# 1

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

fun mochila(W: Float, C: Set of Objeto) ret L: List of Objeto
    var obj: Objeto
    var resto: Float
    var C_aux: Set of Objeto

    L := empty_list()
    C_aux := set_copy(C)
    resto := W

    do (resto > 0)
        obj := select_obj(C_aux)

        if obj.weight <= resto then
            resto := resto - obj.weight
            addl(L, obj)

        elim(C_aux, obj)

    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

        elim(C_aux,o)

    set_destroy(C_aux)
end fun
```

```
var BackpackWeight: Float
var Items: Set of Objeto
var A, B, C, D: Objeto

A := {id: 1, value: 30, weight: 7}  // 4.28pts
B := {id: 2, value: 20, weight: 6}  // 3.33pts
C := {id: 3, value: 15, weight: 4}  // 3.75pts

Items := {A, B, C}
BackpackWeight := 10
//  Mejor combinación posible: [B,C]

mochila(BackpackWeight, Items) // Devuelve: [A]
```

# 2

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.

- Supongamos
  - un conjunto `1, 4, 9`
  - un monto `12`
- La solución óptima es `3 monedas de 4`
- El algoritmo voraz devuelve `1 moneda de 9 y 3 de 1`, total de `4 monedas`

---

Para cada uno de los siguientes ejercicios:

- Describa cuál es el criterio de selección.
- ¿En qué estructuras de datos representará 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 elije de tal conjunto el que satisface...”).
- Escriba el algoritmo en el lenguaje de programación de la materia.

# 3

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 $l_1$, ..., l*{n−1}$ en ese orden. Se conoce cada distancia $d_i \le A$ entre la localidad $l*{i−1}$ y la localidad $l_i$ (para $1 \le i \le n$), 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.

> - Se selecciona la parada $l_M$ que cumpla $\max _M \sum _{i=n} ^m d_i < A$, donde $n$ es la parada siguiente a la última carga de combustible o a la parada 0.

- Datos:
  - Una tupla con un campo para la cantidad de cargas de combustibles y un campo para los lugares donde se harían dichas cargas.
  - Un arreglo con las distancias entre paradas, $d_i$.
- Se van sumando los $d_i$ hasta que el total supere a $A$, en ese entonces se suma una carga de combustible y se registra la parada $l_{i-1}$. Se reinicia la suma y se vuelve a empezar.

```
type GasChanges =   tuple
                        gasChanges: Nat
                        placesWhereChange: List of Nat
                    end tuple

fun calcutaGasChanges(A: Nat, stands: array[1..n] of Nat) ret gc: GasChanges
    var rest: Nat
    rest := 0
    gs.gasChanges := 0
    gs.placesWhereChange := empty()

    for i := 1 to n do
        rest := rest + stand[i]

        if rest > A then
            rest := 0
            gs.gasChanges := gs.gasChanges + 1
            addr(gs.placesWhereChange, i - 1)

end fun
```

# 4

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-comunicación. 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 $s_1, s_2, ..., s_n$ 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.

- Se selecciona la ballena que esté más pronto a morir pero que aguante también a ser rescatada.
- Datos:
  - Una lista con las ballenas que deben ser salvadas.
  - Un set con las ballenas.
  - El tiempo de rescate

> - Se van sumando los $t$ a medida que se van salvando las ballenas, y se selecciona la ballena más proxima a morir, cuyo $s_i$ sea mayor o igual a la suma acumulada de $t's$.

```
type Whale =   tuple
                    id: Nat
                    timeOfLife: Float
                end tuple

fun whalesToRescue(
    timeOfRescue: Float,
    whales: Set of Whale
) ret whalesToSave: List of Whale
    var timeSpent: Float
    var whales_copy: Set of Whale
    var whale: Whale

    timeSpent := 0
    whales_copy := set_copy(whales)
    whalesToSave := empty()

    while not is_empty_set(whales_copy) do
        whale := chooseWhale(whales_copy)

        if whale.timeOfLife >= timeSpent then
            addr(whalesToSave, whale)

            timeSpent := timeSpent + timeOfRescue

        elim(whales_copy, whale)

    set_destroy(whales_copy)
end fun

fun chooseWhale(whales: Set of Whale) ret whale: Whale
    var whales_copy : Set of Whale
    var whale_aux: Whale

    whales_copy := set_copy(whales)
    whale.timeOfLife := +∞

    while not is_empty_set(whales_copy) do
        whale_aux := get(whales_copy)

