EXAMEN FINAL 7 DE JULIO DE 2021 (TEMA 1)

EJERCICIO 2

2. (Algoritmos voraces) Llega el final del cuatrimestre y tenés la posibilidad de rendir algunas de las materias $1, ..., n$. Para cada materia i conocés la fecha de examen f_i, y la cantidad de días inmediatamente previos d_i que necesitás estudiarla de manera exclusiva (o sea, no podés estudiar dos materias al mismo tiempo). Dar un algoritmo voraz que obtenga la mayor cantidad de materias que podés rendir.

   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:

- Hay n materias
- La fecha de examen de cada materia i es f_i
- La cantidad de días inmediatamente previos para estudiar la materia i es d_i.
- NO se pueden estudiar dos materias al mismo tiempo.

Se desea obtener la mayor cantidad de materias que se pueden rendir.

a) CRITERIO DE SELECCIÓN

Elijo rendir la materia cuya fecha de examen sea la más próxima a la fecha actual, siempre y cuando me alcancen los días para estudiarla. Si no me alcanzan, elijo la siguiente más próxima.

b) ESTRUCTURAS DE DATOS

Representaré a cada materia con una tupla compuesta por dos elementos: uno es la fecha de examen y otro es la cantidad de días que me lleva preparar esa materia.

```
type Materia = tuple
                 fecha: nat
                 estudio: nat
               end tuple
```

c) EXPLICACIÓN

Inicialmente, recibo el conjunto de materias y elimino aquellas que no llego a estudiar (no me dan los días).

Una vez hecho esto, recorro el conjunto de materias eligiendo en cada paso la materia cuya fecha de examen sea más próxima al día actual, siempre y cuando me alcancen los días que requiere estudiarla.

Una vez que rindo una materia, sumo una unidad al contador de materias rendidas, y la elimino de las materias que me quedan por rendir, y avanzo hasta el día siguiente a la fecha de examen de esa materia (pues el día que rindo no puedo estudiar).

Una vez que estoy en esa fecha, elimino de las materias que me quedan por rendir a aquellas que no me alcanzan los días para estudiarla estando en la fecha actual.

Repito este procedimiento hasta que no me queden más materias disponibles para rendir.

d) IMPLEMENTACIÓN

```
fun aRendir(materias: Set of Materia) ret max_materias: nat
     var materias_aux: Set of Materia
     var dia: nat
     var laRindo: Materia

     dia := 0
     max_materias := 0
     materias_aux := copy_set(materias)

     elimNoLlego(materias_aux)
     while not is_empty_set(materias_aux) do
          laRindo := selecMateria(materias_aux)

          max_materias := max_materias + 1
          elim(materias_aux, laRindo)

          dia := laRindo.fecha + 1

          elimNoLlego(materias_aux, dia)
     od

     destroy_set(materias_aux)
end fun

fun selecMateria(m: Set of Materia) ret materia: Materia
     var m_aux: Set of Materia
     var materia_aux: Materia
     var fechaMin: nat

     materia := get(m)
     fechaMin := materia.fecha

     m_aux := copy_set(m)

     while not is_empty_set(m_aux) do
          materia_aux := get(m_aux)
          if materia_aux.fecha < fechaMin then
               fechaMin := materia_aux.fecha
               materia := materia_aux
          fi
          elim(m_aux, materia_aux)
```

```
     od

     destroy_set(m_aux)
end fun

proc elimNoLlego(in/out m: Set of Materia, in d: nat)
     var materia: Materia
     var m_aux: Set of Materia

     m_aux := copy_set(m)

     while not is_empty_set(m_aux) do
          materia := get(m_aux)
          if materia.fecha - materia.estudio < d then
               elim(m, materia)
          fi
          elim(m_aux, materia)
     od

     destroy_set(m_aux)
end proc
```

OTRA OPCIÓN: con arreglos

```
fun aRendir(materias: array[1..n] of Materia) ret max_materias: nat
     var materias_aux: array[1..n] of Materia
     var dia: nat

     materias_aux := copy array(materias)

     {- En materias_aux voy a tener las materias ordenadas crecientemente
     según su fecha de examen -}
     sort_by_fecha(materias_aux)

     max_materias := 0
     dia := 0

     for i:=1 to n do
          if materias_aux[i].fecha - materias_aux[i].estudio ≥ dia then
               max_materias := max_materias + 1
               dia := materias_aux[i].fecha + 1
          fi
     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).

