horno -}
        hour := next_bun.max_time
        {- Añadimos uno a la cuenta de veces que abrimos el horno -}
        t := t + 1
        {- Sacamos todas las facturas que ya estén listas -}
        remove_ready(in_oven, hour)
    od
end fun

fun choose_bun(buns : Set of Bun) ret chosen: Bun
    var aux_bun : Bun
    var min_max_time : nat
```

```
    var aux_set : Set of Bun

    {- Elegimos la factura que tenga el menor tiempo máximo en el horno -}
    min_max_time := +infinito
    aux_set := copy_set(buns)
    while (!is_empty_set(aux_set)) do
        aux_bun := get(aux_set)
        if aux_bun.max_time < min_max_time then
            min_max_time := aux_bun.max_time
            chosen := aux_bun
        fi
        elim_set(aux_set, aux_bun)
    od
    destroy_set(aux_set)
end fun

proc remove_ready(in/out buns : Set of Bun, in hour : nat)
    var aux_set : Set of Bun
    var aux_bun : Bun

    aux_set := copy_set(buns)

    while (!is_empty_set(aux_set)) do
        aux_bun := get(aux_set)
        if (aux_bun.min_time < hour) then
            elim_set(buns, aux_bun)
        fi
        elim_set(aux_set, aux_bun)
    od
    destroy_set(aux_set)
end proc
```

---

9. (*sobredosis de limonada*) Es viernes a las 18 y usted tiene ganas de tomar limonada con sus amigos. Hay n bares cerca, donde cada bar i tiene un precio P_i de la pinta de limonada y un horario de happy hour H_i, medido en horas a partir de las 18 (por ejemplo, si el happy hour del bar i es hasta las 19, entonces $H_i = 1$), en el cual la pinta costará un 50% menos. Usted toma una cantidad fija de 2 pintas por hora y no se considera el tiempo de moverse de un bar a otro. Se desea obtener el menor dinero posible que usted puede gastar para tomar limonada desde las 18 hasta las 02 am (es decir que usted tomará 16 pintas) eligiendo en cada hora el bar que más le convenga.

Datos:
    Desde 18 a 02 tomamos 2 pintas por hora (16 total)
    Conocemos el precio Pi y el horario Hi de cada bar

Criterio:
    Para cada hora elegimos el bar que más barato sea, considerando también los precios con descuento gracias a los happy hour.

Representación:
```
    type Bar = tuple
                    price : nat
                    happy_hour : nat
```

```
               end tuple

fun limonada(bars: Set of Bar) ret final_price: nat
    var hour: nat
    var current_bar: Bar

    {- Comenzamos a las 18, que es nuestra 'hora 0' habiendo gastado $0 -}
    hour := 0
    final_price := 0

    while (hour < 8) do
        {- Elegimos el bar que visitaremos en esa hora según el criterio -}
        current_bar := bar(bars, hour)
        {- Sumamos el precio de tomar 2 pintas en ese bar -}
        final_price := final_price + (2 * current_bar.price)
        {- Modificamo la hora para continuar con la ejecución -}
        hour := hour + 1
    od
end fun

fun bar(bars: Set of Bar, hour: nat) ret chosen: Bar
    var aux_set : Set of Bar
    var aux_bar : Bar
    var min_price : nat

    aux_set := copy_set(bars)
    min_price := inf

    while (!is_empty_set(aux_set)) do
        aux_bar := get(aux_set)
        if (aux_bar.happy_hour < hour && aux_bar.price/2 < min_price) then
            chosen := aux_bar
            min_price := (aux_bar.price)/2 {- se divide por el happy -}
        else if (aux_bar.price < min_price) then
            chosen := aux_bar
            min_price := aux_bar.price
        fi
        elim_set(aux_bar, aux_set)
    od
    destroy_set(aux_set)
end fun
```

Ejercicio 7 (Submarino)

datos

- n sobrevivientes en el interior.
- c1,...,cn son las cantidades de oxígeno que cada uno consume por minuto.
- se puede rescatar a uno por vez.
- El rescate lleva t minutos.

Criterio salvar a quienes consumen más oxígeno.

```
type Submarino = tuple
                   id: String
                   oxigeno: Float
             end tuple

fun rescate_a(as: array[1..N] of Submarino, C: float, t: nat) ret
                                             ls: List of Submarino
      var oxigeno: float
      var aux_as: array[1..N] of Submarino
      var s: Submarino

      aux_as := copy(as)
      sort_submarino(as) {-ordena de mayor a menor-}
      oxigeno := C
      ls := empty_list()
      i := 1

      while oxigeno > 0 and i <= N do
            s := get(aux_as[i])
            addr(ls, s)
            oxigeno := oxigeno - s.oxigeno
      od
      destroy(aux_as)
end fun
```

Ejercicio 8 (estufa)

Datos:

- Tengo una estufa y n troncos de leña
- Todos los troncos del mismo tamaño y en la estufa entra uno por vez
- cada tronco i irradia temperatura k y dura t

Se requiere encontrar el orden en que se utilizarán la menor cantidad posible de troncos a quemar entre las 22 y las 12hs del día siguiente, asegurando que entre las 22 y las 6 la estufa irradia constantemente una temperatura no menor a K1 y entre las 6 y las 12 am, una temperatura no menos a K2.

Criterio de selección:

Durante las 22 y las 6 (8hs) elijo el tronco con tiempo mayor, tal que ki es mayor o igual a K1.
Entre las 6 y las 12 (6hs) idem, pero con K2

```
type Tronco = tuple
                  id: nat
                  calor: float
                  tiempo: float
            end tuple


fun estufa_voraz(S: Set of Tronco, K1: float, K2: float) res ls:
                                                       List of Tronco
      var C: Set of Tronco
      vat t: Tronco
      var h: float

