EXAMEN FINAL 7 DE JULIO DE 2021 (TEMA 2)

EJERCICIO 2

2. (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.

   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:

- Tengo n días de vacaciones
- Tengo m raciones diarias, con m > n.
- Cada ración i tiene una fecha de vencimiento v_i, contada desde el día en que llego a la montaña.

Para cada día j, debo indicar qué vianda voy a comer, asegurando que nunca como algo vencido.

a) CRITERIO DE SELECCIÓN

En cada momento, elijo comer la vianda que esté más próxima a vencerse, asegurándome que no se haya vencido aún.

b) ESTRUCTURAS DE DATOS

Representaré a las viandas con una tupla, cuyos elementos son el nombre de la vianda y la fecha de vencimiento de la misma:

```
type Vianda = tuple
                id: string
                vencimiento: nat
              end tuple
```

Para todas las viandas que preparé para el viaje, voy a usar un set (conjunto) de viandas.
El algoritmo devolverá una lista cuyo elemento i es la vianda que se comerá el día i.

c) EXPLICACIÓN DEL ALGORITMO

Del conjunto de viandas, selecciono la vianda cuya fecha de vencimiento sea la más próxima. En otras palabras, aquella vianda cuyo vencimiento sea mínimo. Esta vianda seleccionada es la que voy a comer el día j.
Luego, como ya voy a haber comido esa vianda, la elimino del conjunto de viandas disponibles para comer y la agrego al conjunto solución de viandas comidas.
Avanzo un día para ver qué voy a comer en el día j+1.

Una vez que estoy en el “nuevo” día, elimino las viandas que ya han vencido y no puedo comer. De esta forma, me aseguro que la próxima vianda que seleccione no va a estar vencida.
Cabe observar que puede haber días en los que no coma.
OBSERVACIÓN: asumo que el día en el que vence una vianda, no la puedo comer. Por ejemplo, si estoy eligiendo qué voy a comer el día 2, y tengo una vianda que vence el día 2, entonces no la voy a poder comer.

d) IMPLEMENTACIÓN

```
fun queComo (viandas: Set of Vianda, n: nat) ret comidas: List of string
     var viandas_aux: Set of Vianda
     var laComo: Vianda
     var dia: nat

     viandas_aux := copy_set(viandas)
     dia := 1

     comidas := emtpty_list()

     while not is_empty_set(viandas_aux) && dia ≤ n do
          laComo := selecVianda(viandas_aux)

          addr(comidas, laComo.id)
          elim(viandas_aux, laComo)

          dia := dia + 1

          elimVencidas(viandas_aux, dia)
     od

     destroy(viandas_aux)
end fun

fun selecVianda(v: Set of Vianda) ret vianda: Vianda
     var v_aux: Set of Vianda
     var vianda_aux: Vianda
     var minVenc: nat

     v_aux := copy_set(v)
     minVenc := infinito

     while not is_empty_set(v_aux) do
          vianda_aux := get(v_aux)
          if vianda_aux.vencimiento < minVenc then
```

```
                minVenc := vianda_aux.vencimiento
                vianda := vianda_aux
           fi
           elim(v_aux, vianda_aux)
     od

     destroy(v_aux)
end fun

proc elimVencidas(in/out v: Set of Vianda, in d: nat)
     var v_aux: Set of Vianda
     var vianda: Vianda

     v_aux := copy_set(v)

     while not is_empty_set(v_aux) do
          vianda := get(v_aux)
          if vianda.vencimiento ≤ d then
               elim(v, vianda)
          fi
          elim(v_aux, vianda)
     od

     destroy(v_aux)
end proc
```

OTRA SOLUCIÓN:

Usando un arreglo de viandas ordenado crecientemente de acuerdo a las fechas de vencimiento de las viandas. Cabe aclarar que esta solución ASUME que todos los días voy a comer algo, es decir para n días mi solución va a tener n viandas (1 vianda por día).

```
fun queComo (viandas: array[1..m] of Vianda) ret comidas: array[1..n] of
string
     var i: nat
     var viandas_aux: array[1..m] of Vianda

     viandas_aux  := copy_array(viandas)
     sorty_by_vencimiento(viandas_aux)
     i := 1

     for j:=1 to n do
          while viandas[i].vencimiento < j do
```

```
                i := i+1
            od
            comidas[j] := viandas[i].id
            i := i + 1
        od
end fun
```

EJERCICIO 3

3. (Comprensión de algoritmos) 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).

```
fun s(p: array[1..n] of nat, v,w: nat) ret y: nat
    y:= v
    for i := v+1 to w do
        if p[i] < p[y] then y:= i fi
    od
end fun

fun t(p: array[1..n] of nat, v,w: nat) ret y: nat
    y:= v
    for i := v+1 to w do
        if p[y] < p[i] then y:= i fi
    od
end fun
```

```
proc r(p: array[1..n] of nat)
    for i := 1 to n div 2 do
        swap(p, i, s(p, i, n-i+1));
        swap(p, n-i+1, t(p, i+1, n-i+1));
    od
end fun
```

a) ¿QUE HACEN?

