PRÁCTICO 3 - PARTE 1 : ALGORITMOS VORACES

EJERCICIO 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 que no permita fragmentación y encontrar un ejemplo para el cual no haya una solución óptima.

Recordemos antes en qué consistía el problema de la mochila:
Tenemos una mochila de capacidad W.
Tenemos n objetos de valor $v_1, v_2, \dots, v_n$ y peso $w_1, w_2, \dots, w_n$.
Se quiere encontrar la mejor selección de objetos para llevar en la mochila, entendiendo por mejor selección aquella que totaliza el **mayor valor posible** sin que su peso exceda la capacidad W de la mochila.
Para que el problema no sea trivial, asumimos que la suma de los pesos de los n objetos excede la capacidad de la mochila, obligándonos entonces a seleccionar cuáles cargar en ella.

Ahora, lo que nos pide este ejercicio, es que supongamos también que no se permite la fragmentación de los objetos.

El criterio de selección sigue siendo el mismo que el visto en clase: elegir el de mayor valor relativo (cociente entre valor y peso). Dicho cociente expresa el valor promedio de cada kg de ese objeto.

Para demostrar que el algoritmo voraz para el problema de la mochila **sin fragmentación** NO siempre obtiene la solución óptima, doy un contraejemplo:

La falla de este algoritmo se basa en que al elegir el de mayor valor relativo, lo que puede terminar ocurriendo es que se descartan otros objetos de apenas menos valor relativo pero que aprovechaban mejor la capacidad de la mochila. Por ejemplo:

Supongamos que mi mochila tiene una capacidad de 15 kilogramos.
Supongamos también que tengo lo siguientes objetos (los nombro por comodidad):

Un libro de valor 20 y peso 10 (valor relativo = 2).
Un parlante de valor 15 y peso 8 (valor relativo = 1,875).
Una cartuchera 13 y peso 7 (valor relativo = 1,857).

Ahora bien, si se elige el de mayor valor relativo primero, se elegiría el libro pero en la mochila no me va a quedar más capacidad para meter ninguno de los otros dos objetos, pues ya voy a haber ocupado 10 kilogramos de los 15 disponibles, y los otros dos objetos pesan 8 y 7 kilogramos.

En cambio, no es difícil de ver que si se elegían los otros dos objetos, de apenas menor valor relativo que el libro, iba a poder meter ambos objetos y de esta manera aprovechar mejor la capacidad disponible en la mochila.

Finalmente, con este contraejemplo, he demostrado que el algoritmo voraz para el problema de la mochila **sin fragmentación** no siempre obtiene la solución óptima.

EJERCICIO 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.

Recordemos en qué consistía el problema de dar cambio:
Tenemos una cantidad infinita de monedas de distintas denominaciones.
Se desea pagar un cierto monto de manera exacta con la menor cantidad de monedas posible.

Recordemos también el criterio de selección: seleccionar una moneda de la mayor denominación posible que no exceda el monto a pagar, y utilizar exactamente el mismo algoritmo para el importe remanente.

En este caso, según lo que dice el enunciado del ejercicio, tenemos que **el valor de cada moneda es al menos el doble de la anterior**, y la moneda de menor valor es 1.

Trato de hallar un contraejemplo:
Supongamos que tengo monedas de denominaciones 1, 6 y 13 (notar que estas denominaciones cumplen con lo que pide el enunciado), y el monto que quiero pagar de manera exacta con la menor cantidad de monedas posibles es 18.

Lo que hará el algoritmo es primero elegir una moneda de 13, y luego 5 monedas de 1, teniendo así un total de 6 monedas.
Sin embargo, la solución óptima era pagar con 3 monedas de 6.

Como encontré un contraejemplo, se puede afirmar que 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 NO SIEMPRE arroja una solución óptima.

EJERCICIO 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 $l_1, \dots, l_{n-1}$ 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 localidad 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.