        if whale.timeOfLife > whale_aux.timeOfLife then
            whale := whale_aux

        elim(whales_copy, whale_aux)

    set_destroy(whales_aux)
end fun
```

# 5

Sos el flamante dueño de un teléfono satelital, y se lo ofrecés a tus $n$ amigos para que lo lleven con ellos cuando salgan de vacaciones el próximo verano. Lamentablemente cada uno de ellos irá 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 prestárselo al mayor número de amigos posible.

Suponiendo que conoces los días de partida y regreso ($p_i$ y $r_i$ 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.

> - Se selecciona el amigo cuyo viaje con la menor cantidad de viajes superpuestos.
> - Datos:
>   - Un set con todos los amigos y la información de sus viajes.
>   - Un set con todos los amigos a los que se les prestará el celular.
> - Se selecciona y se guarda el mejor viaje, cuyas fechas no se encuentren dentro del registro de fechas ocupadas; se actualiza el registro de fechas que ya no están disponibles; Se repite el procedimiento hasta no poder seleccionar más viajes.

```
type Travel =   tuple
                    friend: String
                    start: Date
                    end: Date
                end tuple

fun friendsToLend(travels: Set of Travel) ret friends: List of String
    var travels_copy: Set of Travel
    var travelsApproved: List of Travel
    var travel: Travel

    travels_copy := set_copy(travels)
    friends := empty()
    travelsApproved := empty()

    travel := chooseTravel(travels_copy)
    addl(friends, travel.friend)
    addl(travelsApproved, travel)
    elim(travels_copy, travel)

    while not is_empty_set(travels_copy) && travel != null do
        travel := chooseTravel(travels_copy)

        if travel != null then
            addr(friends, travel.friend)
            addl(travelsApproved, travel)

        elim(travels_copy, travel)

    destroy(travelsApproved)
    set_destroy(travels_copy)
end fun
```

```
fun superposedTravels(
    travelA: Travel,
    travelB: Travel
) ret b: bool
    var startsThen, endsThen: bool

    startsThen := travelA.start <= travelB.end + 1 &&
        travelA.start >= travelB.start

    endsThen := travelA.end <= travelB.end + 1 &&
        travelA.end >= travelB.start

    b := startsThen || endsThen
end fun

fun validTravel(
    travel: Travel,
    travelsApproved: List of Travel
) ret b: bool
    var travels_copy: List of Travel
    var travel_aux: Travel

    b := true
    travels_copy := copy_list(travelsApproved)

    while not is_empty(travels_copy) && b do
        travel_aux := head(travels_copy)
        tail(travels_copy)

        b := not superposedTravels(travel_aux, travel)

    destroy(travels_copy)
end fun

fun chooseTravel(
    travels: Set of Travel,
    travelsApproved: List of Travel
) ret travel: Travel
    var travels_copy, travels_aux: Set of Travel
    var travel_aux, travel_rec: Travel
    var superposedTravels: Nat
    var minSuperposedTravels: Nat

    travels_copy := set_copy(travels)
    minSuperposedTravels := +∞

    while not is_empty_set(travels_copy) do
        travel_aux := get(travels_copy)
        elim(travels_copy, travel_aux)

        if validTravel(travel_aux, travelsApproved) then
            travel_rec := set_copy(travels_copy)

            while not is_empty_set(travels_aux) do
                travel_rec := get(travels_copy)
```

```
                elim(travels_copy, travel_rec)

                if superposedTravels(travel_rec, travel_aux) then
                    superposedTravels := superposedTravels + 1

            if minSuperposedTravels > superposedTravels then
                minSuperposedTravels := superposedTravels
                travel := travel_aux

    set_destroy(travels_aux)
end fun
```

# 6

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 panadería (facturas, medialunas, etc). Cada pieza $i$ que se encuentra en el horno tiene un tiempo mínimo necesario de cocción $t_i$ y un tiempo máximo admisible de cocción $T_i$ .
Si se la extrae del horno antes de $t_i$ quedará cruda y si se la extrae después de $T_i$ se quemará.

Asumiendo que abrir el horno y extraer piezas de él no insume tiempo, y que $t_i \le T_i$ para todo $i \in {1, ..., n}$, ¿qué criterio utilizaría 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.

> - Se selecciona un tiempo que superponga la mayor cantidad de rangos $(t_i, T_i)$ de cocción.
> - Datos:
>   - Un set con todos los tiempos de cocción.
>   - Cantidad de piezas de panadería.
> - Se selecciona el mejor tiempo y se eliminan todas las piezas de panadería cuyo rango de tiempo $(t_i, T_i)$ contenga dicho tiempo; se repite el procedimiento hasta no haber más piezas de panadería.