```
proc p(a: array[1..n] of nat)
    var d: nat
    for i:= 1 to n do
        d:= i
        for j:= i+1 to n do
            if a[j] < a[d] then d:= j fi
        od
        swap(a, i, d)
    od
end proc
```

```
fun f(a: array[1..n] of nat) ret b : array[1..n] of nat
    var d: nat
    for i:= 1 to n do b[i] := i od
    for i:= 1 to n do
        d:= i
        for j:= i+1 to n do
            if a[b[j]] < a[b[d]] then d:= j fi
        od
        swap(b, i, d)
    od
end fun
```

a) ¿QUE HACEN?

➔ Algoritmo p

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

➔ Algoritmo f

Dado un arreglo a[1..n] de naturales, devuelve un arreglo b[1..n] de naturales cuyos elementos son los índices de los elementos del arreglo a e indican cómo deberían estar ordenados los elementos del arreglo a para que a esté ordenado.

Por ejemplo, si a es un arreglo de tamaño 5, es decir a: array[1..5] y el resultado de ejecutar el algoritmo f es b = [4,2,1,5,3], quiere decir que para que a esté ordenado a[4] debería ir en la primera posición, a[2] en la segunda, a[1] en la tercera, a[5] en la cuarta y a[3] en la última.

b) ¿COMO LO HACEN?

➔ Algoritmo p

Es el algoritmo de ordenación selection sort:

1. Busca el elemento mínimo del arreglo a (es decir del segmento a[1..n]) y lo intercambia con el elemento que se encuentra en la primera posición.

2. Busca el elemento mínimo de los restantes (es decir del segmento a[2..n]) y lo intercambia con el elemento que se encuentra en la segunda posición.
   (... en cada paso, ordena un elemento…)
   Repite el procedimiento hasta ordenar todo el arreglo.

➔ Algoritmo f

Hace exactamente lo mismo que el algoritmo p nada más que a los intercambios los efectúa sobre el arreglo b, que es el que inicialmente tiene los índices de los elementos del arreglo a. Más explícitamente:

1. Busca el elemento mínimo del arreglo a (es decir del segmento a[1..n]) e intercambia en el arreglo b el índice del elemento mínimo con el índice 1 (que indica la primera posición de a).
2. Busca el elemento mínimo de los restantes (es decir del segmento a[2..n]) e intercambia en el arreglo b el índice del elemento mínimo con el índice 2 (que indica la segunda posición de a).
   (... en cada paso, ordena un índice …)
   Repite el procedimiento hasta ordenar todos los índices.

c) ORDEN DE LOS ALGORITMOS

➔ Algoritmo p

Es claro que la operación representativa, es decir la que más se repite, es la comparación entre elementos del arreglo de entrada, puesto que está en un ciclo que está dentro de otro ciclo.
Veamos entonces cuántas comparaciones entre elementos de a se hacen en una ejecución arbitraria del algoritmo p:
En el ciclo “interno”, se hace exactamente una comparación por cada iteración del ciclo. Como el ciclo va desde i+1 hasta n, es claro que se hacen n - (i+1) + 1 = n - i - 1 + 1 = n - i comparaciones. Ahora bien, esta cantidad de comparaciones se hace por cada iteración del ciclo “principal”, es decir se hacen n-i comparaciones para $i \in \{1,2,3,...,n-1,n\}$. Luego, en total se hacen
(n-1) + (n-2) + (n-3) + … + (n-(n-1)) + (n-n) = (n-1) + (n-2) + (n-3) + … + 1 + 0 =
$= \sum_{i=1}^{n-1} i = \frac{n*(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$ = O(n^2) comparaciones entre elementos del arreglo de entrada.
Luego, el algoritmo p es de orden O(n^2).

➔ Algoritmo f

Si bien en este algoritmo hay un ciclo extra, la operación representativa vuelve a ser la comparación entre elementos del arreglo de entrada, puesto que está en un ciclo que está dentro de otro ciclo.
Veamos también que los límites de ambos ciclos son iguales a los del algoritmo anterior.
Luego, siguiendo un procedimiento análogo al del algoritmo anterior, se llega a que el algoritmo f es también de orden O(n^2).

d) NOMBRES MÁS ADECUADOS

