EXAMEN FINAL 8 DE MARZO DE 2021

EJERCICIO 1

1. (Voraz) Malena te pide que le cuides el departamento por N días y te deja la heladera llena con N productos. Cada producto i tiene fecha de vencimiento v_i contada desde el día en que llegás a la casa. Puede haber productos ya vencidos ($v_i \leq 0$). Como no tenés un mango, te vas a alimentar comiendo un producto por día, y no te vas a hacer drama por comer algo vencido. Sin embargo te gustaría que lo que comas lleve la menor cantidad posible de días vencido. Se pide indicar para cada día j con $1 \leq j \leq N$ qué producto vas a comer, minimizando la cantidad de días que llevan vencidos los productos vencidos que comés. Para ello:

   (a) Indicá de manera simple y concreta, cuál es el criterio de selección voraz para construir la solución.

   (b) Indicá qué estructuras de datos utilizarás para resolver el problema.

   (c) Explicá en palabras cómo el algoritmo resolverá el problema.

   (d) Implementá el algoritmo en el lenguaje de la materia de manera precisa.

Primero, analizo el enunciado:

- Tenemos N días y N productos.
- Cada producto i tiene una fecha de vencimiento v_i, que se cuenta desde el día en que llego a la casa.
- Puede haber productos ya vencidos ($v_i \leq 0$)
- Voy a comer un producto por día, sin importar si como algo vencido.

Se desea comer lo que lleve la menor cantidad posible de días vencido.
Se pide indicar para cada día j con $1 \leq j \leq N$ qué producto voy a comer, MINIMIZANDO la cantidad de días que llevan vencidos los productos vencidos que como.

a) CRITERIO DE SELECCIÓN

Tenemos productos vencidos ($v_i \leq 0$) y productos NO vencidos ($v_i > 0$).
Primero, me voy a comer los productos vencidos para que no pasen muchos más días de vencimiento.
Más aún, para comer lo que lleve la menor cantidad POSIBLE de días vencido, de los productos vencidos voy a elegir cada día el que esté MÁS VENCIDO, es decir el producto cuya fecha de vencimiento sea mínima.
De esta forma, voy a evitar que los productos más vencidos se sigan venciendo (y se pudran), y los menos vencidos van a seguir relativamente en buen estado.
Además, al usar este criterio de selección, supongamos que tengo todos productos NO vencidos, voy a terminar eligiendo el que esté más próximo a vencerse.

b) ESTRUCTURAS DE DATOS

Representaré a cada producto mediante una tupla cuyos elementos son el nombre del producto y la fecha de vencimiento del producto, contada desde el día en que nos encontramos:

```
type Producto = tuple
                  id: string
                  vencimiento: int
                end tuple
```

c) EXPLICACIÓN

En cada día, elegiré de los productos vencidos a aquel cuyo vencimiento sea mínimo. De esta forma, voy a elegir de los vencidos, el que más días lleve vencido y si no hubiera vencidos, voy a elegir de los no vencidos, el más próximo a vencerse.
Luego, como ya voy a haber comido uno de los productos, lo elimino de los productos disponibles.
Este proceso se repite hasta que no queden más productos por comer, o equivalentemente hasta que no queden más días.

d) IMPLEMENTACIÓN

```
fun menosVencidos(productos:Set of Producto) ret comidos:List of string
     productos_aux: Set of Producto
     elegido: Producto

     productos_aux := copy_set(productos)

     comidos := empty_list()

     while not is_empty_set(productos_aux) do
          elegido := masVencido(productos_aux)

          elim_set(productos_aux, elegido)

          addr(elegido.id, comidos)
     od

     destroy_set(productos_aux)
end fun

fun masVencido(productos:Set of Producto) ret prod:Producto
     productos_aux: Set of Producto
     menorVencimiento: nat
     producto: Producto

     productos_aux := copy_set(productos)
     menorVencimiento := infinito

     while not is_empty_set(productos_aux) do
          producto := get(productos_aux)

          if producto.vencimiento ≤ menorVencimiento then
               menorVencimiento := producto.vencimiento
               prod := producto
          fi
```

```
          elim_set(productos_aux, producto)
     od

     destroy_set(productos_aux)
end fun
```

