EXAMEN FINAL 24 DE JULIO DE 2019

EJERCICIO 1

1. Tipos de datos.

   (a) **Especificar** el TAD Urna que permita registrar votos para una elección entre dos partidos (partido X y partido Y). El TAD debe permitir las siguientes operaciones: urna vacía, votar X, votar Y, votar en blanco, juntar dos urnas, averiguar si la urna está vacía, cuál es el número total de votos, si hay al menos un voto X, si hay al menos un voto Y, anular un voto X, anular un voto Y, averiguar si gana X, si gana Y, si empatan.

   (b) **Implementar** el TAD Urna utilizando una representación que le resulte conveniente, de manera de que todas las operaciones sean constantes.

a) ESPECIFICACIÓN DEL TAD URNA

```
spec Urna of T where

constructors
     fun vacia() ret u: Urna of T
     {- Retorna una urna vacía -}

     proc votarX(in/out u: Urna of T)
     {- Suma un voto al partido X en la urna u -}

     proc votarY(in/out u: Urna of T)
     {- Suma un voto al partido Y en la urna u -}

     proc votarEnBlanco(in/out: Urna of T)
     {- Suma un voto en blanco en la urna u -}

destroy
     proc destroyUrna(u: Urna of T)
     {- Libera la memoria usada por la urna u en caso de ser necesario -}

operations
     proc juntarUrnas(in/out u: Urna of T, in u0: Urna of T)
     {- Pone los votos que hubo en la urna u0 en la urna u -}

     fun esVacia(u: Urna of T) ret b: bool
     {- Devuelve true si la urna u está vacía -}

     fun votosTotales(u: Urna of T) ret n: nat
     {- Calcular el número total de votos que hay en la urna u -}

     fun hayVotoX(u: Urna of T) ret b: bool
     {- Devuelve true si en la urna u hay al menos un voto para el partido X
     -}
```

```
    fun hayVotoY(u: Urna of T) ret b: bool
    {- Devuelve true si en la urna u hay al menos un voto para el partido
    Y-}

    proc anularVotoX(in/out u: Urna of T)
    {- Anula en la urna u un voto al partido X -}

    proc anularVotoY(in/out u: Urna of T)
    {- Anula en la urna u un voto al partido Y -}

    fun ganaX(u: Urna of T) ret b: bool
    {- Devuelve true si en la urna u el partido X tiene más votos que el
    partido Y -}

    fun ganaY(u: Urna of T) ret b: bool
    {- Devuelve true si en la urna u el partido Y tiene más votos que el
    partido X -}

    fun hayEmpate(u: Urna of T) ret b: bool
    {- Devuelve true si en la urna u la cantidad de votos para ambos
    partidos X e Y es la misma -}

end spec
```

b) IMPLEMENTACIÓN DEL TAD URNA

```
implement Urna of T where

type Urna of T = tuple
                    votosX: nat
                    votosY: nat
                    enBlanco: nat
                end tuple

constructors
    fun vacia() ret u: Urna of T
        u.votosX := 0
        u.votosY := 0
        u.enBlanco := 0
    end fun

    proc votarX(in/out u: Urna of T)
```

```
          u.votosX := u.votosX + 1
     end proc

     proc votarY(in/out u: Urna of T)
          u.votosY := u.votosY + 1
     end proc

     proc votarEnBlanco(in/out: Urna of T)
          u.enBlanco := u.enBlanco + 1
     end proc

destroy
     proc destroyUrna(u: Urna of T)
          skip
     end proc

operations
     proc juntarUrnas(in/out u: Urna of T, in u0: Urna of T)
          u.votosX := u.votosX + u0.votosX
          u.votosY := u.votosY + u0.votosY
          u.enBlanco := u.enBlanco + u0.enBlanco
     end proc

     fun esVacia(u: Urna of T) ret b: bool
          b := (u.votosX = 0) && (u.votosY = 0) && (u.enBlanco = 0)
     end fun

     fun votosTotales(u: Urna of T) ret n: nat
          n := u.votosX + u.votosY + u.enBlanco
     end fun

     fun hayVotoX(u: Urna of T) ret b: bool
          b := u.votosX > 0
     end fun

     fun hayVotoY(u: Urna of T) ret b: bool
          b := u.votosY > 0
     end fun

     proc anularVotoX(in/out u: Urna of T)
          if u.votosX != 0 then
               u.votosX := u.votosX - 1
          fi
```

```
    end proc

    proc anularVotoY(in/out u: Urna of T)
        if u.votosV != 0 then
            u.votosV := u.votosV - 1
        fi
    end proc

    fun ganaX(u: Urna of T) ret b: bool
        b := u.votosX > u.votosY
    end fun

    fun ganaY(u: Urna of T) ret b: bool
        b := u.votosY > u.votosX
    end fun

    fun hayEmpate(u: Urna of T) ret b: bool
        b := u.votosX = u.votosY
    end fun
end implementation
```