      C := copy_set(S)
      h := 0
      ls := empty_list()

      while h < 14 do
            if h < 8 then
                  t := elegirTronco(C, K1)
            else
                  t := elegirTronco(C, K2)
            fi
            addR(ls, t)
            elim_set(C, t)
            h := h + t.tiempo
      od
      destroy_set(C)
end fun

fun elegirTronco(C: Set of tronco, K: float) ret t: Tronco
      var B: Set of Tronco
      var max_tiempo: Float
      var t´: Tronco

      max_tiempo := -infinito
      B := copy_set(C)

      while not set_empty(B) do
            t´ := get(B)
            if t´.tiempo > max_tiempo && t´.calor >= K then
                  max_tiempo := t´.tiempo
                  t := t´
            fi
            elim(B, t´)
      od
      destroy(B)
end fun
```

Ejercicio 2

```
proc p (in/out l: list)
  var a, b: pointer to node
  a:= l
  while a /= null do
    b:= a→next
    if b /= null then
      a→next := b→next
      free(b)
    fi
    a:= a → next
  od
end proc
```

a) p toma como argumento una l: list, donde luego usa un a aux que es igual a l, después hace un while mientras a sea distinto de null. Luego inicializa a b con el segundo elemento de a. y si b es distinto de null, y luego libera el b. Es decir, que elimina los elementos en las posiciones impares de la lista mientras los array arranquen en 0.
b) El orden es n recorre el array una vez.

Ejercicio 3
Dada la siguiente función:

```
fun f(n: nat) ret m: nat
  if n ≤ 1 then m:= 2 * n
  else
    m := 1
    for i:= n downto 1 do
      m := n * m
    od
    m := 3 * f(n div 2)
  fi
end fun
```

a) cantidad de llamadas recursivas hay -> 1
b) Expresar la ecuación de recurrencia en función de la cantidad de asignaciones a la variable m.

   t(n) = “Calcula la cantidad de asignaciones a m, cuando pasamos n como argumento”

   t(n) = 1 si n <= 1
   = 1 + n + t(n/2) cc

   c) Orden de asignaciones a la variable m

   g(n) = n + 1

   a = 1 -> núm que multiplica a t()

```
b = 2 -> t(n/b)
k = 1 -> es el grado de g(n)

Como a  < b^k, entonces esta implementación tiene orden lineal: n
```

```
Formas de decidir:
t(n) = at(n/b) + g(n) b in N, g(n) in O(n^k)

O(n^(log_b a))   si a > b^k
O(n^k log n)     si a = b^k
O(n^k)           si a < b^k
```

Ejercicio 1

Un colectivero conduce su pequeño colectivo. Muy pequeño. Solamente hay lugar para un pasajero. Más que un colectivo, parece una moto. Su recorrido o viaje va de la parada 1 hasta la parada n pasando por las paradas intermedias 2, 3, ..., n − 1. Hay m pasajeros esperando. Para cada pasajero i sabemos en qué parada se quiere subir (s_i), y en qué parada se va a bajar (b_i con 1 ≤ s_i < b_i ≤ n). La intención del colectivero es trasladar en un viaje a la mayor cantidad de pasajeros posible. El colectivero no tiene obligación de levantar un pasajero por más que esté libre, puede preferir reservarlo para un pasajero que sube después. **Se debe obtener el número máximo de pasajeros trasladables en un único viaje.** Se pide lo siguiente:

(a) Indicar de manera simple y concreta, cual es el criterio de selección voraz para construir la solución?
(b) Indicar qué estructuras de datos utilizarías para resolver el problema.
(c) Explicar en palabras como resolvería el problema el algoritmo.
(d) Implementar el algoritmo en el lenguaje de la materia de **manera precisa.**

(a) **El criterio de selección es subir, siempre que podamos al pasajero que más pronto se baje.**
(b)
```
type Pasajero
      id: nat
      s: nat
      b: nat
end tuple
```
(c) Recorremos el conjunto y seleccionamos al pasajero que primero se baja, descartamos a los que no sean compatibles (quienes se suben cuando ya está montado nuestro pasajero, es decir, vamos a descartar a los pasajeros cuya subida sea anterior a la bajada del pasajero seleccionado) y a partir de ahí de los restantes repetimos el proceso hasta que no haya más pasajeros, o sea la última parada.

```
fun motito(p: Set of Pasajero) ret res: nat
      var p_aux: Set of Pasajero
      var w: Pasajero
```

```
	p_aux := copy_set(p)

	while not is_empty_set(p_aux) do
		w := elegir_pasajero(p_aux)
		elim_set(p_aux, w)
		res := res + 1
		remover_pasajeros(p_aux, w)
	od
	destroy(p_aux)
end fun

fun elegir_pasajero(p: Set of pasajero) ret res: Pasajero
	var p_aux: Set of pasajero
	var w: Pasajero
	var min: nat

	p_aux := set_copy(p)
	min := +infinito

	while not is_empty_set(p_aux) do
		pasajero := get(p_aux)
		if pasajero.b < min then
			res := pasajero.b
			min := res
		fi
	od
	destroy(p_aux)
end fun

proc remover_pasajeros(in/out p: Set of pasajero, in pasajero:
                                                            Pasajero)
	var p_aux: Set of Pasajero
	var w: Pasajero

	p_aux := copy_set(p)
	while not is_empty_set(p_aux) do
		w :=  get(p_aux)
		if w.s < pasajero.b then
			elim(p, w)
		fi
		elim(p_aux, w)
	od
	destroy_set(p_aux)
end proc
```

---

fun minimo(a: array[1..n] of nat, i, k: nat) **ret** m : **nat**

