AYED II
Lautaro Gastón Peralta

Práctico 3.1

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.

Solucion voraz: elegir los objetos por su valor relativo (v/w).
Ejemplo:
Supongamos que tenemos 3 objetos:
Objeto 1 => v1 = 60 ; w1 = 3 ; v1/w1 = 20
Objeto 2 => v2 = 100 ; w2 = 2 ; v2/w2 = 50
Objeto 3 => v3 = 120 ; w3 = 4 ; v/3w3 = 30

Tenemos una mochila con W = 5.
Como no se puede fragmentar la cantidad a seleccionar de los objetos, procederemos a elegir el de mayor v/w colocando su w comunmente.
Por obvias razones, el orden de los objetos segun su v/w es: Objeto 2 > Objeto 3 > Objeto 1
Cuando estemos en el proceso de seleccion:

-Seleccionaremos el Objeto 2, dejandonos un W = W - w2 = 5 - 2 = 3.
Apartir de aca, solo podemos seleccionar objetos con w <= 3.
-Si intentamos agregar el Objeto 3, veremos que: w3 > 3.
Por ende, no podemos seleccionar este Objeto.
-Seleccionamos el objeto 1, dejandonos un W = W - w1 = 3- 3 = 0

Haciendo la suma de los valores, obtenemos 20 + 30 = 50.
Cuando la solución óptima para cuando se permite fragmentación, hubiese sido elegir el Objeto 2 y Objeto 3.

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.

sdasdsaadsad

Ejemplo:

- Monedas disponibles: 1,2,4,81, 2, 4, 81,2,4,8.
- Monto M=29M = 29M=29.

Solución voraz:

1. Toma 1 moneda de 16: queda 29−16=13.
2. Toma 1 moneda de 8: queda 13−8=5.
3. Toma 1 moneda de 4: queda 5−4=1.
4. Toma 1 moneda de 1: queda 1−1=0.

Resultado: 16+8+4+1=29, utilizando 4 monedas.

Solución óptima: La misma, ya que no hay una forma más eficiente de alcanzar 29 usando menos monedas.

Se cumple debido a que cualquier moneda Mi+1 >= 2 * Mi. Posibilitando que cualquier valor pueda cubrirse con monedas más chicas.

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 $l_1, \ldots, l_{n-1}$ 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.

***Criterio de seleccion:** Hacer que dure lo mas que pueda la autonomia, haciendo que la autonomia actual, si es mayor a la distancia a la localidad recorrer, solamente se reste la autonomia - distancia, en el caso de que la localidad siguiente tenga distancia mayor, recargar combustible.

***Estructura de datos:**

```
type Cargas = tuple
                cargtot : nat
                localidades : List of string
              end tuple
```

***Algoritmo:**

```
fun min_cargas_loc (l : array[0..N] of String, d : array[0..N] of Nat, A : nat ) ret res : Cargas
    var autonomia : nat
    autonomia := 0
    res.cargtot := 0
    res.localidades := empty_list()
    for i:=0 to N-1 do
        if autonomia < d[i+1] then
            autonomia:=A
            res.cargtot:= res.cargtot + 1
            addr(res.localidades, i[i+1]
        else
            autonomia:= autonomia - d[i+1]
        fi
    od
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 $s_1, s_2, \ldots, s_n$ 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.

*Criterio de selección: Seleccionar aquella ballena que tenga menor tiempo de sobrevivir. Mientras este viva.

FORMA 1:

*Estructura de datos:

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

*Algoritmo:

```
fun rescate (tiempos : Set of Ballena, t : nat) ret ordRes : List of Ballena
    var t_trans : nat
    var tiempos_aux : Set of Ballena
    var b_rescatar : Ballena
    t_trans := 0
    tiempos_aux := copy_set(tiempos)
    While(not is_emptySet (tiempos_aux)) do
        b_rescatar = min_life_ballena(tiempos_aux)
        if b_rescatar.life >= t_trans then
            addr(ordRes, b_rescatar)
            t_trans := t_trans + t
        else
            elim(tiempos_aux, b_rescatar)
        fi
    od
    destroy_set(tiempos_aux)
end fun

{-funcion auxiliar-}
fun min_life_ballena (c : Set of Ballena) ret min_life : Ballena
    var c_aux : Set of Ballena
    var ballena : Ballena
    c_aux = copy_set(c)
    min_life := get(c_aux)
    elim(c_aux, min_life)
    while (not is_emptySet(c_aux)) do
        ballena := get(c_aux)
        if ballena.life <= min_life.life then
```