- ➔ Algoritmo s

  Dado un arreglo p[1..n] de naturales y dos naturales v y w, devuelve la posición dentro del arreglo p del elemento mínimo en el segmento p[v..w].

- ➔ Algoritmo t

  Dado un arreglo p[1..n] de naturales y dos naturales v y w, devuelve la posición dentro del arreglo p del elemento máximo en el segmento p[v..w].

- ➔ Algoritmo r

  Dado un arreglo p[1..n] de naturales, ordena los elementos del arreglo de forma creciente (de menor a mayor).

b) ¿COMO LO HACEN?

- Algoritmo s

  Recorre el segmento [v..w] del arreglo de entrada de izquierda a derecha.
  Comienza suponiendo que el elemento mínimo del segmento está en la posición v (y:=v).
  Luego, recorriendo los elementos del segmento [v+1..w] de izquierda a derecha, si encuentra un elemento que sea menor al elemento en la posición y, modifica el valor de la variable y con la posición de este "nuevo" elemento mínimo, de forma tal que en la variable y quede la posición del elemento mínimo del segmento recorrido hasta el momento.

- Algoritmo t

  Su forma de encontrar el máximo del segmento [v..w] es semejante a la del algoritmo s de encontrar el mínimo:
  Recorre el segmento [v..w] del arreglo de entrada de izquierda a derecha.
  Comienza suponiendo que el elemento máximo del segmento está en la posición v (y:=v).
  Luego, recorriendo los elementos del segmento [v+1..w] de izquierda a derecha, si encuentra un elemento que sea mayor al elemento en la posición y, modifica el valor de la variable y con la posición de este "nuevo" elemento máximo, de forma tal que en la variable y quede la posición del elemento máximo del segmento recorrido hasta el momento.

- Algoritmo r

  Lo que hace este algoritmo es llamar en cada iteración a los dos algoritmos anteriores.
  De esta forma, va ordenando el arreglo "desde afuera hacia adentro". Más explícitamente:
  En la primera iteración (i=1):
    Llamando al algoritmo s, encuentra el elemento mínimo del segmento p[1..n] y lo intercambia con el elemento que está en la primera posición del arreglo.
    Llamando al algoritmo t, encuentra el elemento máximo del segmento p[2..n] (pues en p[1] ya está el mínimo) y lo intercambia con el elemento que está en la última posición del arreglo.
  En la primera iteración (i=2):
    Llamando al algoritmo s, encuentra el elemento mínimo del segmento p[2..n-1] y lo intercambia con el elemento que está en la segunda posición del arreglo.
    Llamando al algoritmo t, encuentra el elemento máximo del segmento p[3..n-1] (pues en p[2] ya está el mínimo del segmento actual) y lo intercambia con el elemento que está en la penúltima posición del arreglo.
  En cada iteración, va ordenando dos elementos, dejándolos en su posición definitiva, es una especie de selection sort.
  Notar que como en cada iteración ordena dos elementos, está bien que itere (n div 2) veces.

c) ORDEN DE LOS ALGORITMOS

- Algoritmo s

  Es claro que la operación representativa de este algoritmo es la comparación entre elementos del arreglo de entrada.
  Por cada iteración del ciclo, se realiza exactamente una comparación. Luego, se realizarán tantas comparaciones como veces itere el ciclo. El ciclo itera desde v+1 hasta w, es decir

w - (v+1) + 1 = w - v - 1 + 1 = w-v veces.
Luego, el algoritmo realiza w-v comparaciones entre elementos del arreglo de entrada.
Por lo tanto, el algoritmo s es de orden O(w-v).

➔ Algoritmo t
Nuevamente, la operación representativa es la comparación entre elementos del arreglo de entrada, que se hace tantas veces como itere el ciclo.
Es fácil de ver que el ciclo itera tantas veces como el ciclo del algoritmo s.
Luego, siguiendo un razonamiento análogo al del algoritmo anterior, es claro que el algoritmo s es de orden O(w-v).

