EXAMEN FINAL 22 DE FEBRERO DE 2021

EJERCICIO 1

1. (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.

Primero, analizo el enunciado:

- Me voy n días de vacaciones.
- Llevo 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 llego a la montaña. Una vez que pasa la fecha de vencimiento, la vianda no puede comerse.

Se desea minimizar la cantidad de viandas que se vencen (y no se comen).
Para cada día j, con $1 \leq j \leq n$, indicar qué vianda se come, asegurando que nunca se coma algo vencido.

a) CRITERIO DE SELECCIÓN

Cada día, elijo comer la vianda que tenga fecha de vencimiento más próxima. Es decir, elijo la vianda i cuya fecha de vencimiento v_i sea mínima.

b) ESTRUCTURAS DE DATOS

Representaré a las viandas con una tupla compuesta por su nombre y su fecha de vencimiento:

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

c) EXPLICACIÓN DEL ALGORITMO

Como dije antes, cada día elijo comer la vianda cuya fecha de vencimiento sea mínima.
Cuando ya sé que vianda voy a comer, la agrego a la solución y la elimino del conjunto de viandas disponibles para comer.
Ya pasó un día, entonces elimino de las viandas disponibles aquellas que para este “nuevo” día ya se hayan vencido.

Aplico este procedimiento hasta que no me queden más viandas disponibles para comer o hasta que ya hayan pasado todos los días de vacaciones.

d) IMPLEMENTACIÓN

```
fun viandas (cocinadas: Set of Vianda, n: nat) ret comidas: List of string
     var cocinadas_aux: Set of Vianda
     var elegida: Vianda
     var dia: nat

     cocinadas_aux := copy_set(cocinadas)
     comidas := empty_list()
     dia := 1

     while not is_empty_set(comidas_aux) && i ≤ n do
          elegida := vencimientoMasCercano(comidas_aux)

          {- Agrego la vianda elegida a la lista solución. -}
          addr(comidas, elegida.id)

          {- Elimino la vianda elegida de las viandas disponibles para comer.
-}
          elim(comidas_aux, elegida)

          dia := dia + 1

          {- Elimino de las viandas vencidas, teniendo en cuenta el día que
          estoy considerando. -}
          elimVencidas(comidas_aux, dia)
     od

     destroy_set(comidas_aux)
end fun

fun vencimientoMasCercano (s: Set of Vianda) ret v: Vianda
     var s_aux: Set of Vianda
     var menorVencimiento: nat
     var v_aux: Vianda

     menorVencimiento := infinito
     s_aux := copy_set(s)

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

```
                menorVencimiento := v_aux.vencimiento
                v := v_aux
            fi
            elim(s_aux, v_aux)
        od

        destroy_set(s_aux)
end fun

proc elimVencidas (s: Set of Vianda, fechaVenc: nat)
        var s_aux: Set of Vianda
        var vianda: Vianda

        s_aux := copy_set(s)

        while not is_empty_set(s_aux) do
            vianda := get(s_aux)
            if vianda.vencimiento < fechaVenc then
                elim(s, vianda)
            fi
            elim(s_aux, vianda)
        od

        destroy_set(s_aux)
end proc
```

EJERCICIO 2

2. (Backtracking) Te vas de viaje a la montaña viajando en auto k horas hasta la base de un cerro, donde luego caminarás hasta el destino de tus vacaciones. Tu auto no es muy nuevo, y tiene un stereo que solo reproduce cds (compact-disks). Buscás en tu vasta colección que compraste en los años 90 y tenés p cds, con $p > k$, que duran exactamente una hora cada uno. Encontrás también un cuaderno donde le diste una puntuación entre 1 y 10 a cada cd de tu colección. Cuanto mayor la puntuación, más es el placer que te da escucharlo. Dado que no sos tan exigente, querés que el puntaje promedio entre dos discos consecutivos que escuches, no sea menor a 6. Así por ejemplo si en la hora 2 escuchás un cd que tiene puntaje 8, en la hora 3 podrías escuchar uno que tenga puntaje al menos 4.

   Encontrar una combinación de cds para escuchar en las k horas de viaje, cumpliendo la restricción de que en dos horas consecutivas el puntaje promedio de los dos discos sea mayor o igual a 6, maximizando el puntaje total de los k discos que escucharás.

   Se pide lo siguiente:

   (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.

Primero, veamos los datos que nos da el enunciado:

- Me voy de viaje a la montaña viajando k horas en auto.
- Tengo p cds, con p > k (es decir, tengo más cds que la cantidad de horas que viajo en auto).
- Cada cd tiene un puntaje del 1 al 10. Cuanto mayor la puntuación, más es el placer que te da escucharlo.
- El puntaje promedio entre dos discos consecutivos que escuche tiene que ser mayor o igual a 6.

Se desea encontrar una combinación de cds para escuchar en las k horas de viaje, cumpliendo la restricción de que en dos horas consecutivas el puntaje promedio de los dos discos sea mayor o igual a 6, MAXIMIZANDO el puntaje total de los k discos que voy a escuchar.
Es claro que debo calcular un máximo entre algo.

a) FUNCIÓN RECURSIVA

discos(cds, i, j) = "máximo puntaje posible que se puede obtener de los discos del conjunto cds para las horas de viaje desde la 1 hasta la i, escuchando en la hora i el disco j (con puntuación s_j), con la condición de que en dos horas consecutivas el puntaje promedio de los dos discos debe ser mayor o igual a 6"

Argumentos de la función recursiva:

El argumento cds indica los cds que todavía no se han escuchado en lo que va del viaje.
El argumento i indica la cantidad de horas de viaje que se está considerando.
El argumento j indica el disco que se escucha en la hora i.

b) LLAMADA PRINCIPAL A LA FUNCIÓN RECURSIVA

La llamada principal a la función recursiva que resuelve el problema está dada por la siguiente expresión:

$$\max(\text{discos}_{\{j \in cds\}}(cds,k,j))$$

c) FUNCIÓN RECURSIVA EN NOTACIÓN MATEMÁTICA

$$\begin{aligned}\text{discos}(cds, i, j) = \{&\text{- CASO BASE -}\}\\ &| \; i = 0 \dashrightarrow 0\\ &\\ &\{\text{- CASO RECURSIVO -}\}\\ &| \; i \geq 1 \dashrightarrow s_j + \max_{\{c \in cds \text{ tq } (s_c + s_j)/2 \geq 6\}}(\text{discos}(cds-\{ j \}, i-1, c))\end{aligned}$$

EJERCICIO 3

3. Para cada uno de los siguientes algoritmos determinar **por separado** cada uno de los siguientes incisos.

   (a) ¿Qué hace? ¿Cuáles son las precondiciones necesarias para haga eso?

   (b) ¿Cómo lo hace?

   (c) El orden del algoritmo, analizando los distintos casos posibles.

   (d) Proponer nombres más adecuados para los identificadores (de variables, funciones 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 p(in/out a : array[1..N] of int, in i : nat)
    q(a, i-1)
    r(a, i+1)
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
```