```
  var j : nat
  if i = k then m := a[i]
  else
    j := (i + k) div 2
    m := min(minimo(a, i, j), minimo(a,j+1,k))
  fi
end fun
```

Corramos el algoritmo con a=[2,5,1,8]

minimo(a,1,4)

i = k false entonces va al else

j := 5 div 2 -> 2 entonces
min(minimo(a,1,2), minimo(a,3,4))

veamos minimo(a,1,2)
i = k es false entonces va al else
j = 3 div 2 = 1
min(minimo(a,1,1), minimo(a,2,2))

i = k entonces a[i]

min(2,5) = 2
entonces minimo(3,4) = 1

entonces m = 1

Es del tipo divide y vencerás:

El tamaño del problema es k-i.
Si el arreglo tiene 1 elemento se hace 1 asignación.

Se llama recursivamente 2 veces.

a = 2
b = 2
k = 0

t(m) = Cantidad de operaciones que realiza el algoritmo minimo(a,i,k) donde m es k-i

```
t(m) = 1            ,si m=1
       2t(m/2) + 1  ,si m>1
```

a ? b^k

2 > 2^0 por lo tanto es orden m^(log_2 2) == m osea el orden es m -> es lineal.

---

1. Calculá el orden de complejidad de los siguientes algoritmos:

(a)
```
proc f1(in n : nat)
    if n ≤ 1 then skip
    else
        for i := 1 to 8 do f1(n div 2) od
        for i := 1 to n^3 do t := 1 od
```

(b)
```
proc f2(in n : nat)
    for i := 1 to n do
        for j := 1 to i do t := 1 od
    od
    if n > 0 then
        for i := 1 to 4 do f2(n div 2) od
```

```
a)

t(m) = Número de operaciones realizadas por f1(n) donde m es n/2

El skip es O(0)

t(m) = 0                  si m = 1
       8t[m/2] + n^3      cc

entonces a=8, k=3, b=2.

entonces 8 = 2^3 por lo que la complejidad es n^3 log n

b)

t(m)  = Número de operaciones realizadas por f2(n) donde m es n/2

t(m) = ?   si n = 0

ops(C) = ops(for i:=1 to n do C(i) od)
       = \sum_{i=1}^n ops(for j:=1 to n do C(i,j) od)
       = \sum_{i=1}^n \sum_{j=1}^i ops(t:=1)
       = \sum_{i=1}^n \sum_{j=1}^i 1
       = \sum_{i=1}^n i = n*(n+1)/2

Entonces:

t(m) = n*(n+1)/2     si n = 0
       at(n/b) + g(n)

donde -> a = 4, b = 2, k=0
Entonce 4 > 2^0 -> n^(log2 4) = n ^ (log 2 2^2) = n^2 (log 2 2) = n^2
```

8. Calculá el orden de complejidad del siguiente algoritmo:

> **proc** $f3(n : \textbf{nat})$
>   **for** $j := 1$ **to** 6 **do**
>   **if** $n \leq 1$ **then skip**
>   **else**
>   **for** $i := 1$ **to** 3 **do** $f3(n$ **div** $4)$ **od**
>   **for** $i := 1$ **to** n^4 **do** $t := 1$ **od**
>   **od**

t(m) = Número de operaciones realizadas por f3()

Primero tenemos un for que se ejecuta 6 veces, independientemente del valor de n.

Si n <= 1 -> tenemos que hacer el skip 6 veces, pero como es O(0), entonces t(m) = 0

Para los otros casos, tengo que buscar los parámetros de la ecuación:

t(m) = at(n/b) + g(n)

Entonces, solo viendo lo que está en el bloque del else

a = 3 -> porque la llamada recursiva está dentro de un for que va de 1 a 3
b = 4 -> por el n div 4

y g(n)?

```
ops(C) = ops(for i:=1 to n^4 do C od)
      = \sum_{i=1}^{n^4} ops(t:=1)
      = \sum_{i=1}^{n^4} 1
      = n^4
```

entonces b = 4, a = 18 y k=4 por lo que 18 < 4^4 entonces n^4.

1. (Voraz) Anacleta reparte pedidos en bicicleta. Todos los clientes viven en la Avenida San Wachín y la dirección del trabajo de Anacleta es Avenida San Wachín 0. Al llegar a trabajar, Anacleta tiene cada pedido con la altura de la casa donde debe entregarlo. En la bici, Anacleta puede transportar cinco pedidos o menos. Escriba un algoritmo que reciba una lista de naturales llamada pedidos y determine la menor distancia que debrá pedalear Anacleta para entregar todos los pedidos.

Anacleta reparte pedidos en bicicleta
Al llegar a trabajar Anacleta tiene cada pedido con la altura de la casa donde debe entregarlo.
En la bici, puede transportar 5 pedidos o menos.

Escriba un algoritmo que reciba una lista de naturales llamadas pedidos y determine la menor distancia que deberá pedalear Anacleta para entregar todos los pedidos.

tipo de datos el array que define la consigna

Criterio de selección: Entregar primero el pedido más lejos.