OTRA OPCIÓN: con arreglos

```
fun menosVencidos(productos:array[1..N] of Product) ret comidos:array[1..N] of
string
     productos_aux: array[1..N] of Producto

     productos_aux := copy_array(productos)

     {- En productos_aux voy a tener los productos ordenados de manera
     creciente según su fecha de vencimiento -}
     sort_by_vencimiento(productos_aux)

     for i:=1 to N do
          comidos[i] := productos_aux[i].id
     od
end fun
```

EJERCICIO 2

2. (Backtracking) Luego de que te dan el alta por intoxicación, Malena te pide de nuevo que le cuides el departamento. Esta vez te deja N productos que no vencen pero los tenés que pagar. Cada producto i tiene un precio p_i y un valor nutricional s_i. Tu presupuesto es M. Se pide comer productos para obtener el máximo valor nutricional sin superar el presupuesto M. No hace falta comer todos los días ni vaciar la heladera.

   (a) Especificá precisamente qué calcula la función recursiva que resolverá el problema, indicando qué argumentos toma y la utilidad de cada uno.

   (b) Da la llamada o la expresión principal que resuelve el problema.

   (c) Definí la función en notación matemática.

Como siempre, primero analizo el enunciado:

- Tenemos N productos que NO VENCEN pero hay que pagarlos.
- Cada producto i tiene un precio p_i y un valor nutricional s_i.
- Tengo un presupuesto M.

Se pide comer productos para obtener el máximo valor nutricional sin superar el presupuesto M. No hace falta comer todos los días ni vaciar la heladera.
Es claro que debo calcular un máximo.

a) FUNCIÓN RECURSIVA

productos(i ,j) = “máximo valor nutricional que se puede alcanzar comiendo algunos de los productos desde 1 hasta i (o todos) con el presupuesto j” ,

donde:

- i representa el producto hasta el cual se va a considerar
- j representa el presupuesto restante

b) LLAMADA PRINCIPAL

La llamada principal a la función recursiva productos que resuelve el problema es:

productos(N, M)

c) DEFINICIÓN DE LA FUNCIÓN EN NOTACIÓN MATEMÁTICA

```
productos(i,j) =
{- CASO BASE -}
{- No tengo productos que comer, entonces el máximo valor nutricional que puedo alcanzar es 0 -}
| i = 0 -------------> 0

{- CASOS RECURSIVOS -}
{- Caso recursivo 1: el precio del producto i es mayor al presupuesto disponible.
Entonces, no considero ese producto i. -}
| i > 0 && p_i > j -----------> productos(i-1, j)

{- Caso recursivo 2: el precio del producto i es menor o igual al presupuesto disponible. Entonces, calculo el máximo valor nutricional entre comer ese producto y no comerlo. -}
| i > 0 && p_i ≤ j -----------> max(s_i + productos(i-1, j-p_i), productos(i-1, j))
```

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 q(in/out a : array[1..N] of int, in x : nat)
    for j:= 1 to x do
        m:= j
        for k:= j+1 to x do
            if a[k] < a[m] then m:= k fi
        od
        swap(a,j,m)
    od
end proc
```

```
proc r(in/out a : array[1..N] of int, in y : nat)
    for j:= y to n do
        m:= j
        while m > y && a[m] < a[m-1] do
            swap(a,m,m-1)
            m:= m-1
        od
    od
end proc
```

