Ej1)

Demostrar que el algoritmo voraz para el problema de la mochila sin fragmentación no siempre obtiene la solución óptima. Para ello puede modificar el algoritmo visto en clase de manera de que no permita fragmentación y encontrar un ejemplo para el cual no halla la solución óptima.

```
type Objeto = tuple
                id : Nat
                value: Float
                weight: Float
              end tuple

Criterio: elegir el de mayor valor relativo (cociente entre el valor y el peso): dicho cociente
expresa el valor promedio de cada kg de ese objeto.

fun mochila(W: Float, C: Set of Objeto) ret L : List of Obj_Mochila
        var o_m : Objeto
        var resto : Float
        var C_aux : Set of Objeto
        S:= empty_list()
        C_aux:= set_copy(C)
        resto:= W
        do (resto > 0) →
                o_m := select_obj(C_aux)
                resto := resto - o_m.weight
                addl(S,o_m)
                elim(C_aux,o_m)
        od
        set_destroy(C_aux)
end fun

fun select_obj(C: Set of Objeto) ret r : Objeto
        var C_aux : Set of Objeto
        var o : Objeto
        var m : Float
        m := -∞
        C_aux := set_copy(C)
        do (not is_empty_set(C_aux)) →
                o := get(C_aux)
                if (o.value/o.weight > m) then
                        m := o.value/o.weight
                        r := o
                fi
                elim(C_aux,o)
        od
        set_destroy(C_aux)
end fun

Falla: puede que al elegir un objeto dejemos de lado otro de peor cociente, pero que aprovecha mejor
la capacidad.
Ejemplo:
Mochila de capacidad 10, objetos de valor 12, 11 y 8, y peso 6, 5 y 4.
El criterio elige al que pesa 5, ya que cada kg de ese objeto vale más de 2. Pero convenía elegir los
otros dos.
```

Ej2)

Considere el problema de dar cambio. Pruebe o dé un contraejemplo: si el valor de cada moneda es al menos el doble de la anterior, y la moneda de menor valor es 1, entonces el algoritmo voraz arroja siempre una solución óptima.

(1,9,200,401)

409 = 401 + 1 + 1 + 1 + 1 + 1 + 1 + 1 (solución del algoritmo)
409 = 200 + 200 + 9 (solución más óptima)

Para cada uno de los siguientes ejercicios:
• Describa cual es el criterio de selección.
• ¿En qué estructuras de datos representa la información del problema?
• Explique el algoritmo, es decir, los pasos a seguir para obtener el resultado. No se pide que ”lea” el algoritmo (”se define una variable x”, ”se declara un for”), si no que lo explique (”se recorre la lista/el arreglo/” o ”se elige de tal conjunto el que satisface...”).
• Escriba el algoritmo en el lenguaje de programación de la materia.

Ej3)

Se desea realizar un viaje en un automóvil con autonomía A (en kilómetros), desde la localidad l_0 hasta la localidad l_n pasando por las localidades $l_1, \ldots, l_{n-1}$ en ese orden. Se conoce cada distancia $d_i \leq A$ entre la localidad l_{i-1} y la localidad l_i (para $1 \leq i \leq n$), y se sabe que 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.

Suponer que inicialmente el tanque de combustible se encuentra vacío y que todas las estaciones de servicio cuentan con suficiente combustible.

- Describa cual es el criterio de selección:
    - Si el tanque está vacío, se llena el tanque en esa localidad o si con lo que tenemos no nos alcanza para llegar a la siguiente localidad, se llena el tanque en la localidad actual.

- ¿En qué estructuras de datos representa la información del problema?
    - Las localidades son una lista de pares que se asume ordenada, en donde el par contiene la localidad i-ésima, y la distancia en kilómetros a la siguiente localidad.
    - El resultado es una lista de localidades desde 0 a n, el largo de la lista indica las veces que se debe parar a cargar nafta, y cada uno de los elementos indica una localidad.
- Explicación:
    - Se recibe la lista con las localidades y se recorre para ir consultando las distancias entre las localidades para saber si es necesario llenar el tanque o no, esto último se indica con un valor de verdad true : lleno o alcanza, false: vacío o no alcanza. Cuando este sea verdadero, no agregamos la localidad al resultado final, cuando este sea falso, agregamos la localidad a la solución.

