f EXAMEN FINAL 8 DE FEBRERO DE 2021

EJERCICIO 1

1. (Voraz) Anacleta reparte pedidos en bicicleta. Todos los clientes viven en la Avenida San Wachín y la dirección del trabajo de Anacleta es Avenida San Wachín 0. Al llegar a trabajar, Anacleta tiene cada pedido con la altura de la casa donde debe entregarlo. En la bici, Anacleta puede transportar cinco pedidos o menos. Escriba un algoritmo que reciba una lista de naturales llamada pedidos y determine la menor distancia que debrá pedalear Anacleta para entregar todos los pedidos.

Primero, analizo el enunciado:

- Todos los clientes viven en la misma avenida San Wachín.
- La dirección del trabajo de Anacleta es Avenida San Wachín 0.
- Anacleta tiene cada pedido con la altura de la casa donde debe entregarlo. Cada pedido es la dirección del lugar (= a la distancia al negocio).
- En la bici, Anacleta puede transportar cinco pedidos o menos.

Se desea determinar la MENOR DISTANCIA que deberá pedalear Anacleta para entregar todos los pedidos.

CRITERIO DE SELECCIÓN

En cada momento, Anacleta elige entregar el pedido que le queda más lejos. Cuando va a entregar ese pedido, entrega también los 4 o menos pedidos “anteriores” que están más cerca de ese, y después vuelve a su trabajo a buscar más.
Para la justificación de este criterio, pensar qué hubiera pasado si entregaba primero el más cercano (y los 4 o menos que le seguían) en una situación en donde tengo que entregar un número NO múltiplo de 5 (mayor que 5).

IMPLEMENTACIÓN

```
fun menorDistancia (pedidos: array[1..N] of nat) ret dist: nat
     var pedidos_aux: array[1..N] of nat
     var masLejano: nat
     var i: nat

     pedidos_aux := copy_array(pedidos)

     {- Ordeno el arreglo de mayor a menor, poniendo los pedidos que quedan
     más lejos primero -}
     reverse_sort(pedidos_aux)

     dist := 0
     i := 1

     while i ≤ N do
          {- Por como ordené el arreglo, el pedido más lejos es el que está
          primero (pensando en la primera iteración, en las próximas es lo
          mismo) -}
```

```
            masLejano := pedidos[i]

            {- recorre ida y vuelta la distancia del pedido que está más lejos
-}
            dist := dist + 2*masLejano

            {- Entregando el pedido más lejano, a la vuelta entregué los 4
            pedidos que estaban antes de él, entonces el próximo pedido más
            lejano a entregar es el que está inmediatamente antes de estos 5 -}
            i := i + 5
      od
end fun
```

EJERCICIO 2

2. (Backtracking) Se tienen n objetos de peso $p_1, \ldots, p_n$ respectivamente. Se tiene una mochila de capacidad K. Dar un algoritmo que utilice backtracking para calcular el menor desperdicio posible de la capacidad de la mochila, es decir, aquél que se obtiene ocupando la mayor porción posible de la capacidad, sin excederla. Definir primero en palabras la función aclarando el rol de los parámetros o argumentos.

DATOS

- Tenemos n objetos
- Cada objeto i tiene un peso p_i
- Mochila de capacidad K

Se desea obtener el menor desperdicio posible de la capacidad de la mochila, es decir aquel que se obtiene ocupando la mayor porción posible de la capacidad, sin excederla.

ESPECIFICACIÓN DE LA FUNCIÓN (EXPLICACIÓN, TIPO Y ARGUMENTOS)

mochila :: Nat -> Nat -> Nat

mochila(i, j) = “menor desperdicio posible que se puede poner en la mochila poniendo algunos de los elementos entre 1 e i (o todos) cuando la capacidad restante de la mochila es j”

Rol de los argumentos:

El argumento i indica hasta qué objeto estamos considerando.
El argumento j indica la capacidad restante de la mochila.