```
fun anacleta(pedidos: array[1..N] of nat) ret r: nat
	var pedidos_aux: array[1..N]
	var i: nat

	copy_array(pedidos, pedidos_aux)
	reverse_sort(pedidos_aux) {- ordena de mayor a menor -}

	r := 0
	i := 1

	while i <= N do
		r := r + pedido_aux[i]*2
		i := i + 5
	od
end fun
```

1. (Algoritmos voraces) Te vas n días de vacaciones al medio de la montaña, lejos de toda civilización. Llevás con vos lo imprescindible: una carpa, ropa, una linterna, un buen libro y comida preparada para m raciones diarias, con $m > n$. Cada ración i tiene una fecha de vencimiento v_i, contada en días desde el momento en que llegás a la montaña. Por ejemplo, una vianda con fecha de vencimiento 4, significa que se puede comer hasta el día número 4 de vacaciones inclusive. Luego ya está fuera de estado y no puede comerse.

   Tenés que encontrar la mejor manera de organizar las viandas diarias, de manera que la cantidad que se vencen sin ser comidas sea mínima. Deberás indicar para cada día j, $1 \leq j \leq n$, qué vianda es la que comerás, asegurando que nunca comas algo vencido.

- Hay n días de vacaciones al medio de la montaña
- m raciones diarias, con m>n
- Cada ración i tiene un fecha de vencimiento vi, contada en días desde el momento en que llegas a la montaña

Hay que encontrar la mejor manera de organizar las viandas diarias, de manera que la cantidad que se vencen sin ser comidas sea mínima.

Criterio de selección = Comer primero las viandas que tiene menor fecha de vencimiento.

```
type Vienda = tuple
            id: nat
            vencimiento: nat
      end tuple
```

Armar un arreglo de índice y ordenarlo de menor a mayor de acuerdo al vencimiento, recuerdo en una variable i el índice de la última vianda que consideré.

Para cada día j, me fijo si la vianda actual se puede comer, si venció voy a la siguiente hasta encontrar una que pueda comer. Cuando encuentro una que se pueda comer, selecciona esa para el día j y actualizo el valor de i.

```
fun viandas(venc: array[1..m] of nat, n: nat) ret r: array[1..n] of nat
      var venc_aux: array[1..m] of nat
      var i: nat

      index_sort(vec, venc_aux)
{- devuelve en venc_aux los índices del arreglo venc ordenados de acuerdo a
los valores de venc de menor a mayor -}

      i := 1

      for j:=1 to n do
            {- elijo la ración para el día-}
            while venc[venc_aux[i]] < j do
                  i:=i+1
            od
            r[j] := venc_aux[i]
            i := i+1
      od
end fun

fun viandas(venc: array[1..m] of nat, n: nat) ret r: array[1..n] of nat
      var venc_aux: array[1..m] of nat
      var i: nat
      index_sort(venc, venc_aux)
      i := 1

      for j:=1 to n do
            while i<m && venc[venc_aux[i]] < j do
                  i := i+1
            od
            if i>m then
                  r[j] := -1
            else
                  r[j] := venc_aux[i]
                  i := i+1
```

```
            fi
      od
end fun
```

con conjuntos:

```
fun viandas(venc: Set of Vianda, n: nat) ret r: List of Vianda
      var venc_aux: Set of Vianda
      var v: Vienda
      var i: nat

      i := 0
      venc_aux := set_copy(venc)
      r := empty_list()

      while not is_empty_set(venc_aux) do
            v := seleccionar_vianda(venc_aux)
            add(r, v)
            elim_set(venc_aux, v)
            i := i + 1 {- cuento los días de vacaciones -}
            eliminar_vencidas(venc_aux, i) {- elimina las viandas menores al
            dia i-}
      od
end fun

{-Selecciona la vianda a comer el día i, siempre y cuando no este vencida, es
como un mín-}
fun seleccionar_vianda(v: Set of Vianda) ret r: Vianda
      var aux_v: Set of Vianda
      var w: Vianda
      var min: nat
      var n: nat

      n := 0 {- Contador de días -}
      min := +infinito
      aux_v := copy_set(v)

      while not is_empty_set(aux_set) do
            w := get(aux_set)
            if w.vencimiento < min then
                  min := w.vencimiento
                  r := w
                  n := n + 1
            fi
            elim_set(aux_set, n)
      od
      destroy_set(aux_set)
endo fun

{-Elimina las viandas que ya vencidas-}
proc eliminar_vencidas(in/out v: Set of Vianda, dia: nat)
      var aux_v: Set of Vianda
      var w: Vianda
```

```
    aux_v := set_copy(v)

    while not is_empty_set(aux_v) do
          w := get(aux_v)
          if w.vencimiento < dia then
                elim(v, w)
          fi
          elim(aux_v,w)
    od
    destroy(aux_v)
end fun
```

---

2. Para cada uno de los siguientes algoritmos determinar **por separado** cada uno de los siguientes incisos.

   (a) ¿Qué hace?

   (b) ¿Cómo lo hace?

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

   (d) Proponer nombres más adecuados para los identificadores (de variables y procedimientos).

```
proc p(in/out a : array[1..n] of int)
    var i : nat
    i:= 1
    while i <= n do
        swap(a,i,q(a,i))
        i:= i+1
    od
end proc
```

```
fun q(a : array[1..n] of int, i : nat) ret j: nat
    var m,k : nat
    j:= i
    if i mod 2 == 0 then
        m:= a[i]
        k:= i+2
        while k ≤ n do
            if a[k] < m then
                m:= a[k]
                j:= k
            fi
            k:= j+2
        od
    fi
end fun
```

```
Ejemplo de ejecución de q() con [2,3,4,5], i=1

j:=1
if i mod 2 == 0 (1 mod 2 == 1) -> entonces j:=1

j:=2
if i mod 2 == 0 (2 mod 2 == 0) es true, entonces entra al if
    m:=a[2] (m = 3)
    k:=i+2 (k = 4)
    while (4 <= 4) true
          if a[4] < 3 then (5 < 3 es false)
          -> k:=4

j:=3
if i mod 2 == 0 (3 mod 2 == 1) -> j := 3

j:=4
if i mod 2 == 0 (4 mod 2 == 1) es true
    m:= a[4] = 5
    k:=6
    while 6 <= 5 false -> j:=3