```
type PieceOfBakery =    tuple
                            min: Nat
                            max: Nat
                        end tuple

fun qOpenOven(pieces: Set of PieceOfBakery) ret quantity: Nat
    var pieces_copy, times: Set of PieceOfBakery

    pieces_copy := set_copy(pieces)
    quantity := 0

    while not is_empty_set(pieces_copy) do
        times := chooseTimes(pieces_copy)
        removeAsPosible(pieces_copy, times)
        quantity := quantity + 1
```

```
    set_destroy(pieces_copy)
end fun

fun superposeTime(
    timeA: Nat,
    timeB: Travel
) ret b: bool
    b := timeA <= timeB.max && timeA >= timeB.min
end fun

fun superposedTimes(
    timeA: Travel,
    timeB: Travel
) ret b: bool
    var startsThen, endsThen: bool

    startsThen := superposeTime(timeA.min, timeB)
    endsThen := superposeTime(timeA.max, timeB)

    b := startsThen || endsThen
end fun


proc removeAsPosible(
    in/out pieces: Set of PieceOfBakery
    in/out subPieces: Set of PieceOfBakery
)
    var times_copy, times_aux, times: Set of PieceOfBakery
    var time, time_aux: PieceOfBakery
    var maxSuperposedTimes, superposedTimes: Nat

    time := get(subPieces)
    maxSuperposedTimes := -∞
    elim(subPieces, time)
    elim(pieces, time)

    for i := time.min to time.max do
        superposedTimes := 0
        times_copy := set_copy(subPieces)
        times_aux := empty_set()

        while not is_empty_set(times_copy) do
            time_aux := get(times_copy)
            elim(times_copy, time_aux)

            if superposeTime(i, time_aux) then
                add(times_aux, time_aux)
                superposedTimes := superposedTimes + 1

        set_destroy(times_copy)

        if maxSuperposedTimes < superposedTimes then
            maxSuperposedTimes := superposedTimes
            set_destroy(times)
```

```
            times := time_aux
        else
            set_destroy(times_aux)

    set_destroy(subPieces)

    while not is_empty_set(times) do
        time_aux := get(times)

        elim(times_copy, time_aux)
        elim(pieces, time_aux)

    set_destroy(times)
end proc

fun chooseTimes(
    pieces: Set of PieceOfBakery
) ret times: Set of PieceOfBakery
    var pieces_copy, pieces_aux, times_aux: Set of PieceOfBakery
    var time_aux, time_rec: PieceOfBakery
    var maxSuperposedTimes, superposedTimes: Nat

    pieces_copy := set_copy(pieces)
    maxSuperposedTimes := -∞

    while not is_empty_set(pieces_copy) do
        time_aux := get(pieces_copy)
        elim(pieces_copy, time_aux)

        superposedTimes := 0
        pieces_aux := set_copy(pieces_copy)
        times_aux := empty_set()
        add(times_aux, time_aux)

        while not is_empty_set(pieces_aux) do
            time_rec := get(pieces_aux)
            elim(pieces_aux, time_rec)

            if superposedTimes(time_rec, time_aux) then
                superposedTimes := superposedTimes + 1
                add(times_aux, time_rec)

        set_destroy(pieces_aux)

        if maxSuperposedTimes < superposedTimes then
            maxSuperposedTimes := superposedTimes
            set_destroy(times)
            times := time_aux
        else
            set_destroy(times_aux)

    set_destroy(pieces_copy)
end fun
```

# 7

Un submarino averiado descansa en el fondo del océano con $n$ sobrevivientes en su interior. Se conocen las cantidades $c_1, ..., c_n$ de oxígeno 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.

1. 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.
2. Modificar la solución anterior suponiendo que por cada operación de rescate se puede llevar a la superficie a $m$ sobrevivientes (con $m \le n$).

> - Se selecciona el sobreviviente que mayor cantidad de oxígeno consuma y que aguante a ser rescatado.
> - Datos:
>   - Cantidad de sobrevivientes.
>   - Cantidades de oxígeno que consumen los sobrevivientes.
>   - Cantidad de oxígeno total
>   - Tiempo de rescate
> - Se selecciona un superviviente para ser rescatado; se suma todo el oxígeno consumido en $t$, actualizando en valor de $C$ y quitando al seleccionado del grupo de sobrevivientes a rescatar; Se repite el proceso con el nuevo $C$ y el nuevo grupo de sobrevivientes, hasta que ya no quede oxígeno o los que queden no aguanten al rescate.

# 1