- Algoritmo:

```
Location = tuple
             place_id : nat
             distance : nat
           end tuple

fun trip (C : List of Location, A : nat) ret S : List of Location
       var l : Location
       var avail : nat
       var l_aux : List of Location
       l_aux := copy_list(C)
       avail := 0
       S := empty_list()
       do ( not is_empty(l_aux) ) then
              l := loc(l_aux,avail) {- última localidad a la cual llegamos -}
              addr(S,l)
              elimloc(l_aux, l) {- elimina de l_aux las localidades visitadas -}
              avail = A;
       od
       Destroy(c_aux)
end fun

fun loc (C : List of Location, A : nat) ret L : Location
       var l2 : Location
       var avail : nat
       var l_aux : List of Location
       var b : Bool
       l_aux := copy_list(C)
       avail := A
       b := true
       do ( not is_empty(l_aux) and b ) then
              l2 := head(l_aux)
              if (l2.distance ≤ avail) then
                  avail := avail - l2.distance
              else
                     L := l2
                     b := false
              fi
              tail(l_aux)
       od
       Destroy(l_aux)
end fun
```

```
proc elimloc ( in/out C : List of Location, in L : Location  )
```

```
        var l_aux : List of Location
        var l2 : Location
        var b : Bool
        l_aux := copy_list(C)
        b := true
        while ( not is_empty(l_aux) and b ) do
              if( head(l_aux) != L ) then
                    tail(L)
              else
                    b := false
              fi
              tail(l_aux)
        od
        Destroy(l_aux)
end proc
```

Ej4)

En numerosas oportunidades se ha observado que cientos de ballenas nadan juntas hacia la costa y quedan varadas en la playa sin poder moverse. Algunos sostienen que se debe a una pérdida de

orientación posiblemente causada por la contaminación sonora de los océanos que interferiría con su capacidad de inter-comunicacion. En estos casos los equipos de rescate realizan enormes esfuerzos para regresarlas al interior del mar y salvar sus vidas. Se encuentran n ballenas varadas en una playa y se conocen los tiempos s1, s2, . . . , sn que cada ballena es 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 rescate y que una ballena no muere mientras está siendo regresada mar adentro.

- Describa cual es el criterio de selección:
  Opciones:
  Rescatar aquella ballena que tenga menos tiempo de vida:
  Este criterio puede funcionar, pero no garantiza que todas las ballenas sean salvadas.

- ¿En qué estructuras de datos representa la información del problema?
  Proponemos un conjunto de pares, en donde cada par contiene el tipo whale

  type Whales : **tuple**
  N : nat *{- identifica a la ballena -}*
  time_left : float *{- indica el tiempo de vida restante -}*
  end tuple

  Set **of** Whales

  Luego, la solución será una lista de whales en orden:
  List **of** Whales

- Explicación:
  Recorremos todo el conjunto de ballenas y el algoritmo selecciona aquella ballena con menor tiempo de vida para salvarla. Luego, se eliminan aquellas ballenas que pudieron haber muerto mientras salvamos a la primera y así consecutivamente.

- Algoritmo:

```
fun blue_peace ( W : Set of Whales, t : float ) ret S : List of Whales
       var c : Whales
       var hora : Nat
       var w_aux : Set of Whales
       w_aux := set_copy(W)
       S:= empty_list()
       while ( not empty_set(w_aux) ) do
              c :=  minTime(w_aux)
              addr(S,c)
              elim(w_aux,c)
              hora := hora + t
              elimDeadWhales(w_aux, hora)
       od
       destroySet(w_aux)
end fun
proc elimDeadWhales(in/out w_aux : Set of Whales, in hora : Nat)
       var D : Set of Whales
       var b : Whales
       D := copy_set(w_aux)
```

```
        while not is_empty_set(D) do
                b := get(w_aux)
                if (b.time_left < hora) then
                        elim(w_aux,b)
                fi
                elim(D,b)
        od
end proc

fun  minTime ( W : Set of Whales) ret w : Whale
        var w_aux : Whales
        var w_aux : Set of Whales
        var min_aux : Nat
        w_aux := copy_set(W)
        min_aux := inf
        while ( not is_empty_set(w_aux) ) do
                b_aux := get(w_aux)
                if (b_aux.time_left < min_aux) then
                        min_aux :=  b_aux.time_left
                        b := b_aux
                fi
                elim(w_aux, b_aux)
        od
end proc
```