LLAMADA PRINCIPAL

mochila(n, K)

FUNCIÓN RECURSIVA EN NOTACIÓN MATEMÁTICA

mochila(i, j) =

{- CASO BASE -}
{- No tengo más objetos para poner en la mochila. -}

|i=0 -------> j

{- CASO RECURSIVO (el correcto) -}
{- Caso recursivo 1: el peso del objeto i supera la capacidad disponible de la mochila. Es decir, si pusiera este objeto i, excedería la capacidad de la mochila. Entonces, no lo considero. -}
|i > 0 && j > 0 && p_i > j --------> mochila(i-1,j)

{- Caso recursivo 2: el peso del objeto i NO supera la capacidad disponible de la mochila. Entonces, pruebo meterlo y pruebo no meterlo. -}
|i > 0 && j > 0 && $p_i \leq j$ --------> min(mochila(i-1, j-p_i), mochila(i-1,j))

{- CASO RECURSIVO MAL -}
{- Caso recursivo 1: el peso del objeto i supera la capacidad disponible de la mochila. Es decir, si pusiera este objeto i, excedería la capacidad de la mochila. Entonces, no lo considero. -}
|i > 0 && j > 0 && p_i > j --------> mochila(i-1,j)

{- Caso recursivo 2: el peso del objeto i NO supera la capacidad disponible de la mochila. Entonces, pruebo meterlo y pruebo no meterlo. -}
|i > 0 && j > 0 && $p_i \leq j$ --------> j - max(p_i + mochila(i-1,j-p_i), mochila(i-1,j))
min(j - (p_i + mochila(i-1, j-p_i), mochila(i-1,j)) {- opción inicial -}
min((j - p_i) + mochila(i-1, j-p_i), mochila(i-1,j)) {- otra opción -}
(por ahora ninguna anda)

p_i → Esto está representando el peso de un objeto que voy a guardar en la mochila, espacio ocupado
mochila(i-1,j-p_i) → Esto por la definición de la función representa espacio desperdiciado, espacio libre

EJEMPLO:
Capacidad total de la mochila: 12
3 objetos:
Objeto 1: 6
Objeto 2: 5
Objeto 3: 15

mochila(3, 12)
= mochila(2,12)
= 12 - max(5 + mochila(1, 7), mochila(1, 12))
= 12 - max(5 + 0, 0)
= 12 - 5

mochila(1,7) (debería dar 1, pues la capacidad del objeto 1 es 6)
= 7 - max(6 + mochila(0,1), mochila(0,7))

= 7 - max(6 + 1, 7)
= 7 - max(7, 7)
= 0

con la opción inicial:
= min(7 - (6 + mochila(0, 1), mochila(0, 7))
= min(7 - (6 + 1), 7)
= min(7-7, 7)
= 0

con la otra opción
= min((7-6) + mochila(0, 1), mochila(0,7))
= min(1 + 1, 7)
= 2

opcion Ivan
= min(mochila(0, 1), mochila(0, 7))
= min(1, 7)
= 1

mochila(1, 12)
= 12 - max(6 + mochila(0,6), mochila(0,12))
= 12 - max(6 + 6, 12)
= 12 - max(12,12)
= 0

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 ello?
(b) ¿Cómo lo hace?
(c) El orden del algoritmo, analizando los distintos casos posibles.
(d) Proponer nombres más adecuados para las funciones.

```
fun s(v: nat, p: array[1..n] of nat) ret y: nat
    y:= v
    while y < n ∧ p[y] ≤ p[y+1] do
        y:= y+1
    od
end fun
```

```
fun t(p: array[1..n] of nat) ret y: nat
    var z: nat
    y, z:= 0, 1
    while z ≤ n do
        y, z:= y+1, s(z,p)+1
    od
end fun
```

```
fun u(p: array[1..n] of nat) ret v: bool
    v:= (t(p) ≤ 1)
end fun
```

a) ¿QUE HACEN? ¿CUALES SON LAS PRECONDICIONES NECESARIAS PARA ELLO?

➔ Algoritmo s

El algoritmo s, dado un natural v y un arreglo p[1..n] de naturales, calcula el índice hasta donde el arreglo p es creciente comenzando desde la posición v, dada como parámetro.
Es decir, devuelve el índice y del segmento de [v..y] del arreglo p, donde [v..y] es creciente.
En otras palabras, devuelve el índice del primer elemento de p tal que es mayor que el elemento siguiente, partiendo de la posición v.
Para que el algoritmo calcule correctamente lo dicho, se necesita como precondición que $1 \leq v \leq n$, es decir que el parámetro v sea una posición válida dentro del arreglo p.

➔ Algoritmo t

El algoritmo t, dado un arreglo p[1..n] de naturales, devuelve la cantidad de segmentos contiguos (en el sentido que no se solapan) que se encuentran en orden creciente (cada uno del máximo largo posible) dentro del arreglo p.
No es necesaria ninguna precondición.

➔ Algoritmo u

El algoritmo u, dado un arreglo p[1..n] de naturales, devuelve true si el arreglo está ordenado de forma creciente.

b) ¿CÓMO LO HACEN?