a) ¿QUE HACE? ¿CUALES SON LAS PRECONDICIONES NECESARIAS PARA QUE HAGA ESO?

➔ Algoritmo q

Dados un arreglo de enteros a[1..N] y un natural x, ordena el segmento a[1..x] del arreglo en forma creciente.

Precondición necesaria para que el algoritmo logre hacerlo: $1 \leq x \leq N$, es decir el parámetro x debe ser una posición válida dentro del arreglo.

➔ Algoritmo r

Dados un arreglo de enteros a[1..N] y un natural y, ordena el segmento a[y..N] del arreglo en forma creciente.

Precondición necesaria para que el algoritmo logre hacerlo: $1 \leq y \leq N$, es decir el parámetro y debe ser una posición válida dentro del arreglo.

➔ Algoritmo p

Dados un arreglo de enteros a[1..N] y una natural i, ordena el arreglo a de forma creciente salvo el elemento en la posición i, es decir ordena los segmentos a[1..i-i] y a[i+1..N] en forma creciente, dejando al elemento en la posición intacto.

Precondición necesaria para que el algoritmo logre hacerlo: $2 \leq i \leq N-1$, es decir el parámetro i debe ser una posición válida dentro del arreglo.

b) ¿COMO LO HACE?

➔ Algoritmo q

Es un selection_sort en el segmento [1..x] del arreglo: encuentra el mínimo elemento en el segmento [1..x] y lo intercambia con el que se encuentra en la primera posición, encuentra el menor de los restantes y lo intercambia con el que se encuentra en la segunda posición, y así sucesivamente.

➔ Algoritmo r

Es un insertion_sort en el segmento [y..N] del arreglo: 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..j], de forma tal que el segmento [y..j] 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 que inicialmente estaba en la posición j sea menor a este, lo que va a significar que el elemento j está (parcialmente) bien posicionado. Es una especie de insertion sort.

➔ Algoritmo p

Llama al algoritmo q para ordenar el segmento a[1..i-1] y al algoritmo r para ordenar el segmento a[i+1..N].

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

4. Dada la especificación del tad Cola:

spec Queue **of** T **where**

constructors

fun empty_queue() **ret** q : Queue **of** T
{- *crea una cola vacía.* -}

proc enqueue (**in**/**out** q : Queue **of** T, **in** e : T)
{- *agrega el elemento* e *al final de la cola* q. -}

operations

fun is_empty_queue(q : Queue **of** T) **ret** b : Bool
{- *Devuelve* True *si la cola es vacía* -}

fun first(q : Queue **of** T) **ret** e : T
{- *Devuelve el elemento que se encuentra al comienzo de* q. -}
{- **PRE:** not is_empty_queue(q) -}

proc dequeue (**in**/**out** q : Queue **of** T)
{- *Elimina el elemento que se encuentra al comienzo de* q. -}
{- **PRE:** not is_empty_queue(q) -}

Implementá los constructores y operaciones del TAD utilizando la siguiente representación, donde N es una constante de tipo nat:

implement Queue **of** T **where**

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

```
implement Queue of T where

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

constructors
     fun empty_queue() ret q: Queue of T
          q.size := 0
     end fun

     {- PRE: q.size < N (tiene que haber lugar en el arreglo para agregar un
     nuevo elemento a la cola) -}
```

```
    {- En mi implementación, el último elemento de la cola es el de más a la
    derecha. -}
    proc enqueue(in/out q: Queue of T, in e: T)
         q.elems[q.size] := e
         q.size := q.size + 1
    end proc

operations
    fun is_empty_queue(q: Queue of T) ret b: bool
         b := (q.size = 0)
    end fun

    {- PRE: not is_empty_queue(q) -}
    {- Recordar que en esta implementación el primer elemento de la cola es
    el de más a la izquierda. -}
    fun first(q: Queue of T) ret e: T
         e := q.elems[0]
    end fun

    {- PRE: not is_empty_queue(q) -}
    {- Como en mi implementación el primer elemento de la cola es el que en
    el arreglo está más a la izquierda, para desencolarlo simplemente tengo
    que mover todos los elementos del arreglo un lugar a izquierda. -}
    proc dequeue(in/out q: Queue of T)
         for i := 0 to q.size-2 do
              q.elems[i] := q.elems[i+1]
         od
         q.size := q.size - 1
    end proc

end implementation
```