1. Analicemos el enunciado:

- El automóvil tiene una autonomía A medida en kilómetros mayor o igual a cualquiera de las distancias que hay entre las localidades.
- Se quiere ir desde la localidad l_o hasta la localidad l_n, pasando por todas las localidades intermedias, es decir por $l_1, \dots, l_{n-1}$, en orden.
- Se conoce la distancia entre localidad y localidad, para todas ellas.
- En cada localidad hay una estación de combustible.
- Inicialmente, el tanque de combustible se encuentra vacío.

Se pide computar **el menor número de veces que es necesario cargar combustible** para realizar el viaje, y las localidades en donde se realizaría la carga.

2. Criterio de selección

Elijo llegar a la localidad hasta donde me dé mi autonomía. Una vez allí, cargo combustible (lleno el tanque) y aplico el mismo criterio de selección.

Para clarificar, hago un ejemplo:

Supongamos que quiero ir desde Córdoba hasta Río Cuarto, y quiero pasar por las localidades de Alta Gracia, Anisacate, Villa General Belgrano y Santa Rosa de Calamuchita, en ese orden.

Sé que:

Entre Córdoba y Alta Gracia, hay 30 kilómetros.

Entre Alta Gracia y Anisacate, hay 11 kilómetros.

Entre Anisacate y Villa General Belgrano, hay 40 kilómetros.

Entre Villa General Belgrano y Santa Rosa de Calamuchita, hay 25 kilómetros.

Entre Santa Rosa de Calamuchita y Río Cuarto, hay 37 kilómetros.

La autonomía de mi automóvil es de A = 42 kilómetros.

De acuerdo al criterio de selección escogido, la selección se haría de la siguiente forma:

Veamos que con mi autonomía, podría ir desde Córdoba hasta hasta Anisacate sin tener que parar a cargar combustible en Alta Gracia, puesto que entre Córdoba y Anisacate hay una distancia de 41 kilómetros y la autonomía es de 42 kilómetros. Pero ya si quisiera seguir hasta Villa General Belgrano, necesariamente debería cargar combustible en Anisacate.

Se aplica lo mismo hasta llegar a Río Cuarto.

3. Estructuras de datos

Creo un nuevo tipo de dato para representar a cada localidad, con su nombre y la distancia que hay entre esta y la anterior:

```
type Locality = tuple
                    name : string
                    distance : nat
                end tuple
```

4. Prototipo del algoritmo:

Cabe notar que aquí IMPORTA EL ORDEN en el que estén las localidades de entrada, entonces uso listas y NO CONJUNTOS.

OBSERVACIÓN: los profes dijeron que con solo devolver la lista con las ciudades en donde se cargó nafta basta.

```
fun less_fueling(localities:List of Locality, A:Nat) ret stops: List of Locality
```

5. Implementación de algoritmo:

```
fun less_fueling(localities:List of Locality, A:Nat) ret stops: List of string
    var l_aux : List of Locality
    var locality : Locality
```

```
    l_aux := copy_list(localities)

    {- como el tanque inicialmente está vacío, tengo que cargar combustible en la
    localidad desde donde parto para poder arrancar el viaje -}
    locality := head(l_aux)
    addr(stops, locality.name)

    while not is_empty_list(l_aux) do
        {- hallo la ciudad en donde se realiza la próxima carga -}
        locality := chooseLocality(localities, A)

        {- en la próxima iteración, la ciudad en donde cargué nafta por última vez
        es la ciudad desde donde tengo que partir. Entonces, esta última ciudad,
        tiene que pasar a ser el primer elemento de la lista en mi próxima
        iteración -}
        while head(l_aux) != locality do
            tail(l_aux)
        od

        addr(stops, locality.name)
    od

    destroy_list(l_aux)
end fun



fun chooseLocality (L : List of Locality, A: nat) ret loc : Locality
    var L_aux : List of Locality
    var sum_dist : nat

    sum_dist := 0

    while sum_dist + head(l_aux).distance <= A && not is_empty_list(l_aux) do
        sum_dist := sum_dist + head(l_aux).distance
        loc := head(l_aux)
        tail(l_aux)
    od

    destroy_list(l_aux)
end fun
```

EJERCICIO 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 interferirá con su

capacidad de intercomunicación. 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, que hay un único equipo de rescate y que una ballena no muere mientras está siendo regresada mar adentro.