Otro ejemplo [3,2,5,4]
```

```
j:=1
if i mod 2 (1 mod 2 == 1) false

j:=2
if i mod 2 (2 mod 2 == 0) true
      m:=a[2] = 2
      k:=4
      while 4 <= 4 do (true, entro al while)
            if a[k] < m then (4 < 2) Nooo
```

(a) la función q toma un arreglo de longitud n y un natural i, devuelve el mismo natural i si es impar y si no devuelve el índice del menor elemento del arreglo desde la posición im de entre todos los que se encuentren en posiciones impares
(b) En caso que i sea par, recorre el arreglo solo en las posiciones pares de izquierda a derecha, desde la posición i y calcula el índice del menor de esos elementos
(c) orden (n-i)/2

ahora con p:

(a) Toma un arreglo de tamaño n, ordena de menor a mayor los elementos que se encuentran en posiciones pares de a.
(b) Recorre el arreglo de izquierda a derecha, desde la posición 1, para cada elemento, si está en posición impar lo deja igual, si no, lo intercambia por el mínimo elemento de entre los que están en posiciones impares
(c) es n^2

____________________________________ Ejercicio 3 final 4 de julio 2017

n habitaciones, consecutivas de 1 a n.
carrito con capacidad de 5 desayunos.
nos dicen cuántos desayunos hay que dar a cada habitación.
Se debe dar el orden en que se entregan los desayunos de manera de recorrer la mínima distancia entregando todo.

Criterio = atiendo los pedidos de las habitaciones más lejanas

```
type Entrega = tuple
                  hab: nat
                  cant: nat
            end tuple

fun desayuno(p: array[1..N] of nat) ret r: List of of Entrega
      var en_carro: nat
      var p_rest: array[1..N] of nat
      var viaje_actual: List of Entrega
      var entrega_actual: Entrega

      for i:=1 to n do
            p_rest[i] := p[i]
      od

      en_carro := 5
      r := empty_list()
      viaje_actual := empty_list()
      addr(r, viaje_actual)

      for h:=N downto 1 do
            while p_rest[h] > 0 do
                  if en_carro >= p_rest[h] then
                        entrega_actual.hab := h
                        entrega_actual.cant := p_rest[h]
                        addr(viaje_actual, entrega_actual)
                        p_rest[h] := 0
                  else
                        entrega_actual.hab := h
                        entrega_actual.cant := en_carro
                        p_rest[h] := p_rest[h] - en_carro
                        addr(viaje_actual, entrega_actual)
                        viaje_actual := empty_list()
                  fi
            od
      od
end fun
```

____

```
proc p(a: array[1..n] of nat)
      var d: nat
      for i:= 1 to n do
            d = i
            for j=i+1 to n do
                  if a[j] < a[d] then d=j fi
            od
            swap(a,i,d)
      od
end proc
```

```
si tenemos [5,4,6,7] -> [1,2,3,4]

primera ejecución
for i:=1 to 4 do
      d := 1
      for j=2 to 4 do
            if a[2] < a[1] (4 < 5) si entonces d=j (d=2)

todavía estoy en el for de adentro entonces, ahora con d=2
      for j=3 to 4 do
            if a[3] < a[2] (6 < 4) no entonces salgo del for y voy a

swap(a,1,2) -> [4,5,6,7]

for i:=2 to 4
      d:=2
      for j:=3 to n do
            if a[3] < a[2] (6 < 5) no entonces

swap(a,2,2) queda como esta

for i:=3 to 4
      d:=3
      for j:=4 to n do
            if a[4] < a[3] (7 < 6) no asique voy a swap(a,3,3)
queda el arreglo como esta [5,4,6,7] y termina, entonces ordena el arreglo
```

El primer for recorre el arreglo y en d va guardando los índices del arreglo a, luego en el segundo for se empieza a recorrer desde el segundo elemento del arreglo y compara ese elemento con el índice anterior que guardo en d y en casos que el segundo elemento sea menor que el primero el primer elemento es igual al segundo y caso contrario hace un swap de esos elementos.
Complejidad es n^2 xq recorre el arreglo 2 veces.

Complejidad?

```
Ops(C) = ops(for i:=1 to n do C1 od)
       = ops(for i:=1 to n do (ops(d:=1) + ops(j=i+1 to n do (if a[j] <
```

```
                                                   a[d]) then d:=j) od )) od)
       = sum_{i=1}^{n} (ops(for j+1 to no do(if a[j]<a[d])then d:=j)od)od)
       = sum_{i=1}^{n} sum_{j=i+1}^{n} ops(if a[j]<a[d] then d:=j od)
       = sum_{i=1}^{n} sum_{j=i+1}^{n} 1
       = sum_{i=1}^{n} n-i+2
       = sum_{i=1}^{n} n - sum_{i=1}^{n} i + sum_{i=1}^{n} 2
       = n*n - n(n+1)/2 + 2*n.
```

____ Calcular complejidad

```
t := 1
do t < n
      t :=t*2
od

Si n=1 -> t = 1 -> 1 < 1 -> false (1 operación)

Si n=2 -> t = 1 -> 1 < 2 -> true -> t = 2
          t = 2 -> 2 < 2 -> false (2 operaciones)

Si n=3 -> t = 1 -> 1 < 3 -> true -> t = 2
          t = 2 -> 2 < 3 -> true -> t = 4
          t = 4 -> 4 < 3 (3 operaciones)

Si n=4 -> t = 1 -> 1 < 4 -> true -> t = 2
          t = 2 -> 2 < 4 -> true -> t = 4
          t = 4 -> 4 < 4 (3 operaciones)


Si n=5 -> t = 1 -> 1 < 5 -> true -> t = 2
          t = 2 -> 2 < 5 -> true -> t = 4
          t = 4 -> 4 < 5 -> true -> t = 8
          t = 8 -> 8 < 5 (4 operaciones)


tengo que buscar una función t(ops). Entonces, t = 2^(operaciones -1)

t = 2^(1-1) = 2^0 = 1
t = 2^(2-1) = 2^1 = 2
t = 2^(3-1) = 2^2 = 4
t = 2^(4-1) = 2^3 = 8

entonces log t = (operaciones - 1)
         operaciones = log t + 1 -> O(log t)
```

____

```
proc r(in/out a: array[1..N] of int, in y: nat)
      for j:=y to n do
            m := j
            while m > y and a[m] < a[m-1] do
```

```
            swap(a,m,m-1)
            m := m-1
        od
    od
end proc
```

Complejidad?