```
proc p(in/out a : array[1..N] of int, in i : nat)
    q(a, i-1)
    r(a, i+1)
end proc
```

a) ¿QUÉ HACEN?

➔ Algoritmo q

Dados un arreglo a de enteros y un natural x, ordena el segmento desde 1 hasta x del arreglo a de menor a mayor.

➔ Algoritmo r

Dados un arreglo a de enteros y un natural y, ordena el segmento desde y hasta n del arreglo a de menor a mayor.

➔ Algoritmo p

Dados un arreglo a de enteros y un natural i, ordena los segmentos [1..i-1] y [i+1..N] del arreglo a de menor a mayor, dejando intacto el elemento en la posición i del arreglo.

b) ¿CÓMO LO HACEN?

➔ Algoritmo q

El algoritmo va tomando las posiciones 1 a x del arreglo de entrada. Para cada posición j que va tomando, encuentra el índice m del elemento mínimo en el segmento [j+1..x] del arreglo a. Una vez que encuentra dicho índice m, intercambia (usando el procedimiento swap) los elementos que se encuentran en las posiciones j y m, de forma tal que el elemento mínimo quede primero. Es una especie de selection sort.

➔ Algoritmo r

El algoritmo va tomando las posiciones y a N del arreglo de entrada. Para cada posición j que va tomando, acomoda el elemento que se encuentra en dicha posición en el lugar que le corresponde dentro del segmento [y..N], de forma tal que el segmento [y..N] quede ordenado de menor a mayor.

Para realizar este acomodamiento, se fija si el elemento anterior al elemento en la posición j que considera es mayor al elemento en la posición j, y en caso afirmativo, intercambia (usando el procedimiento swap) estos dos elementos, logrando que el menor quede primero. Realiza esto hasta que el elemento anterior al elemento en la posición j sea menor a este, lo que va a significar que el elemento j está bien posicionado (por lo menos hasta ese momento). Es una especie de insertion sort.

➔ Algoritmo p

El algoritmo p simplemente llama:

Al algoritmo q para lograr ordenar el segmento [1..i-1] del arreglo a.

Al algoritmo r para lograr ordenar el segmento [i+1..N] del arreglo a.

c) ORDEN DE LOS ALGORITMOS

➔ Algoritmo q

No es difícil de ver que en este algoritmo la operación representativa, es decir aquella que más se repite, es la comparación entre elementos del arreglo a. Veamos entonces:

En el ciclo de “más adentro”, se realizan x - (j+1) + 1 = x - j - 1 + 1 = x - j comparaciones.

Ahora bien, estas comparaciones se hacen para $j \in \{1,2,...,x-1,x\}$

Por lo tanto, en total son:

(x-1) + (x-2) + … + (x-(x-1)) + (x-x) = (x-1) + (x-2) + … + 1 + 0 = (x-1) + (x-2) + … + 1 =

$$= \sum_{r=1}^{x-1} r = \frac{x*(x-1)}{2} = \frac{x^{2^{\square}}-x}{2} = \frac{x^2}{2} - \frac{x}{2} \approx x^2$$

Por lo tanto, el algoritmo q es de orden x^2.

➔ Algoritmo r

Nuevamente, la operación representativa es la comparación entre elementos del arreglo a.

Aquí, podemos distinguir dos casos:

- MEJOR CASO (aquel en el que se realizan la mínima cantidad de operaciones posible): este caso ocurre cuando el arreglo a ya viene ordenado en el segmento [y..N]. En esta situación, se realiza exactamente una comparación para $j \in \{y+1,...,n-1,n\}$ (notar que para j = y, ni siquiera se entra al ciclo while porque no se cumple la guarda m > y). Es decir, en el mejor caso, se realizan n - y comparaciones.

- PEOR CASO (aquel en el que se realiza la máxima cantidad de operaciones posible): este caso ocurre cuando el arreglo a viene ordenado de mayor a menor en el segmento [y..N]. Para contar las operaciones en este caso, haré un cuadro, teniendo en cuenta lo siguiente:
  En el peor caso, siempre se va a cumplir a[minp] < a[minp-1] y el ciclo se ejecutaría minp-y veces (mientras minp > y, dado que en el ciclo se va decrementando minp de a 1). El valor de minp depende de la iteración en la que estemos del ciclo principal, pues antes de entrar al ciclo tengo minp := j. Entonces el peor caso del ciclo interno tiene j - y operaciones.

Si el valor de j es	Máxima cantidad de comparaciones posible
y	y - y = 0
y+1	(y+1) - y = 1
y+2	(y+2) - y = 2
...	...
n	n - y
TOTAL	$\sum_{r=1}^{n-y} r = \frac{(n-y)*((n-y)+1)}{2} = \frac{(n-y)^2+(n-y)}{2} = \frac{(n-y)^2}{2} - \frac{(n-y)}{2} \approx (n-y)^2$

Es decir, en el peor caso se realizan $\frac{(n-y)^2}{2} - \frac{(n-y)}{2}$ comparaciones.

Como es el peor caso el que da el orden del algoritmo, entonces el algoritmo r es del orden de $(n-y)^2$.

➔ Algoritmo p
El algoritmo p llama a q(a,i-1), que como ya vimos es de orden $(i-1)^2$; y llama a r(a, i+1), que como ya vimos es de orden $(n-(i+1))^2$.
Luego, el algoritmo p es de orden:
En el mejor caso: $(i-1)^2 + (n-(i+1))$
En el pero caso: $(i-1)^2 + (n-(i+1))^2$

d) NOMBRES MÁS ADECUADOS