Ej5)

Sos el flamante dueño de un teléfono satelital, y se lo ofreces a tus n amigos para que lo lleven con ellos cuando salgan de vacaciones el próximo verano. Lamentablemente cada uno de ellos iría a un lugar diferente y en algunos casos, los períodos de viaje se superponen. Por lo tanto es imposible prestarle el teléfono a todos, pero quisieras prestarlo al mayor número de amigos posible. Suponiendo que conoces los días de partida y regreso (pi y ri respectivamente) de cada uno de tus amigos, ¿cuál es el criterio para determinar, en un momento dado, a quien conviene prestarle el equipo? Tener en cuenta que cuando alguien lo devuelve, recién a partir del día siguiente puede usarlo otro. Escribir un algoritmo voraz que solucione el problema.

```
type Friends = tuple
                        friend_id : nat
                        day_partida : nat
                        day_regreso : nat
                   end tuple
```

Criterios:

- Prestarle el equipo a aquellos amigos que salgan en los primeros días del verano e ir pasándolo al siguiente más próximo a salir. (Puede generar conflicto)
  F = ((Pedro,2,5),(Pablo,6,15),(Jose,7,8),(Juan,10,14))
  Con este criterio sólo podría prestar el teléfono a Pedro y a Pablo.
- Prestarle el equipo a aquellos amigos que menos dias salgan de vacaciones:
  F = ((Pedro,2,5),(Pablo,6,15),(Jose,7,8),(Juan,10,14))
  Con este criterio sólo podría prestar el teléfono a Pedro, Jose y Juan.
  No se si es el más óptimo pero da mejor resultado que el anterior.
  F = ((Pedro,2,5),(Pablo,3,7),(Jose,8,20),(Juan,19,21))

Con este criterio, le prestó el equipo a Pedro y a Juan. Cuando pude prestarlo a Pedro Jose y Juan.

- Prestarle el equipo a aquellos que salgan menos días tq no se superpongan con otros:
  F = ((Pedro,2,5),(Pablo,3,7),(Jose,8,20),(Juan,19,21)
  Con este criterio solo se lo prestaria a Juan

- Prestarle el equipo a aquellos que vuelven más antes de vacaciones:
  Creo que esto puede funcionar ya que, estamos priorizando a aquellos amigos que salen menos tiempo de vacaciones, así podemos asegurarnos de tener más disponible el teléfono. Y si hay un amigo que sale todo el verano, ni se lo prestamos ya. Aquellos que se superponen desgraciadamente van a ir sin equipo pero necesitamos el equipo lo más pronto posible para prestarle al siguiente, que vuelva antes que cualquier otro, para pasar al siguiente y así.

Ejemplo de ejecución:

F = ((Pedro,2,5),(Pablo,3,7),(Jose,8,9),(Juan,19,21),(Marti,2,27),(Manuel,1,6),(Fer,9,15))
S = ()
Dia 1
¿Quién vuelve antes? Pedro.
Desde el día 2 al 5, salió Pablo, Manuel y Marti
F = ((Jose,8,9),(Juan,19,21),(Fer,9,15))
S = (Pedro,)
Dia 6
¿Quién vuelve antes? Jose.
El dia que Jose volvió, se fue Fer
F = ((Juan,19,21))
S = (Pedro, Jose)
Dia 10
¿Quién vuelve antes? Juan.
F = ( )
S = (Pedro, Jose,Juan)
Dia 22