```
ops(r) = sum_{j=y}^{n} ops(m:= j
                        while m > y and a[m] < a[m-1] do
                              swap(a,m,m-1)
                              m := m-1
                        od)
```

En el peor caso en el que a[m] < a[m-1] es siempre cierto y el while se recorre hasta que m = y(j-y) veces

Veamos con más detalle, que pasa con el while:

```
m:= j
while m > y and a[m] < a[m-1] do
      swap(a,m,m-1)
      m := m-1   -> O(1)
od
```

vemos que y es un parámetro que se le pasa a la función, supongamos que y = 1 en ese caso se recorre todo el arreglo con el for y se la asigna a m el índice que guarda j, entonces:

```
si y=1 -> m=1 entonces 1>1 es false.
si y=2 -> m=2 entonces 2>1 es true y se entra al while y a[2] < a[1]
            se swapean los elementos y luego m=1
       -> m=1 no se entra al ciclo.
```

Como m va disminuyendo en el while se recorre en ciclo y veces mientras y>1 xq sino no entra al while, el arreglo se recorre de izquierda a derecha.

Entonces:

ops(r) = sum_{j=y}^{n} y si

entonces en el peor caso y -> ops(r) sum_{j=i}^{n} = n = n^2 (por le for de afuera).

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

(a) ¿Qué hace? ¿Cuáles son las precondiciones necesarias para ello?

(b) ¿Cómo lo hace?

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

(d) Proponer nombres más adecuados para las funciones.

```
fun s(v: nat, p: array[1..n] of nat) ret y: nat
    y:= v
    while y < n ∧ p[y] ≤ p[y+1] do
        y:= y+1
    od
end fun
```

```
fun t(p: array[1..n] of nat) ret y: nat
    var z: nat
    y, z:= 0, 1
    while z ≤ n do
        y, z:= y+1, s(z,p)+1
    od
end fun
```

```
fun u(p: array[1..n] of nat) ret v: bool
    v:= (t(p) ≤ 1)
end fun
```

(a)

s() -> el índice del elemento de p tal que es mayor que el siguiente.
t() -> la cantidad de segmentos ordenados.
u() -> ve si el arreglo está ordenado.

(b)

s() -> recorre el arreglo desde la posición y avanza hasta que encuentra el primer elemento que es mayor que el siguiente

t() -> recorre el arreglo en segmento ordenados, usando s para determinar dónde termina cada segmento, contando la cantidad de segmentos que se recorrieron

u() -> llama a t() y se fija si el resultado es <= (igual 0 no va a dar nunca)

(c)
s() -> mejor caso O(1), cuando el primer elemento a considerar es mayor al siguiente. Peor caso O(n-v), cuando está ordenado desde v hasta el final.

t() -> mejor y peor caso O(n) ya que recorre el arreglo desde la posición cero hasta la última posición (por medio de s)

u() -> igual que t() entonces O(n)

```
proc r(in/out a : array[1..N] of int, in y : nat)
    for j:= y to n do
        m:= j
        while m > y ∧ a[m] < a[m-1] do
            swap(a,m,m-1)
            m:= m-1
        od
    od
end proc
```

```
a = [2,5,3,1], y = 1

for j:=1 to 4 do
      m:=1
      while 1 > 1 /\ a[1]<a[0] do (false no entra al while)

entonces

for j:=2 to 4 do
      m:=2
      while 2>1 /\ a[2] < a[1] do (5<2) false

for j:=3 to 4 do
      m:=4
      while 4>1 /\ a[4] > a[1] do (1<2 = true)
            swap(a,4,3) -> [2,5,1,3]
            m:=3


      while 3>1 /\ a[3] < a[2] do (3<5 = true)
            swap(a,3,2) -> [2,1,5,3]
            m:=2

      while 2>1 /\ a[2] < a[1] do (1<2 = true)
            swap(a,2,1) -> [1,2,5,3]
            m:=1

      while 1>1 = false

for j:=4 to 4 do
      m:=4
      while 4>1 /\ a[4] < a[3] (3<5 = true)
            swap(a,4,3) -> [1,2,3,5]
            m:=3

      while 3>1 /\ a[3] < a[2] (3<2) false

y termina la -> ordena el array el proc
```

1. (Algoritmos voraces) Estás en época de exámenes y tenés n materias cursadas, no correlativas entre sí, que podrías rendir. Cada materia tiene un día de examen: $d_1, \ldots, d_n$, y una cantidad de días previos **consecutivos** al examen que vos necesitás dedicar exclusivamente a su estudio: $c_1, \ldots, c_n$. También asumimos que el día que rendís un examen se dedica solamente a eso, no podés estudiar otra materia. Así por ejemplo si la materia "Bases de Datos" se rinde el día 10, y necesita 2 días de estudio, para poder rendirla tenés que dedicar el día 8 y 9 exclusivamente a la misma, y en el día 11 ya podrías empezar a estudiar otra materia. Se supone que solo estudiás la materia que estás por rendir, por más que te sobren días no comenzás a estudiar la siguiente para no confundir los temas.

Todos los d_i y los c_i son números naturales, inicialmente estamos al comienzo del día 1.

Se debe obtener la mayor cantidad de materias que podés rendir.

Se pide lo siguiente:

(a) Indicar de manera simple y concreta, cuál es el criterio de selección voraz para construir la solución?

(b) Indicar qué estructuras de datos utilizarás para resolver el problema.

(c) Explicar en palabras cómo resolverá el problema el algoritmo.

(d) Implementar el algoritmo en el lenguaje de la materia de manera precisa.

Datos:
n materias cursadas, no correlativas entre sí
día de examen d1,...,dn
días de estudio necesario c1,...,cn
solo se rinde un examen por día.

-> mayor cantidad de materias que puedes rendir?

(a) **El criterio de selección es rendir las materias que menos días de estudio lleven.**

(b) Estructura