EJERCICIO 2

2. 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(in/out a : array[1..n] of int)
    var i : nat
    i:= 1
    while i <= n do
        swap(a,i,q(a,i))
        i:= i+1
    od
end proc
```

```
fun q(a : array[1..n] of int, i : nat) ret j: nat
    var m,k : nat
    j:= i
    if i mod 2 == 0 then
        m:= a[i]
        k:= i+2
        while k ≤ n do
            if a[k] < m then
                m:= a[k]
                j:= k
            fi
            k:= j+2
        od
    fi
end fun
```

a) ¿QUE HACEN?

➔ Función q

Dado un arreglo de enteros a[1..n] y un natural i:

- Si i es impar, devuelve i.

- Si i es par (que es el caso más interesante), devuelve el índice del elemento mínimo de entre los elementos que se encuentran en posiciones pares a partir de la posición i en el arreglo a.

➔ Procedimiento p

Dado un arreglo de enteros a[1..n], ordena los elementos que se encuentran en posiciones pares en forma creciente, es decir de menor a mayor. Los elementos que se encuentran en posiciones impares se mantienen intactos.

b) ¿COMO LO HACEN?

➔ Función q

Como ya dije antes, si el argumento i es impar, entonces la función simplemente devuelve i, pues no se cumple la guarda del if.

Ahora bien, si i es par, entonces el algoritmo comienza tomando como elemento mínimo del segmento a[i..n] al elemento en la posición i. A partir de allí, recorre el arreglo de izquierda a derecha fijándose solamente en los que se encuentran en posiciones pares, guardando en j el índice del elemento mínimo.

➔ Procedimiento p

Es básicamente un selection sort sobre las posiciones pares:

Cuando i es impar, el arreglo no se modifica, pues ya vimos que q(a,i) = i, y por lo tanto swap(a,i,q(a,i)) = swap(a,i,i), que no introduce ningún cambio sobre el arreglo.

Cuando i es par, lo que ocurre es: se llama a la función q para calcular la posición del mínimo elemento de entre los que están en posiciones pares, y lo intercambia con el elemento que se encuentra en la posición i (par). Más detalladamente:

Primero (cuando i=2), halla el elemento mínimo (de entre los que están en posiciones pares) del segmento a[2..n], y lo intercambia con el elemento que está en la posición 2.

Luego (cuando i=4), halla el elemento mínimo de los restantes (como siempre, de entre los que están en posiciones pares), es decir del segmento a[4..n], y lo intercambia con el elemento que está en la posición 4.

Así sucesivamente, ordenando en cada iteración con una llamada a la función q con un i par a uno de los elementos que se encuentran en las posiciones pares.

c) ORDEN DE LOS ALGORITMOS

➔ Función q

Veamos los distintos casos:

Si i es impar, se hace solamente la asignación j:=i y no se entra al if. Por lo tanto, si i es impar, la función q es de orden 1 (constante).

Si i es par, se cumple la guarda del if y lo que va a dar el orden del algoritmo va a ser la cantidad de veces que se ejecute el ciclo. Veamos que antes de entrar al while, se hace la asignación k:=i+2, y la guarda del ciclo es while $k \leq n$. Además, dentro del ciclo, a k se lo va incrementando de a 2. Por lo tanto, el ciclo se va a ejecutar hasta que $k = n + 2$. Es decir, la cantidad de veces que se va a ejecutar el ciclo es la cantidad de veces que haga falta sumarle 2 a k para que llegue a “ser” $n + 2$. Es decir:

n + 2 = k + 2*ops => n + 2 - k = 2*ops => n + 2 - (i + 2) = 2*ops => (n - i)/2 = ops

El ciclo se ejecutará (n-i)/2 veces.

Luego, siempre teniendo en cuenta que i es par, tenemos:

Mejor caso: i = n ----> el algoritmo es de orden 0