```
type Survivor = tuple
                    id: Nat
                    o: Nat
                end tuple

type DataOfRescue = tuple
                        survivor: Survivor
                        o: Nat
                    end tuple

fun orderOfRescue(
    t: Nat,
    C: Nat,
    survivors: Set of Survivor
) ret survivorsToRescue: Queue of Survivor
    var survivors_copy: Set of Survivor
    var data: DataOfRescue

    survivors_copy := set_copy(survivors)

    while not is_empty_set(survivors_copy) && C >= 0 do
        data := selectSurvivor(survivors_copy)

        C := C - data.o * t
```

```
        if C >= 0 then
            enqueue(survivorsToRescue, data.survivor)

        elim(survivors_copy, data.survivor)

    set_destroy(survivors_copy)
end fun

fun selectSurvivor(
    timeOfRescue: Nat,
    totalOxigen: Nat,
    survivors: Set of Survivor
) ret data: DataOfRescue
    var survivors_copy: Set of PieceOfBakery
    var survivor: Survivor

    survivors_copy := set_copy(survivors)
    data.o := 0
    survivor.o := -∞
    data.survivor := survivor

    while not is_empty_set(survivors_copy) do
        survivor := get(survivors_copy)
        elim(survivors_copy, survivor)

        data.o := data.o + survivor.o

        if survivor.o > data.survivor.o then
            data.survivor := survivor

    set_destroy(survivors_copy)
end fun
```

## 2

```
type Survivor = tuple
                    id: Nat
                    o: Nat
                end tuple

type DataOfRescue = tuple
                        survivors: Set of Survivor
                        o: Nat
                    end tuple

fun orderOfRescue(
    t: Nat,
    C: Nat,
    m: Nat,
    survivors: Set of Survivor
) ret survivorsToRescue: Queue of Survivor
    var survivors_copy: Set of Survivor
    var survivor, survivor_aux: Survivor
    var data: DataOfRescue
```

```
    survivors_copy := set_copy(survivors)

    while not is_empty_set(survivors_copy) && C >= 0 do
        data := selectSurvivor(survivors_copy)

        C := C - data.o * t

        if C >= 0 then
            while not is_empty_set(data.survivors) do
                survivor := get(data.survivors)
                elim(data.survivors, survivor)
                elim(survivors_copy, survivor)
                enqueue(survivorsToRescue, survivor)
        else
            while not is_empty_set(data.survivors) do
                survivor := get(data.survivors)
                elim(data.survivors, survivor)
                elim(survivors_copy, survivor)

        set_destroy(data.survivors)

    set_destroy(survivors_copy)
end fun

fun selectSurvivor(
    m: Nat,
    survivors: Set of Survivor
) ret data: DataOfRescue
    var survivors_copy, survivors_aux: Set of PieceOfBakery
    var survivor, survivor_aux: Survivor
    var inListToRescue: Nat

    survivors_copy := set_copy(survivors)
    data.o := 0
    inListToRescue := 0
    data.survivor := empty_set()

    while not is_empty_set(survivors_copy) do
        survivor := get(survivors_copy)
        elim(survivors_copy, survivor)

        data.o := data.o + survivor.o

        survivors_aux := set_copy(data.survivors)

        if n = m then
            while not is_empty_set(survivors_aux) do
                survivor_aux := get(survivors_aux)
                elim(survivors_aux, survivor_aux)

                if survivor.o > survivor_aux.o then
                    elim(data.survivors, survivor_aux)
                    add(data.survivors, survivor)
        else
```

```
            add(data.survivors, survivor)
            n := n + 1

        set_destroy(survivors_aux)

    set_destroy(survivors_copy)
end fun
```

# 8

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 tamaño y en la estufa entra solo uno por vez. Cada tronco $i$ es capaz de irradiar una temperatura $k_i$ mientras se quema, y dura una cantidad $t_i$ de minutos encendido dentro de la estufa. Se requiere encontrar el orden en que
se utilizarán 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 $K2$.

> - Se selecciona el tronco que más dure pero cuyo $k$ sea mayor al $K$ correspondiente.
> - Datos:
>   - Hora de inicio.
>   - Hora de fin.
>   - Hora de cambio de temperaura.
>   - Temperatura 1.
>   - Temperatura 2.
>   - Troncos.
>   - Tiempo por tronco
>   - Temperatura por tronco
> - Se divide el problema en dos subproblemas, uno para la temperatura más alta y otro para la más baja; se eligen todos los troncos que generan una temperatura mayor o igual a la pedida; de los seleccionados, se eligen los troncos que más duren, hasta cumplir con el tiempo pedido. El procedimiento es el mismo para los dos subcasos, con la diferencia de que primero se realiza el caso de mayor temperatura (ya que este puede llegar a ser más escaso de troncos que el de menor temperatura).