➔ Algoritmo r
El algoritmo r llama en cada iteración a los dos algoritmos anteriores, que ya vimos que son de orden O(w-v).
Al algoritmo s, lo llama con la instanciación v := i, w:= n-i+1. Luego, en cada iteración del ciclo, el algoritmo s es de orden O(n-i+1 - i) = O(-2i+n+1).
Al algoritmo t, lo llama con la instanciación v := i+1, w:= n-i+1. Luego, en cada iteración del ciclo, el algoritmo s es de orden O(n-i+1 - (i+1)) = O(n-i+1 - i - 1) = O(-2i+n).
El ciclo itera desde 1 hasta n/2. Luego, el algoritmo en total:

$$\sum_{i=1}^{n/2}(-2i+n+1)+(-2i+n)=\sum_{i=1}^{n/2}(-4i+2n+1)=-4\sum_{i=1}^{n/2}i+2n\sum_{i=1}^{n/2}1+\sum_{i=1}^{n/2}1=\frac{-4*n/2*(n/2+1)}{2}+2\frac{n*n}{2}+\frac{n}{2}=¿¿$$

$$¿-\frac{4*n^2+2n}{8}+n^2+\frac{n}{2}=\frac{-n^2+2n}{2}+n^2+\frac{n}{2}=\frac{-n^2}{2}-n+n^2+\frac{n}{2}=\frac{n^2}{2}-\frac{n}{2}=O(n^2)$$

Por lo tanto, el algoritmo r es de orden O(n^2).

d) NOMBRES MÁS ADECUADOS

```
fun local_min_pos(p: array[1..n] of nat, start,end: nat) ret min_pos: nat
     min_pos := start
     for i:=start + 1 to end do
          if p[i] < p[min_pos] then min_pos := i fi
     od
end fun

fun local_max_pos(p: array[1..n] of nat, start,end: nat) ret max_pos: nat
     max_pos := start
     for i:=start + 1 to end do
          if p[max_pos] < p[i] then max_pos := i fi
     od
end fun

proc sort_array (p: array[1..n] of nat)
     for i:=1 to n div 2 do
          swap(p, i, local_min_pos(p, i, n-i+1))
```

```
            swap(p, n-i+1, local_max_pos(p, i+1, n-i+1))
      od
end proc
```

EJERCICIO 4

(a) A partir de la siguiente implementación de listas mediante punteros, implementá los constructores y las operaciones addr, take, y length.

```
implement List of T where

type Node of T = tuple
                   elem : T
                   next : pointer to (Node of T)
                 end tuple

type List of T = pointer to (Node of T)

fun empty() ret l : List of T
    l := null
end fun

fun empty ()ret l : List of T
    l := NULL
end fun

proc addl (in e : T, in/out l : List of T)
  var p : pointer to (Node of T)
  alloc(p)
  p->elem := e
  p->next := l
  l := p
end proc
```

a) IMPLEMENTACIÓN DE OPERACIONES

Los constructores ya están implementados.

Operación addr (agrega el elemento e al final de la lista l)

```
proc addr(in/out l: List of T, in e: T)
     var new_node: pointer to (Node of T)
     var p: pointer to (Node of T)
```

```
     alloc(new_node)
     new_node->elem := e
     new_node->next := null

     if l = null then
          l := new_node
     else
          p := l
          while p->next != null do
               p := p->next
          od

          p->next := new_node
     fi
end proc
```

Operación take (deja en l solo los primeros n elementos, eliminando el resto)

```
proc take(in/out l: List of T, in n: nat)
     var p: pointer to (Node of T)
     var p_aux: pointer to (Node of T)
     var i: nat

     i := 1
     p := l
     while p != null && i ≤ n do
          p := p->next
          i := i + 1
     od
     while p != null do
          p_aux := p
          p := p->next
          free(p_aux)
     od
end proc
```

Operación length (devuelve la cantidad de elementos de la lista l)

```
fun length(l: List of T) ret n: nat
     var p: pointer to (Node of T)

     n := 0
     p := l
```

```
     while p != null do
          n := n+1
          p := p->next
     od
end fun
```

b) USO DEL TAD

(b) Implementá una función que reciba una lista de enteros y encuentre la posición donde se encuentra el máximo. Dicha función debe usar el tipo **abstracto** lista, sin importar cuál es su implementación.

OBSERVACIÓN: voy a suponer que el primer elemento de la lista está en la posición 0

```
{- PRE: not is_empty_list(l) -}
fun list_max_pos(l: List of T) ret max_pos: nat
     max_pos := 0
     for i:=1 to length(l) do
          if index(l,i) > index(l,max_pos) then
               max_pos := i
          fi
     od
end fun
```