Peor caso: i = 0 -----> el algoritmo es de orden n/2

- ➔ Procedimiento p

El ciclo while se ejecuta n veces, ya que se comienza desde 1 y la variable i se incrementa en 1 en cada iteración (esto tranquilamente se podría haber hecho con un for).

Ahora bien, en cada iteración se llama a la función q para $i \in \{1,2,3,...,n\}$.

Ya vimos que cuando i es impar, la función q es de orden 1.

Por otro lado, cuando i es par, la función q es de orden (n-i)/2.

Entonces, tenemos que la cantidad de operaciones del procedimiento p es (suponiendo que n es par; el caso de n impar es análogo) :

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

$= \frac{n^2}{4} - \frac{\frac{1}{2} * n/2 * (n/2+1)}{2} + \frac{n}{2} = ... = O(n^2)$

Es decir, el procedimiento p es de orden n^2.

d) NOMBRES MÁS ADECUADOS

- ➔ Para la función q:
  - q ------> min_even_pos_from
  - j -------> min_pos
  - m -----> current_min
- ➔ Para el procedimiento p:
  - p ------> even_selection_sort

EJERCICIO 3

3. Miguel Hernández decide invitar a sus amigos/as a un asado. Como vive en un departamento, planifica la realización del asado en la casa de su amiga Josefina Manresa. "Ché, ahí invité a toda la barra. Nos vemos hoy en tu casa para comer el asado", le dice. "Estás loco, che?" le responde Josefina, "tengo la casa hecha un kilombo. Me hubieras avisado." "No te preocupes, voy para allá y te doy una mano". Y sale para allá.

   Cuando Miguel llega a lo de su amiga encuentra a Josefina en el patio preparando el fuego y salando la carne, así que la saluda y se dispone a acomodar la casa antes de que llegue la gente. No puede creer el desorden que encuentra al entrar a la casa. Con un simple vistazo, estima que cada uno de los N ambientes de la casa le va a insumir un tiempo $t_1, t_2, \ldots, t_N$ acomodar y que la valoración que va a recibir por la tarea realizada es $v_1, v_2, \ldots, v_N$. También descubre que el tiempo total T de que dispone hasta que vengan los/as amigos/as no es suficiente para acomodar todos los ambientes, pero sabe que si acomoda parcialmente un ambiente, obtiene el reconocimiento proporcional. Es decir, a modo de ejemplo, si ordena $\frac{2}{5}$ del ambiente i, eso le lleva un tiempo $\frac{2}{5}t_i$ y le genera una valoración $\frac{2}{5}v_i$.

   Se desea escribir un algoritmo que encuentre la mayor valoración total a recibir por el trabajo de acomodar los ambientes realizado en el tiempo T.

   (a) Describí cuál es el criterio de selección.

   (b) ¿En qué estructuras de datos representarás la información del problema?

   (c) Explicá el algoritmo, es decir, los pasos a seguir para obtener el resultado. No se pide que "leas" el algoritmo ("se define una variable x", "se declara un for"), si no que lo expliques ("se recorre la lista/el arreglo/", "se elije de tal conjunto el que satisface...", "se repite lo anterior mientras haya ...", etc.).

   (d) Escribí el algoritmo en pseudocódigo.

Datos:

- Tenemos N ambientes
- Cada ambiente i le va a tomar tiempo t_1, t_2,..., t_N acomodar y la valoración que va a recibir por la tarea realizada es v_1, v_2,..., v_N.

Se desea obtener la mayor valoración total a recibir por el trabajo de acomodar los ambientes realizado en el tiempo T.

a) CRITERIO DE SELECCIÓN

En cada momento, elijo acomodar el que tenga mayor cociente valoración/tiempo, pues al elegir ese me estoy asegurando que por cada unidad de tiempo que pase ordenando ese ambiente, voy a obtener la mejor valoración que si hubiera elegido otro ambiente.

b) ESTRUCTURAS DE DATOS

Representaré a cada ambiente mediante una tupla:

```
type Ambiente = tuple
                    id: nat
                    tiempo: nat
                    valoracion: nat
                    frac: nat
                end tuple
```

donde:

id es el identificador del ambiente, expresado en números

tiempo es el tiempo que lleva ordenar por completo ese ambiente
valoracion es la valoración que obtendría por ordenar ese ambiente por completo
frac representa la fracción que voy a ordenar de ese ambiente (si frac=1, quiere decir que lo ordeno entero)

Todos los ambientes de la casa estarán representado por un conjunto de elementos del tipo Ambiente, donde para todos ellos se asume que frac=1.
Y los elementos del conjunto solución también serán de tipo Ambiente.

c) EXPLICACIÓN DEL ALGORITMO

En cada momento, se elige del conjunto de ambientes aquel que tenga mayor cociente entre valoración y tiempo, es decir aquel que satisfaga el criterio de selección explicado anteriormente.
Si el tiempo restante que tengo me alcanza para ordenar ese ambiente entero, entonces lo ordeno entero, lo elimino del conjunto de ambiente que restan por ordenar, y lo agrego al conjunto solución.
Si el tiempo restante NO me alcanza para ordenarlo por completo, ordeno parcialmente ese ambiente, más específicamente ordeno tiempo_restante/tiempo_que_lleva_ordenarlo partes de ese ambiente, lo elimino del conjunto de ambientes que restan por ordenar, y lo agrego al conjunto solución. (obviamente indicando cuánto tengo que ordenar de este ambiente).
Se repite lo anterior mientras que el tiempo restante que me queda para ordenar ambientes sea mayor que 0 o hasta que ya estén todas ordenadas.
OBSERVACIÓN: la función va a devolver un conjunto con los ambientes que se deben ordenar para obtener la máxima valoración posible.

Más corto: Se elige la habitación óptima, y se la va a ordenar lo más posible dado el tiempo restante. Esto se va a repetir con las restantes hasta que ya estén todas ordenadas o hasta que no tenga más tiempo.

d) IMPLEMENTACIÓN DEL ALGORITMO

```
fun ambientesVoraces(ambientes: Set of Ambiente, T: nat) ret amb: Set of Ambiente
     var ambientes_aux: Set of Ambiente
     var tiempo_restante: nat
     var ambiente_elegido: Ambiente

     ambientes_aux := copy_set(ambientes)
     tiempo_restante := T

     amb := empty_set()

     while tiempo_restante > 0 and not is_empty_set(ambientes) do
          ambiente_elegido := selecAmbiente(ambientes_aux)
          if ambiente_elegido.tiempo ≤ tiempoRestante then
               ambiente_elegido.frac := 1
               tiempo_restante := tiempo_restante - ambiente_elegido.tiempo
```