Algoritmo:

```
type Friends = tuple
               friend_id : nat
```

```
            day_partida : nat
            day_regreso : nat
                end tuple

type BF = tuple
            friend_id : nat
            day : nat
          end tuple

fun fono ( F : Set of Friends ) ret S : List of BF
{- copiamos F (Conjunto de amigos) -}
var f_aux : Set of Friends
var day : Nat
var f : Friends
var bf : BF
S := empty_list()
f_aux := copy_set(F)
day := 1
while ( not is_empty_set(f_aux) ) do
      f := choose_f(f_aux)      {- Elegimos aquel que vuelva antes -}
      elim(f,f_aux)             {- Lo eliminamos del conjunto -}
      bf.friend_id := f.friend_id  {- Armamos la solución actual -}
      bf.day := f.day_partida
      day := (f.day_regreso + 1){- Actualizamos el día en el cual podemos volver a prestarlo -}
      addr(S,bf)                {- Agregamos al resultado final -}
      elimGoneFriends(f_aux, day) {- Eliminamos aquellos amigos con viajes superpuestos -}
od
      destroySet(f_aux) {- Liberamos memoria -}
end fun

fun choose_f ( F : Set of Friends ) ret b : Friends
      var min_day : Nat
      var f_aux : Set of Friends
      var f : Friends
      f_aux := copy_set(F)
      min_day := +inf
      while ( not is_empty_set(f_aux) ) do
            f := get(f_aux)
            if( f.day_regreso < min_day ) then
                  min_day := f.day_regreso
                  b := f
            fi
            elim(f_aux, f)
      o
end fun



proc elimGoneFriends( in/out F : Set of Friends, in day : Nat)
      var f_aux : Set of Friends
      f_aux := copy_set(F)
      var bf : Friends
```

```
        while ( not is_empty_set(f_aux) ) do
                bf := get(f_aux)
                if ( bf.day_partida < day ) then
                        elim(F,bf)
                fi
                elim(f_aux,bf)
        od
end proc
```

Ej6)

Para obtener las mejores facturas y medialunas, es fundamental abrir el horno el menor número de veces posible. Por supuesto que no siempre es fácil ya que no hay que sacar nada del horno demasiado temprano, porque queda cruda la masa, ni demasiado tarde, porque se quema. En el horno se encuentran n piezas de panaderıa (facturas, medialunas, etc). Cada pieza i que se encuentra en el horno tiene un tiempo mínimo necesario de cocción ti y un tiempo máximo admisible de cocción Ti . Si se la extrae del horno antes de ti quedará cruda y si se la extrae después de Ti se quemaría. Asumiendo que abrir el horno y extraer piezas de ´el no insume tiempo, y que ti ≤ Ti para todo i ∈ {1, . . . , n}, ¿qué criterio utilizará un algoritmo voraz para extraer todas las piezas del horno en perfecto estado (ni crudas ni quemadas), abriendo el horno el menor número de veces posible? Implementarlo.

```
type Product = tuple
                product_id : Nat
                minT : float
                maxT : float
               end tuple

fun open_furnace ( P : Set of Products ) ret S : List of float
       var p_aux : Set of Product
       var p : Product
       var time : Float
       time := 0.0
       p_aux := copy_set(P)
       while ( not is_empy_set(p_aux) ) do
              p := minProduct(p_aux) {- Elijo aquel producto que se hornee primero -}
              elim(p_aux,p) {- Lo eliminó del conjunto-}
              time := p.maxT {- El tiempo ahora es el tiempo máximo del producto -}
              elimProductsInRange(p_aux, time) {- Eliminó los que se cocinaron antes de esos
                                                  minutos -}
              addr(S,time) {- guardo el tiempo máximo, es cuando saque el máximo de productos -}
       od
end fun
```

```
fun minProduct(P : Set of Product) ret p : Product
       var p_aux : Set of Product
       var min_time : Float
       var p2 : Product
```

```
        min := +inf
        p_aux := copy_set(P)
        while ( not is_empy_set(p_aux) ) do
                p2 := get(p_aux)
                if ( p2.maxT < min ) then
                        min := p2.maxT
                        p := p2
                fi
                elim(p_aux,p2)
        od
end fun

proc elimProductsInRange(in/out P : Set of Product, in time : Float)
        var p_aux : Set of Product
        var p2 : Product
        p_aux := copy_set(P)
        while ( not is_empy_set(p_aux) ) do
                p2 := get(p_aux)
                if ( p2.minT ≤ time ) then
                        elim(P,p2)
                fi
                elim(p_aux,p2)
        od
end proc
```

Ej7)

Un submarino averiado descansa en el fondo del océano con n sobrevivientes en su interior. Se conocen las cantidades c1, . . . , cn de oxıgeno que cada uno de ellos consume por minuto. El rescate de sobrevivientes se puede realizar de a uno por vez, y cada operación de rescate lleva t minutos.