```
type Materia = tuple
                id: nat
                c: nat
            end tuple
```

(c) Paso directo al algoritmo

```
fun rendir(m: Set of Materia) ret res: nat
      var aux_m: Set of Materia
      var w: Materia

      res := 0
      aux_m := copy_set(d)

      while not is_empty_set(m) do
            w := get(aux_m)
            w := elegir_materia(aux_m)
            elim_set(aux_m, w)
            res := res + 1 {-contador de las materias rendidas-}
            remover_materias(aux_m, w)
      od
      destroy(aux_m)
end fun

fun elegir_materia(m: set of Materia) ret res: Materia
```

```
        var m_aux: Set of Materia
        var w: Pasajero
        var min: nat

        m_aux := set_copu(m)
        min := +infinito

        while not is_empty_set(m_aux) do
              w := get(m_aux)
              if w.c < min then
                    res := w
                    min := res.c
              fi
        od
        destroy(m_aux)
end fun

proc remover_materias(in/out m: set of Materia, in g: materia)
        var m_aux: Set of Materia
        var w: Materia

        m_aux := copy(m)
        while not is_empty(m_aux) do
              w := get(m_aux)
              if w.c < g.c then
                    elim(m, w)
              fi
              elim(m_aux,w)
        od
        destroy(m_aux)
end proc
```

---

1. **(Algoritmos voraces) Un amigo te recomienda que entres en el mundo del trading de criptomonedas asegurándote que siempre vas a ganar, ya que tiene una bola de cristal que ve el futuro.**

   Conocés el valor actual $v_1^0, \ldots, v_n^0$ de n criptomonedas. La bola de cristal indica el valor que tendrá cada una de las criptomonedas durante los m días,siguientes. Es decir, los valores $v_1^1, \ldots, v_1^m$ que tendrá la criptomoneda 1 dentro de 1 día, ..., dentro de m días respectivamente; los valores $v_2^1, \ldots, v_2^m$ que tendrá la criptomoneda 2 dentro de 1 día, ..., dentro de m días respectivamente, etcétera. En general, v_i^j es el valor que tendrá la criptomoneda i dentro de j días.

   Con esta preciada información podés diseñar un algoritmo que calcule el máximo dinero posible a obtener al cabo de m días comprando y vendiendo criptomonedas, a partir de una suma inicial de dinero D.

   Se asume que siempre habrá suficiente cantidad de cada criptomoneda para comprar y que no se cobra comisión alguna por la compra y venta. También se asume que se pueden comprar fracciones de criptomonedas. Recordá que no siempre las criptomonedas incrementan su valor.

   Se pide lo siguiente:

   (a) Indicar de manera simple y concreta, cuál es el criterio de selección voraz para construir la solución?

   (b) Indicar qué estructuras de datos utilizarás para resolver el problema.

   (c) Explicar en palabras cómo resolverá el problema el algoritmo.

   (d) Implementar el algoritmo en el lenguaje de la materia de manera precisa.

```
type Cripto = tuple
            id: nat
            po: nat
            p1: nat
      end tuple
```

Criterio: compro una cripto por día, que se corresponda con la que más variación porcentual tuvo. (ver el caso en que todas bajen de precio)

---

3. Miguel Hernández decide invitar a sus amigos/as a un asado. Como vive en un departamento, planifica la realización del asado en la casa de su amiga Josefina Manresa. "Ché, ahí invité a toda la barra. Nos vemos hoy en tu casa para comer el asado", le dice. "Estás loco, che?" le responde Josefina, "tengo la casa hecha un kilombo. Me hubieras avisado." "No te preocupes, voy para allá y te doy una mano". Y sale para allá.

   Cuando Miguel llega a lo de su amiga encuentra a Josefina en el patio preparando el fuego y salando la carne, así que la saluda y se dispone a acomodar la casa antes de que llegue la gente. No puede creer el desorden que encuentra al entrar a la casa. Con un simple vistazo, estima que cada uno de los N ambientes de la casa le va a insumir un tiempo $t_1, t_2, \ldots, t_N$ acomodar y que la valoración que va a recibir por la tarea realizada es $v_1, v_2, \ldots, v_N$. También descubre que el tiempo total T de que dispone hasta que vengan los/as amigos/as no es suficiente para acomodar todos los ambientes, pero sabe que si acomoda parcialmente un ambiente, obtiene el reconocimiento proporcional. Es decir, a modo de ejemplo, si ordena $\frac{2}{5}$ del ambiente i, eso le lleva un tiempo $\frac{2}{5}t_i$ y le genera una valoración $\frac{2}{5}v_i$.

   Se desea escribir un algoritmo que encuentre la mayor valoración total a recibir por el trabajo de acomodar los ambientes realizado en el tiempo T.

N ambientes
tiempo t1,t2,...,tn acomodar
valoración v1,v2,...,vn
tiempo total T

algoritmo que encuentre la mayor valoración total a recibir por el trabajo de acomodar los ambientes realizado en el tiempo T.