```
          else
               ambiente_elegido.frac                                          :=
tiempo_restante/ambiente_elegido.tiempo
               tiempo_restante := 0
          fi
          elim(ambientes_aux, ambiente_elegido)
          add(amb, ambiente_elegido)
     od

     destroy(ambientes_aux)
end fun

fun selecAmbiente(a: Set of Ambiente) ret ambiente: Ambiente
     a_aux: Set of Ambiente
     ambiente_aux: Ambiente
     cocMax: nat
     var cociente: nat

     a_aux := copy_set(a)
     cocMax := 0

     while not is_empty_set(a_aux) do
          ambiente_aux := get(a_aux)
          cociente := ambiente_aux.valoracion/ambiente_aux.tiempo
          if cocMax < cociente then
               cocMax := cociente
               ambiente := ambiente_aux
          fi
          elim(a_aux, ambiente_aux)
     od

     destroy(a_aux)
end fun
```

EJERCICIO 4

4. En una localidad cordobesa, vive el José Agustín Goytisolo quien, apenado por los padecimientos de la mayoría de los vecinos, decide comprometerse en solucionarlos postulándose a la intendencia. Gracias a su entusiasmo y creatividad, en pocos minutos enumera una larga lista de N propuestas para realizar. Pronto descubre que a pesar de que cada una de ellas generaría una satisfacción popular $p_1, p_2, \ldots, p_N$ también provocaría desagrado $q_1, q_2, \ldots, q_N$ en el sector más acomodado de la sociedad local. En principio, el desagrado de cada propuesta es insignificante en número de votos ya que la alta sociedad no es muy numerosa. Pero a José le interesa cuidar su relación con este sector, ya que el mismo tiene suficientes recursos como para dificultar su triunfo en caso de proponérselo.

   Pronto descubre que las propuestas elaboradas son demasiadas para ser publicitadas: tantas propuestas (N) generarían confusión en el electorado. Esto lo lleva a convencerse de seleccionar solamente K de esas N propuestas ($K \leq N$). Se dispone, entonces, a seleccionar K de esas N propuestas de forma tal que la suma de satisfacción popular de las K propuestas elegidas sea máxima y que el descontento total de esas K propuestas en la alta sociedad no supere un cierto valor M.

   José Agustín te contrata para que desarrolles un algoritmo capaz de calcular el máximo de satisfacción popular alcanzable con K de esas N propuestas sin que el descontento supere M.

Datos:

- N propuestas
- Cada propuesta i generaría una satisfacción popular p_i pero también provocaría un desagrado q_i.

Se desea calcular el máximo de satisfacción popular que se puede alcanzar con K de las N propuestas sin que el descontento supere M.

FUNCIÓN RECURSIVA

propuestas(i,k,m) = "máxima **satisfacción** alcanzable eligiendo k propuestas entre las propuestas 1..i, sin que la suma de los descontentos de las propuestas elegidas supere m"

Rol de los argumentos:

El argumento i indica hasta qué propuesta estamos considerando.
El argumento k indica la cantidad de propuestas que quedan por elegir.
El argumento m indica el máximo descontento que se puede alcanzar.

LLAMADA PRINCIPAL

```
propuestas(N,K,M)
```

FUNCIÓN MATEMÁTICA

```
propuestas(i,k,m) =
{- CASOS BASE -}
{- Caso base 1: tengo que elegir m>0 propuestas de entre 0 propuestas. No hay solución. -}
|i = 0 && k > 0 -------------> -infinito

{- Caso base 2: tengo que elegir 0 propuestas de entre 0 (o más) propuestas. Es claro que eligiendo 0 propuestas, la máxima satisfacción alcanzable es 0. -}
|i ≥ 0 && k = 0 -------------> 0
```

{- CASOS RECURSIVOS -}

{- Caso recursivo 1: si el descontento que generaría elegir la propuesta i es mayor al descontento que no puedo superar, entonces no considero esa propuesta -}

$|i > 0 \&\& k > 0 \&\& q_i > m \text{ ----> } propuestas(i-1, k, m)$

{- Caso recursivo 2: eligiendo la propuesta i no se supera el descontento permitido, entonces pruebo elegirla y no elegirla -}

$|i > 0 \&\& k > 0 \&\& q_i \leq m \text{ ----> } max(p_i + propuestas(i-1,k-1,m-q_i), propuestas(i-1,k,m))$