```
proc selection_sort(a: array[1..n] of nat)
  var min_pos: nat
  for i:=1 to n do
     min_pos := i
     for j:=i+1 to n do
          if a[j] < a[min_pos] then min_pos := j fi
     od
     swap(a, i, min_pos)
  od
end proc

fun index_selection_sort(a: array[1..n] of nat) ret sorted_indexes: array[1..n]
of nat
  var min_pos: nat
  for i:=1 to n do sorted_indexes[i] := i od
  for i:=1 to n do
     min_pos := i
     for j:=i+1 to n do
          if a[sorted_indexes[j]] < a[sorted_indexes[min_pos]] then
min_pos := j fi
     od
     swap(b, i, min_pos)
  od
end fun
```

EJERCICIO 4)a)

(a) Implementá los constructores del TAD Conjunto de elementos de tipo T, y las operaciones member, elim e inters, utilizando la siguiente representación:

```
implement Set of T where

type Set of T = tuple
                  elems : array[0..N-1] of T
                  size : nat
                end tuple
```

CONSTRUCTORES

```
fun empty_set() ret s: Set of T
     s.size := 0
end fun
```

```
{- PRE: s.size < N (tiene que haber lugar en el arreglo para agregar un nuevo
elemento en el conjunto) -}
proc add(in e: T, in/out s: Set of T)
     s.elems[s.size] := e
     s.size := s.size + 1
end proc
```

OPERACIONES

```
fun member(e: T, s: Set of T) ret b: bool
     var i: nat

     i := 0
     b := false

     while i < s.size && not b do
          b := s.elems[i] = e
          i := i + 1
     od
end fun

proc elim(in/out s: Set of T, in e: T)
     var i: nat
     var is_member: bool

     is_member := false
     i := 0

     while i < s.size && not is_member do
          if s.elems[i] = e then
               is_member := true
               for j:=i to s.size-2 do
                    s.elems[i] := s.elems[i+1]
               od
               s.size := s.size - 1
          fi
          i := i + 1
     od
end proc

proc inters(in/out s: Set of T, in s0: Set of T)
     var s_size: nat

     s_size := s.size
```

```
    for i:=0 to s_size-1 do
        if not member(s.elems[i], s0) then
            elim(s, s.elems[i])
        if
    od
end proc
```

¿Existe alguna limitación con esta representación de conjuntos? En caso afirmativo, indica si algunas de las operaciones o constructores tendrán alguna precondición adicional.

La gran limitación que tiene esta representación de conjuntos es que la cantidad de elementos que puede llegar a tener un conjunto está sujeta al tamaño del arreglo dentro del cual está contenido ese conjunto.

Por lo tanto, se podría agregar un invariante de representación s.size ≤ N.

Ante esto, se debería agregar una precondición a la hora de agregar un nuevo elemento a un conjunto que indique que tiene que haber suficiente lugar en el arreglo para añadir un nuevo elemento al conjunto. Así, la precondición del constructor add con esta representación debería ser:

```
{- PRE: s.size < N -}
```

Tener esta precondición me asegura que que s.size + 1, que es el tamaño que va a tener el conjunto s luego de añadirle un nuevo elemento, sea menor o igual a N, que es justamente el invariante de representación.

También se debería agregar una precondición a la operación de unión de conjuntos que indique que en el conjunto s (que es a donde se van a agregar los elementos de s0) haya suficiente lugar como para agregar a los elementos de s0 que correspondan. Es decir, con esta representación, la precondición para la operación union debería ser:

```
{- PRE: cardinal(s) + cardinal(diff(s0,s)) ≤ N(s) -}
```

EJERCICIO 4)b)

(b) Utilizando el tipo **abstracto** Conjunto de elementos de tipo T, implementá una función que reciba un conjunto de enteros s, un número entero i, y obtenga el entero perteneciente a s que está *más cerca* de i, es decir, un $j \in s$ tal que para todo $k \in s$, $|j-i| <= |k-i|$. Por ejemplo si el conjunto es $1, 5, 9$, y el entero 7, el resultado puede ser 5 o 9.

```
fun closer(s: Set of int, i: int) ret res: int
    var s_aux: Set of T
    var minDist: nat
    var elem: int

    res := infinito
```

```
    minDist := infinito
    s_aux := copy_set(s)

    while not is_empty_set(s_aux) do
        elem := get(s_aux)
        if abs(elem-i) < minDist then
            minDist := abs(elem-i)
            res := elem
        fi
        elim(s_aux, elem)
    od

    destroy(s_aux)
end fun
```