(a) Escribir un algoritmo que determine el orden en que deben rescatarse los sobrevivientes para salvar al mayor número posible de ellos antes de que se agote el total C de oxígeno.

(b) Modificar la solución anterior suponiendo que por cada operación de rescate se puede llevar a la superficie a m sobrevivientes (con m ≤ n).

```
type Survivors :  tuple
                    survivor_id : nat
                    oxygen_left : float
                  end tuple
```

a)

```
fun barotrauma ( S : Set of Survivors, t : float ) ret R : List of Survivors
        var s : Survivors
        var hora : Nat
```

```
	var s_aux : Set of Survivors
	s_aux := set_copy(S)
	R := empty_list()
	while ( not empty_set(s_aux) ) do
		s :=  minOxygen(s_aux)
		addr(R,s)
		elim(s_aux,s)
		hora := hora + t
		elimDeadSurvivors(s_aux, hora)
	od
	destroySet(s_aux)
end fun

proc elimDeadSurvivors(in/out S : Set of Survivors, in hora : Nat)
	var s_aux : Set of Survivors
	var s : Survivors
	s_aux := copy_set(S)
	while not is_empty_set(s_aux) do
		s := get(s_aux)
		if (s.oxygen_left < hora) then
			elim(S,s)
		fi
		elim(s_aux,s)
	od
end proc

fun  minOxygen ( S : Set of Survivors) ret s : Survivors
	var s2 : Survivor
	var s_aux : Set of Survivors
	var min_aux : Nat
	s_aux := copy_set(S)
	min_aux := inf
	while ( not is_empty_set(s_aux) ) do
		s2 := get(s_aux)
		if (s.oxygen_left < min_aux) then
			min_aux :=  s2.oxygen_left
			s := s2
		fi
		elim(s_aux, s2)
	od
end proc
```

b)

```
fun barotrauma (S: Set of Survivors, t :Float, m: Nat) ret R: List of Set
	var s : Survivors
	var hora : Nat
	var s_aux : Set of Survivors
```

```
        var m_aux : Nat
        var Rset : Set of Survivors
        m_aux := m
        s_aux := set_copy(S)
        R := empty_list()
        while ( not empty_set(s_aux)) do
                while( i < m || not empty_set(s_aux)) do
                        s :=  minOxygen(s_aux)
                        add_set(Rset,s)
                        elim(s_aux,s)
                od
                addr(R,Rset)
                hora := hora + t
                elimDeadSurvivors(s_aux, hora)
                m_aux := m
                destroySet(Rset)
                Rset := empty_set()
        od
        destroySet(s_aux)
end fun

proc elimDeadSurvivors(in/out S : Set of Survivors, in hora : Nat)
        var s_aux : Set of Survivors
        var s : Survivors
        s_aux := copy_set(S)
        while not is_empty_set(s_aux) do
                s := get(s_aux)
                if (s.oxygen_left < hora) then
                        elim(S,s)
                fi
                elim(s_aux,s)
        od
end proc

fun  minOxygen ( S : Set of Survivors) ret s : Survivors
        var s2 : Survivor
        var s_aux : Set of Survivors
        var min_aux : Nat
        s_aux := copy_set(S)
        min_aux := inf
        while ( not is_empty_set(s_aux) ) do
                s2 := get(s_aux)
                if (s.oxygen_left < min_aux) then
                        min_aux :=  s2.oxygen_left
                        s := s2
                fi
                elim(s_aux, s2)
        od
end proc
```

Ej8)

Usted vive en la montaña, es invierno, y hace mucho frío. Son las 10 de la noche. Tiene una voraz estufa a leña y n troncos de distintas clases de madera. Todos los troncos son del mismo tama˜no y en la estufa entra solo uno por vez. Cada tronco i es capaz de irradiar una temperatura ki mientras se quema, y dura una cantidad ti de minutos encendido dentro de la estufa. Se requiere encontrar el

orden en que se utilizar´an la menor cantidad posible de troncos a quemar entre las 22 y las 12 hs del día siguiente, asegurando que entre las 22 y las 6 la estufa irradie constantemente una temperatura no menor a K1; y entre las 6 y las 12 am, una temperatura no menor a K2.