1. Analicemos el enunciado:

- Hay n ballenas
- Tiempos de vida de cada ballena: s1, s2, … , sn
- Salvar una ballena requiere un tiempo constante t
- La ballena NO MUERE mientras está siendo rescatada

Se pide salvar a la **mayor cantidad de ballenas posible**, y dar el orden en que lo hacemos.

2. Criterio de selección:
Elijo en cada momento la ballena a la que le quede **menos tiempo de vida**. De esta forma, todavía voy a tener “margen” para salvar a las ballenas que más tiempo de vida les queda.

Por ejemplo:

n = 7; tiempos de vida: 3, 5, 11, 21, 27, 33, 44, salvar requiere tiempo t = 10.

Simulemos solucionar este ejemplo, recordando que una ballena NO MUERE mientras está siendo rescatada.

Paso 1:
Salvo a la ballena 1 (con s1 = 3)
Como tardé 10 minutos, se murió la ballena 2 (con s2 = 5) y el resto de las ballenas me quedan con tiempos de vida 1, 11, 17, 23, 34.
Estoy en el minuto 10.
Paso 2:
Salvo a la ballena 3 (con s3 = 11, que es la que actualmente tiene tiempo de vida 1)
Como tardé 10 minutos, el resto de las ballenas me quedan con tiempos de vida 1, 7, 13, 24.
Estoy en el minuto 20.
Paso 3:
Salvo a la ballena 4 (con s4 = 21, que es la ballena que actualmente tiene tiempo de vida 1)
Como tardé 10 minutos, se murió la ballena 5 (con s2 = 27) y el resto de las ballenas me quedan con tiempos de vida: 3, 14
Estoy en el minuto 30.
Paso 4:
Salvo a la ballena 6 (con s6 = 33, que es la que actualmente tiene tiempo de vida 3)
Como tardé 10 minutos, la última ballena me queda con tiempo de vida: 4;
Estoy en el minuto 40.
Paso 5:
Salvo a la ballena 7 (con s7 = 44)
No quedan más ballenas.

Finalmente, salvé a 5 ballenas (a la 1, 3, 4, 6 y 7).

No es difícil de ver que si hubiéramos usado el criterio inverso, es decir salvar a la ballena que le queda más tiempo de vida, habríamos salvado solo a 3 ballenas (a la 5, 6 y 7).

3. Estructuras de datos:
Creo un nuevo tipo de dato para representar a una ballena:

```
type Whale = tuple
                id: Nat
                timeLeft : Nat
             end tuple
```

4. Prototipo del algoritmo:

```
fun saveWhales (whales: Set of Whale, time: Nat) ret saved: List of Whale
```

{- Observar que si devolviera un conjunto de ballenas (Set of Whale), no tengo forma de saber en qué orden salvé a las ballenas (recordar que en los conjuntos NO importa el orden), en cambio en las listas sí. -}

5. Implementar el algoritmo:

```
fun saveWhales (whales: Set of Whale, time: Nat) ret saved: List of Whale
     var aliveWhales : Set of Whale
     var currentTime : Nat
     var whale : Whale

     aliveWhales = set_copy(whales)
     currentTime := 0
     saved := empty_list()

     while not is_empty_set(aliveWhales) do
          {- elijo la ballena candidata según criterio de selección -}
          whale := selectWhale(aliveWhales)

          {- agrego la candidata elegida a la lista solución -}
          addr(saved, whale)

          {- elimino la ballena rescatada de las aún vivas no rescatadas -}
          elim_set(aliveWhales, whale)

          {- actualizo el reloj -}
          currentTime := currentTime + time

          {- elimino las ballenas cuyo tiempo de vida se acabó después del rescate
          de una ballena -}
          elimDead(aliveWhales, currentTime)
     od
```

```
    destroy_set(aliveWhales)
end fun

fun selectWhale (W: Set of Whale) ret minTimeWhale : Whale
    var W0 : Set of T
    aux_whale : Whale

    aux_whale := get(W0)
    minTimeWhale.timeLeft := aux_whale.timeLeft

    while not is_empty_set(W0) do
        aux_whale := get(W0)
        if aux_whale.timeLeft < minTimeWhale.timeLeft then
            minTimeWhale := aux_whale
        fi
        elim(W0, aux_whale)
    od
    destroy_set(W0)
end fun


proc elimDead (in/out W: Set of Whale, time: nat)
    var W0 : Set of Whale
    var whale : Whale

    W0 := copy_set(W)
    while not is_empty_set(W0) do
        whale := get(W0)
        if whale.timeLeft < time then
            elim_set(W, whale)
        fi
        elim(W0, whale)
    od
    destroy_set(W0)
end proc
```