```
                    min_life:=ballena
                fi
                elim(c_aux, ballena)
        od
        destroy_set(c_aux)
end fun
```

FORMA 2:

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

fun salvarBallenas (B : Set of Ballena, t : Nat) ret rescatadas : List of Ballena
end fun


fun salvarBallenas (B : Set of Ballena, t : Nat) ret rescatadas : List of Ballena
  var ballenasAunVivas : set of Ballena
  var hora : Nat
  var ballena : Ballena
  hora:=0
  ballenasAunVivas:=set_copy(B)
  rescatadas:=empty_list()
  {- Invariante: toda ballena en ballenasAunVivas está viva, es decir
  tiempoRestante >= hora, y NO está salvada, es decir no está en rescatadas-}
  while (not is_empty_set(ballenasAunVivas)) do
    {-elijo la ballena candidata segun criterio de selección-}
    ballena:=elegirBallenta(ballenasAunVivas)
    {-agrego candidata elegida a la solución-}
    addR(rescatadas,ballena)
    {-elimino la ballena rescatada de las aun vivas no rescatadas-}
    elim_set(ballenasAunVivas, ballena)
    {-dado que salve a una ballena, tengo que actualizar el reloj-}
    hora:=hora + t
    quitarMuertas(ballenasAunVivas, hora)
end fun

{-Idea: para cada elemento en el conjunto B, me fijo si tiempoRestante es menor a
hora. En ese caso lo elimino de B-}
proc quitarMuertas(in/out B : Set of Ballena, hora : Nat)
  var D : Set of Ballena
  var b : Ballena
  D:=copy_set(B)
  while (not is_empty_set(D)) do
    {-agarro una ballena del conjunto D-}
    b:=get(D)
    if b.tiempoRestante < hora then
```

```
        elim(B,b)
      fi
      elim(D,b)
   od
   destoy_set(D)
end proc

{- elijo la ballena candidata según criterio de selección: la ballena con menor tiempo de vida. PRE: B es no vacío -}
fun elegirBallena(B : Set of Ballena) ret b : Ballena
   var b_aux : Ballena
   var min_tiempo_restante : nat
   var B_aux : Set of Ballena
   min_tiempo_restante:= infinito
   B_aux:= copy_set(B)
   while (not is_empty_set (B_aux)) do
      b_aux:= get(B_aux)
      if (b_aux.tiempoRestante < min_tiempo_restante) then
         min_tiempo_restante := b_aux.tiempoRestante
         b:= b_aux
      fi
      elim_set(B_aux,b_aux)
   od
   destroy_set(B_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.

***Datos importantes:**

- **N amigos (1<= i <= N).**
- **Conocemos p_i y r_i (Dia de partida/Dia de regreso).**

***Problema a solucionar: prestarle el teléfono al mayo número de amigos posibles.**
***Criterio de selección: Seleccionar aquel que primero regrese.**
***Estructura de datos:**

```
type Persona = tuple
                  nombre : String
                  partida : nat
                  regreso : nat
               end tuple
```

Algoritmo:

```
fun prestar_t (amigos : array[1..N] of Persona) ret res : List of Persona
   var orden_a : array [1..N] of Persona
```

```
        var dia_actual : nat
        orden_a := copy_array(amigos)
        orden_a := sort_by_retorno(orden_a)
        res:= empty_list()
        for i:=1 to n do
                if (dia < orden_a[i].partida) then
                        addr (res, orden_a[i])
                        dia:= dia + orden_a[i].regreso
                fi
        od
end fun

fun copy_array (a : array[1..N] of T) ret b : array[1..N] of T
        for i:=1 to N do
                a[i] := b[i]
        od
end fun

fun sort_by_retorno (a : array [1..n] of Persona) ret b : array[1..n] of Persona
   var retorno_min : nat
   for j:=1 to n do
      retorno_min := min_retorno_pos(a,i)
      swap(a, i, retorno_min)
   od
end fun

fun min_retorno_pos (a : array [1..n] of T, i : nat) ret retorno_min : nat
   retorno_min:=i
   for j:=i+1 to n do
      if (a[j].retorno < a[retorno_min].retorno) then
         retorno_min:= j
      fi
   od
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 $t_i \leq T_i$ 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 importantes:**

- **Abrir el horno el menos numero de veces posible.**
- **No sacar nada del horno demasiado temprano ni demasiado tarde.**
- **Horno => N (1<=i<=N)piezas de panadería (facturas,medialunas,etc).**
- **Cada pieza tiene :**

- tiempo de cocción minimo necesario t_i.
- tiempo maximo admisible de coccion T_i.

- **si se extrae del horno antes del t_i = cruda.**
- **si se extra despues de T_i = quemada.**
- **Extraer piezas del horno no quita tiempo.**
- **t_i<=T_i**

***Problema a solucionar: Conseguir abrir el horno el menor número de veces posible.**
***Criterio de seleccion:** Sacar una medialuna cuando este al borde del T_i (tiempo actual = T_i), y aprovechar en ese momento, de sacar otras piezas que esten entre el t_i y T_i.
***Estructura de datos:**

```
type medialuna = tuple
                    crudo : nat
                    quemado : nat
                 end tuple
```

*__Algoritmo:__

```
fun horno (C : set of medialuna) ret open : nat
    var C_aux : set of medialuna
    var M_sacar : set of medialuna
    var tiempo : nat
    M_sacar := empty_set()
    C_aux := copy_set(C)
    tiempo:=0
    open:=0
    while (not is_emptySet(C)) do
        M_sacar := select_medialuna(C_aux, tiempo)
        diferenciaset(C_aux, M_sacar)
        open:=open+1
        tiempo:=tiempo+1
    od
    destroy_set(C_aux)
    destroy_set(M_sacar)
end fun

fun select_medialuna(C : set of  medialuna, t : nat) ret res : set of medialuna
    var C_aux : set of medialuna
    var m : medialuna
    C_aux := copy_set(C)
    while (not is_emptySet (C)) do
        m := get(C_aux)
        if m.quemado >= t v m.crudo <= t then
            add(res,m)
        fi
        elim(C_aux, m)
    od
    destroy_set(C_aux)
end fun
```

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

   (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 $m \leq n$).

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 $K2$.

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.