```
type ambiente = tuple
            t: real
            v: real
      end tuple

fun asado(am: Set of Ambiente, T: float) ret res: real
      var am_aux: Set of Ambiente
      var w: Ambiente
      var t: real
      var max: real

      am_aux = copy_set(am)
      t := T
      res := 0
      max := -infinito

      while (not is_empty(am_aux)) do
            w := promedio(am_aux, time)
            if w.t <= time then
                  t := t - w.t
                  res = res + w.t
```

```
            else
                  res := res + t*w.v/w.t
                  t = 0
            fi
            elim(am_aux)
      od
      destroy(am_aux)
end fun
```

---

1. (Algoritmos voraces) Es principio de mes y tenés que ayudar a tu abuelo a pagar n facturas de servicios. El viejo es medio desconfiado y solo paga él mismo por ventanilla en efectivo, nada de transferencias o homebanking.

   Para cada factura i sabés qué día d_i va a llegar al domicilio y el día de vencimiento v_i. Obviamente no podés ir a pagar si no te ha llegado aún la factura al domicilio, y tenés que pagarlas todas antes del vencimiento (se puede pagar también el mismo día que vence). Como sos un excelente estudiante de Algoritmos 2, vas a diseñar un algoritmo que obtenga qué facturas se pagarán cada día, de manera tal que el abuelo vaya la menor cantidad de veces posible a la ventanilla de pago.

   Se pide lo siguiente:

   (a) Indicar de manera simple y concreta, cuál es el criterio de selección voraz para construir la solución?
   (b) Indicar qué estructuras de datos utilizarás para resolver el problema.
   (c) Explicar en palabras cómo resolverá el problema el algoritmo.
   (d) Implementar el algoritmo en el lenguaje de la materia de manera precisa.

n facturas de servicios por pagar
cada factura i se sabe en día di cuando llega a domicilio y el vencimiento vi.
Hay que pagar todas antes del vencimiento (se puede pagar el mismo día que vence)

```
Tipo de datos:
type Factura = tuple
            id: nat
            di: nat
            vi: nat
      end tuple
```

Criterio de selección: pagar las facturas que vencen primero.

```
fun factura(f: Set of Facturas, n: nat) ret res: List of Facturas
      var f_aux: Set of Facturas
      var w: Facturas
      var t: nat
      var dia: nat

      f_aux := set_copy(f)
      t := n

      while not is_empty(f_aux) && n>0 do
            w := facturas_para_pagar(f_aux)
            add(res, w.id)
            elim_set(res, w)
            n := n - 1
      od
```

```
end fun

fun facturas_para_pagar(f: Set of Facturas) ret Factura
      var f_aux: Set of Factura
      var w: Factura
      var min: nat

      min := +infinito
      f_aux := set_copy(f)

      while not is_empty(f_aux) do
            w := get(f_aux)
            if w.v < min then
                  min := w.v
                  ret := w
            fi
            set_elim(f_aux, w)
      od
      set_destroy(f_aux)
end fun
```

tiene que devolver List(nat, List of Factura) ver después

---

1. (Algoritmos voraces)

Dado un grafo dirigido G con costos no negativos en sus aristas, representado por su matriz de adyacencia, y un vértice v del mismo, el algoritmo de Dijkstra calcula, para cada vértice w del grafo G, el costo del camino de costo mínimo de v a w.

(a) De qué manera podés modificar el algoritmo de Dijkstra (llamémosle algoritmo de Artskjid) para que en lugar de calcular costos de caminos desde v, calcule costos de caminos hacia v. Es decir, para que dado G tal como se dijo, y dado un vértice v del mismo, calcule, para cada vértice w del grafo G, el costo del camino de costo mínimo de w a v. Escribí el algoritmo.

(b) ¿Cómo podrías utilizar los algoritmos de Dijkstra y de Artskjid para calcular, para cada vértice w de G el costo del camino de costo mínimo de ida y vuelta de v a w. Incluso si no resolviste el inciso anterior, podés intentar resolver éste utilizando ambos algoritmos.

a)
en lugar de calcular costos de caminos desde v, calcule costos de caminos hacia v. Es decir, para que dado un G tal como se dijo, y dado un vértice v del mismo, calcule, para cada vértice w del grafo G, el costo del camino de costo mínimo de w a v.

D[1] = minimo entre D[1] y D[3] + L[3,1]

```
fun artskjid(L: array[1..n,1..n] of Nat, v: Nat) ret D: array[1..n] of nat
      var c: nat
      var C: Set of nat
      for i:=1 to no do add(C,i) od
      elim(C,w)
      for j:=1 to n do D[j] := L[j,v] od
      while not is_empty_set(C) do
            c := elijo elementos c de C tal que D[c] sea minimo
            elim(C,c)
```

```
            for j in C D[j]:=min(D[j],D[c]+L[j,c] od
      od
end fun
```

b)

La funcion recibe un arreglo que representa el coste de ir desde el vertice del campo 1 al campo 2, recibe v que es un vertice (el de origen) y w otro vertice (el de destino). Se aplica Dijkstra apra obtener el arreglo con los valores de ir desde v hasta cada uno de los vertices, luego se aplica Artskjid para obtener los valores de cada uno de los vertices hacia v. Luego para calcular el coste de la ide y vuelta desde w se suman D[w] (coste de ir desde v hasta e) y A[w] (Coste desde ir desde w hacia v)

```
fun a(L: array[1..n,1..n] of nat, v: nat, w: nat) ret idavuelta:
                                                 array[1..n] of nat
      D := array[1..n] of nat
      A := array[1..n] of nat
      D := Dijkstra(L,v)
      A := Artskdid(L,v)
      for i:=1 to n do
            idavuelta[i] := D[i] + A[i]
      od
end fun
```