```
proc selection_sort_till (in/out: array [1..N] of int, in x: nat)
     for j:=1 to x do
          min_pos := j
          for k := j + 1 to x do
               if a[k] < a[min_pos] then min_pos := k fi
          od
```

```
          swap(a,j,min_pos)
     od
end proc

proc insertion_sort_from (in/out a: array[1..N] of int, in y: nat)
     for j:=y to n do
          min_pos := j
          while min > y && a[min_pos] < a[min_pos-1] do
               swap(a, min_pos, min_pos - 1)
               min_pos := min_pos - 1
          od
     od
end proc

proc sort_except (in/out a: array[1..N] of int, in i: nat)
     selection_sort_till(a, i-1)
     insertion_sort_from(a, i+1)
end proc
```

EJERCICIO 4

a) IMPLEMENTACION DE TAD

(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

¿Existe alguna limitación con esta representación de conjuntos? En caso afirmativo indicá si algunas de las operaciones o constructores tendrán alguna precondición adicional.
NOTA: Si necesitás alguna operación extra para implementar lo que se pide, debes implementarla también.

```
implement Set of T where

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

```
Constructors
     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 poder
     agregar un nuevo elemento al conjunto) -}
     proc add (in e: T, in/out s: Set of T)
          s.elems[s.size] := e
          s.size := s.size + 1
     end proc

Operations
     fun member (e: T, s: Set of T) ret b: bool
          var i: nat
          b := false
          i := 0
          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)
     {- Tengo que encontrar la posición en la que se encuentra el elemento
     e, y una vez que tengo la posición, corro todos los elementos que están
     a la derecha del elemento e, un lugar a la izquierda para terminar
     pisándolo. -}
          var i: nat
          var b: bool

          i := 0
          b := false

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

          {- Si b = true, quiere decir que el elemento e se encuentra en la
          posición i-1. -}
          if b then
```

```
            for j:=i-1 to s.size-2 do
                s.elems[j] := s.elems[j+1]
            od
        fi

        {- Si i = s.size, quiere decir que se recorrió todo el conjunto y
        no se encontró el elemento e. Entonces, dejo al conjunto intacto.
        Para lograr esto, simplemente no agrego una guarda para este caso.
        -}
    end proc

    proc inters (in/out s: Set of T, in s0: Set of T)
        for i:=0 to s.size-1 do
            if not member(s.elems[i], s0) then
                elim(s, s.elems[i])
            fi
        od
    end proc
end implementation
```

b) USANDO EL TAD

(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 mas_cercano (s: Set of int, i: int) ret cercano: int
    s_aux : Set of int
    distancia_min: nat

    cercano := infinito
    distancia_min := infinito
    s_aux := copy_set(s)

    while not is_empty_set(s_aux) do
        elem := get(s_aux)
        if abs(elem - i) ≤ distancia_min then
            distancia_min := abs(elem - i)
            cercano := elem
        fi
        elim(s_aux, elem)
    od
```

```
end fun
```