EJERCICIO 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.

1. Analizo el enunciado:

- Hay n cantidad de amigos
- Se conocen los días de partida y de regreso de cada uno de los amigos.
- Cuando alguien devuelve el teléfono, recién a partir del día siguiente puede usarlo otro.

Se desea prestarle el teléfono al **mayor número de amigos posible**.

2. Criterio de selección:

Elijo en cada momento al amigo cuya fecha de regreso es la más cercana a la fecha actual. De esta forma, voy a tener más chances de prestárselo a los amigos que se van después.

3. Estructuras de datos:

Creo un nuevo tipo de dato para representar a un amigo (para las fechas uso naturales):

```
type Friend = tuple
                departure_date : nat
                arrival_date : nat
              end tuple
```

4. Prototipo del algoritmo:

```
fun sat_phone (friends: Set of Friend, date: nat) ret max_amount: nat
     friends_aux: Set of Friend
     chosen: Friend

     friends_aux := copy_set(friends)

     max_amount := 0
     while not is_empty_set(friends_aux) do
          chosen := selectFriends(friends_aux, date)

          {- Ya se lo presté a un amigo -}
          max_amount := max_amount + 1

          {- Elimino de los amigos restantes al amigo al que ya se lo presté -}
          elim_set(friends_aux, chosen)

          {- La nueva fecha a partir de la cual puedo prestar el teléfono es recién
          el día siguiente al que volvió el último amigo al que se lo presté. -}
          date := chosen.arrival_date + 1

          {- Elimino de los amigos a aquellos a los que ya no se lo puedo prestar
          por la fecha en la que estoy. Es decir, si se lo presté al amigo que
          volvía el día 9, es claro que no se lo puedo prestar a los amigos que se
          fueron antes del día 9. -}
          elim_friends(friends_aux, date)
     od
     destroy_set(friends_aux)
end fun
```

EJERCICIO 6

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 $t_i \leq T_i$ para todo $i \in \{1, \ldots, n\}$, ¿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.

DATOS:

- Tenemos n facturas
- Cada factura i tiene un tiempo mínimo necesario de cocción t_i y un tiempo máximo admisible de cocción T_i
- Extraer facturas del horno no insume tiempo.

Se desea EXTRAER todas las facturas del horno en perfecto estado, abriendo el horno el menor número de veces posible.

Extraer cada factura i en perfecto estado quiere decir que las saco del horno cuando hay pasado una cantidad de tiempo t tal que $t_i \leq t \leq T_i$.

Ahora bien, analicemos algunos posibles criterios de selección:

Si saco cada factura apenas está, es decir cuando $t = t_i$, voy a abrir el horno el mayor número de veces ya que por cada vez que lo abra solo voy a sacar una factura (o a lo sumo todas las que tengan el mismo t_i).

Si abro el horno cuando la última factura esté lista, es decir la factura con t_i máximo, voy a poder sacar todas las facturas pero hay muchas que se me van a haber quemado.