```
type Log =  tuple
                id: Nat
                t: Nat
                k: Nat
            end tuple

fun orderOfLogs(
    K1: Nat,
    K2: Nat,
    logs: Set of Log
) ret logsToBurn: List of Log
    var logs_copy, logs_aux: Set of Log
    var logsMax, logsMin: List of Log
    var log, log_aux: Log
    var maxK, minK, fstT, sndT: Nat
```

```
    if K1 >= K2 then
        maxK := K1
        minK := K2
        fstT := 8
        sndT := 6
    else
        maxK := K2
        minK := K1
        fstT := 6
        sndT := 8

    logsMax := empty()
    logsMin := empty()
    logs_copy := set_copy(logs)
    logs_aux := selectLogs(maxK, logs_copy)

    while not is_empty_set(logs_aux) && fstT > 0 do
        log := selectLog(logs_aux)
        fstT := fstT - log.t

        elim(logs_aux, log)
        elim(logs_copy, log)

        addr(logsMax, log)

    set_destroy(logs_aux)

    logs_aux := selectLogs(minK, logs_copy)

    while not is_empty_set(logs_aux) && sndT > 0 do
        log := selectLog(logs_aux)
        sndT := sndT - log.t

        elim(logs_aux, log)

        addr(logsMin, log)

    if maxK = K1 then
        logsToBurn := logsMax
        concat(logsToBurn, logsMin)
        destroy(logsMin)
    else
        logsToBurn := logsMin
        concat(logsToBurn, logsMax)
        destroy(logsMax)

    set_destroy(logs_aux)
    set_destroy(logs_copy)
end fun

fun selectLogs(
    K: Nat,
    logs: Set of Log
) ret logsToBurn: Set of Log
```

```
    var logs_copy: Set of Log
    var log: Log

    logs_copy := set_copy(logs)
    logsToBurn := empty_set()

    while not is_empty_set(logs_copy) do
        log := get(logs_copy)
        elim(logs_copy, log)

        if log.k >= K then
            add(logsToBurn, log)

    set_destroy(logs_copy)
end fun

fun selectLog(
    logs: Set of Log
) ret log: Log
    var logs_copy: Set of Log
    var log_aux: Log

    logs_copy := set_copy(logs)
    log.t := -∞

    while not is_empty_set(logs_copy) do
        log_aux := get(logs_copy)

        if log_aux.t > log.t then
            log := log_aux

        elim(logs_copy, log_aux)

    set_destroy(logs_copy)
end fun
```

# 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.

> - Se selecciona el bar más barato para una hora dada, considerando que si en un bar hay HH a dicha hora, el precio a considerar es el del descuento.
> - Datos:
>   - Pintas por hora.
>   - Pintas en total.
>   - Descuento HH.

    - Bares.
    - Horarios de HH por bar.
    - Precios por bar.
- Recorro la lista de bares, evaluando sus precios, considerando de que si la hora actual es hora de HH en tal bar, su precio es el 50%; Elijo el precio más barato y lo sumo multiplicado por dos; Si aún no son las 02am, vuelvo a implementar con la hora siguiente.

```
type Bar =  tuple
                id: Nat
                p: Float
                h: Nat
            end tuple

fun minMoney(
    bars: Set of Bar
) ret money: Float
    money := 0

    for i := 0 to 8 do
        money := money + selectPint(i, bars) * 2

end fun

fun selectPint(
    hour: Nat
    bars: Set of Bar
) ret pint: Float
    var bars_copy: Set of Bar
    var bar: Bar
    var pint_aux: Float

    bars_copy := set_copy(bars)
    pint := +∞

    while not is_empty_set(bars_copy) do
        bar := get(bars_copy)

        if bar.h >= hour then
            pint_aux := bar.p * 0.5
        else
            pint_aux := bar.p

        if pint > pint_aux then
            pint := pint_aux

        elim(bars_copy, bar)

    set_destroy(bars_copy)
end fun
```