1) Entender enunciado
   - n troncos
   - cada tronco i, irradia temperatura k_i y dura un periodo t_i
   - Se pide encontrar el **orden** en que se utilizan la menor cantidad de troncos entre las 22hs y las 12hs del día siguiente tq:
     - Entre las 22 y las 6 la temperatura irradiada no es menor a K1
     - Entre las 6 y las 12, la temperatura irradiada no es menor a K2
     - Asumimos k1 ≥ k2

2) Criterio de selección:
   - Entre las 22 y las 6 elijo el tronco con tiempo t_i tq k_i es mayor a K1
   - Entre las 6 y las 12: idem, pero con k2

3) Estructuras de datos

```
type Tronco = tuple
                id : Nat
                calor : Float
                tiempo : Float
              end tuple

fun estufaVoraz (S : Set of Tronco, K1: Nat, K2: Nat) ret L : List of Tronco
```

```
Algoritmo:

{- PRE: Siempre hay suficientes troncos  -}
fun estufaVoraz (S : Set of  Tronco, K1: Float, K2: Float) ret L : List of Tronco
        var C: Set of Tronco
        var t: Tronco
        var h: Float
        C := copy_set(S,C)
        h := 0
        L := empty_list()
        while(h < 8) do
                if (h < 8) then
                        t := elegirTronco(C,K1) {- tronco con duración máxima con temp mayor a K1 -}
                else
                        t := elegirTronco(C,K2) {- tronco con duración máxima con temp mayor a K2 -}
                fi
                elim(C,t)
                addr(L,t)
                h := h + t.tiempo
        od
        destroy_set(C)
end fun
```

```
fun elegirTronco(C: Set of Troncos, K: Float) ret t: Tronco
        var B: Set of Tronco
        var max_tiempo : Float
        var t' : Tronco
```

```
        max_tiempo := -inf
        B := copy_set(C)
        while(not is_empty_set(B)) do
                t’ := get(B)
                if(t’.tiempo > max_tiempo and t’.calor ≥ K) then
                        max_tiempo := t’.tiempo
                        t := t’
                fi
                elim(B,t’)
        od
end fun
```

Ej9)

(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 02am (es decir que usted tomará 16 pintas) eligiendo en cada hora el bar que más le convenga.

1) Entender enunciado
   - n bares
   - cada bar tiene P_i (precio pinta de limonada) y H_i (horario de happy hour medido en horas a partir de las 18hs)
   - Se desea obtener el menor dinero posible que se puede gastar desde las 18hs hasta las 02am, se tomarán 16 pintas en total (8*2)
2) Criterio de selección:
   - Seleccionamos (dependiendo del horario en el que estemos) aquel bar que ofrezca el menor precio posible.
3) Estructuras de datos

```
type Bar = tuple
             bar_id : Nat
             precio : Float
             hora : Float
           end tuple
```

```
Algoritmo:

{- PRE: n ≥ 8 -}
fun overdose (L : Set of Bar) ret P : Float
        var l_aux : Set of Bar
        var b : Bar
        var h : Float
        var i : Nat
        i := 0
        h := 0
        l_aux := set_copy(L)
        while( i < n ) do
                b := elegirBar(l_aux,h)
                elim(l_aux, b)
                P := b.precio + P
                h := h + b.hora
                i++;
        od
        destroy_set(l_aux)
end fun
fun elegirBar( L : Set of Bar, h : Float) ret b : Bar
        var l_aux : Set of Bar
        var b_aux : Bar
        var min_p : Float
        min_p := +inf
        l_aux := set_copy(L)
```

```
        while( not is_empy_set(l_aux) ) do
                b_aux := get(l_aux)
                if(b_aux.precio < min_p and b_aux.hora == h+1 ) then
                        min_p := b_aux.precio
                        b := b_aux
                fi
                elim(l_aux,b_aux)
        od
        destroy_set(l_aux)
end fun
```