Ahora bien, si elijo abrir el horno justo antes que se queme la factura que tenga T_i mínimo, es decir justo en el tiempo $t = T_i$ (con T_i mínimo), me aseguro que ninguna factura se me quema y además, voy a poder sacar esa factura y todas las que ya se pueden sacar, es decir todas aquellas cuyo t_i sea menor o igual al T_i de la factura elegida. **Elijo este último criterio.**

Estructuras de datos:

Para representar a cada factura, usaré una tupla que contiene los dos tiempos de cada factura.

```
type Factura = tuple
                    minT : nat
                    maxT : nat
               end tuple
```

Implementación del algoritmo:

```
fun ricasFacturas (facturas: Set of Factura) ret veces: nat
    var facturas_aux: Set of Factura
    var elegida: Factura
    tiempo: nat

    facturas_aux := copy_set(facturas)
```

```
     veces := 0
     tiempo := 0

     {- Mientras aún queden facturas por sacar -}
     while not is_empty_set(facturas_aux) do
          elegida := selecFactura(facturas_aux)

          veces := veces + 1

          tiempo := elegida.maxT

          elimSacadas(facturas_aux, tiempo)
     od

     destroy(facturas_aux)
end fun

fun selecFactura(f: Set of Factura) ret factura: Factura
     var f_aux: Set of Factura
     var factura_aux: Factura
     var minTiempo: nat

     f_aux := copy_set(f)
     minTiempo := infinito

     while not is_empty_set(f_aux) do
          factura_aux := get(f_aux)
          if factura_aux.maxT < minTiempo then
               minTiempo := factura_aux.maxT
               factura := factura_aux
          fi
          elim(f_aux, factura_aux)
     od

     destroy(f_aux)
end fun

proc elimSacadas (in/out f: Set of Factura, in t: nat)
     var f_aux: Set of Factura
     var factura: Factura

     f_aux := copy_set(f)

     while not is_empty_set(f_aux) do
          factura := get(f_aux)
          if factura.minT ≤ t then
               elim(f, factura)
          fi
```

```
        elim(f_aux, factura)
    od

    destroy(f_aux)
end proc
```

EJERCICIO 8

Analizo el enunciado:

- Hay n troncos de leña.
- Cada tronco i, irradia una temperatura k_i y dura prendido un tiempo t_i.
- Se pide encontrar el **orden** en que se utilizan la menor cantidad de troncos posible entre las 22hs y las 12hs del día siguiente, tal que:
    - Entre las 22hs y las 6hs, la temperatura irradiada no es menor a K_1.
    - Entre las 6hs y las 12hs, la temperatura irradiada no es menor a K_2.

Asumiremos $K_1 > K_2$

Criterio de selección

Entre las 22hs y las 6hs: elijo el tronco con mayor tiempo t_i, tal que k_i es mayor o igual que K_1.
Entre las 6hs y las 12hs: idem, pero con K_2.

Estructuras de datos:

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

Prototipo de la función:

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

OBSERVACIÓN: Tratamos de hacerlo con arreglos para ordenarlo según el criterio de selección y así hacer más eficiente el algoritmo, pero nos dimos cuenta que el criterio para la ordenación no es trivial, por lo tanto pasamos a usar conjuntos.

Implementación:

```
fun estufaVoraz(S: Set of Tronco, K1: float, K2: float) ret L: List of Tronco
    var S1: Set of Tronco
    var t: Tronco
    var h: float {- la variable h lleva la cuenta de las horas -}

    S1 := copy_set(S,S1)
    h := 0
    L := empty_list()


    while h < 14 do {- entre las 22hs y las 12hs hay 14 horas -}
```

```
          if h < 8 then {- cuando estoy entre las 22hs y las 6hs -}
                t := elegirTronco(S1,K1)
          else {- cuando estoy entre las 6hs y las 12hs -}
                t := elegirTronco(S1,K2)
          fi

          elim_set(S1, t)
          addr(L, t)
          h := h + t.tiempo
     od

     destroy_set(S1)
end fun

fun elegirTronco(S: Set of Tronco, K: float) ret t: Tronco
     var S1: Set of Tronco
     var max_tiempo : float
     var t1: Tronco

     max_tiempo := -infinito
     S1 := copy_set(S)

     while not is_empty_set(S1) do
          t1 := get(S1)
          if t1.tiempo > max_tiempo && t1.calor >= K then
                max_tiempo := t1.tiempo
                t := t1
          fi
          elim_set(S1, t1)
     od

     destroy_set(S1)
od
```