➔ Algoritmo s

Recorre el arreglo p de izquierda a derecha a partir de la posición v, y avanza incrementando en uno la variable hasta que encuentra el primer elemento que es mayor que el siguiente.

➔ Algoritmo t

Recorre el arreglo en segmentos ordenados crecientemente, usando el algoritmo s para determinar dónde termina cada segmento, contando con la variable y la cantidad de segmentos ordenados crecientemente recorridos.

➔ Algoritmo u

Llama a t y se fija si el resultado es ≤ 1 (de todas formas, nunca va a dar 0), lo que significa que en el arreglo p hay exactamente un segmento de largo máximo posible ordenado crecientemente, que es justamente todo el arreglo.

c) ORDEN DE LOS ALGORITMOS

➔ Algoritmo s

Aquí, la operación representativa es la comparación entre elementos del arreglo p. Podemos tener dos casos:

MEJOR CASO: cuando el elemento en la posición v es (estrictamente) mayor al elemento siguiente. En este caso, se realiza una única comparación.
Luego, el mejor caso es de orden O(1).

PEOR CASO: cuando el arreglo está ordenado desde la posición v hasta el final. Aquí, se realizan tantas comparaciones como veces haya que incrementar (en uno) la variable y para que se termine el ciclo, esto es n-v comparaciones.
Es decir, el peor caso es de orden O(n-v).

➔ Algoritmo t

Nuevamente, aquí la operación representativa es la comparación entre elementos del arreglo p, que está "dentro" de la llamada al algoritmo s en cada iteración. También podemos reconocer dos casos:

MEJOR CASO: cuando el arreglo está todo ordenado crecientemente. Se llama una vez al algoritmo s (con v:=1), y por su resultado (=n) se terminan el ciclo y el programa. Se hacen n-1 comparaciones.
Luego, el mejor caso es de orden O(n-1).
PEOR CASO: cuando el arreglo está todo ordenado decrecientemente. Se llama al algoritmo s una vez por cada índice del arreglo. Como el arreglo está todo ordenado decrecientemente, en cada llamada al algoritmo s solo se realiza una comparación (salvo en el caso en que z = n), que no se realiza ninguna. Es decir que en total se realizan n-1 comparaciones.
Luego, el peor caso también es de orden O(n-1).

➔ Algoritmo u

Como este algoritmo simplemente llama a t, es del mismo orden que t. Por lo tanto, el orden del algoritmo u es O(n).

d) NOMBRES MÁS ADECUADOS

```
fun sorted_till_from (from: nat, p: array[1..n] of nat) ret till: nat
     till := from
     while till < n && p[till] ≤ p[till+1] do
          till := till + 1
     od
end fun

fun sorted_subarrays (p: array[1..n] of nat) ret amount: nat
     var subarray_start: nat
     amount, subarray_start := 0,1
     while subarray_start ≤ n do
          amount,subarray_start:=  amount  +1,sorted_till_from(subarray_start,
p) +1
     od
end fun

fun is_sorted (p: array[1..n] of nat) ret res: bool
     res := (sorted_subarrays(p) ≤ 1)
```

end fun

EJERCICIO 4

4. Completar la especificación del tipo Conjunto de elementos de tipo T, agregando las operaciones
   - **elim**, que elimina de un conjunto un elemento dado.
   - **union**, que agrega a un conjunto todos los elementos que pertenecen a otro conjunto dado.
   - **dif**, que elimina de un conjunto todos los elementos que pertenecen a otro conjunto dado.

```
spec Set of T where
(...)

operations
(...)

    proc elim (in/out s: Set of T, in e: T)
    {- Elimina al elemento e del conjunto s -}

    proc union (in/out s: Set of T, in s0: Set of T)
    {- Agrega al conjunto s todos los elementos que pertenecen exclusivamente
    al conjunto s0 (recordar que en un conjunto no puede haber elementos
    repetidos). -}

    proc dif (in/out s: Set of T, in s0: Set of T)
    {- Elimina del conjunto s todos los elementos de s que pertenecen al
    conjunto s0. -}

end spec
```

EJERCICIO 5

5. A partir de la siguiente implementación de conjuntos utilizando listas ordenadas, implemente el constructor **add**, y las operaciones **get**, **inters** y **elim**. La implementación debe mantener el invariante de representación por el cual todo conjunto está representado por una lista ordenada crecientemente. Puede utilizar todas las operaciones especificadas para el tipo lista vistas en el teórico. Para cada operación que utilice, especifique su encabezado, es decir: si es función o procedimiento, cómo se llama, qué argumentos toma y devuelve.

```
implement Set of T where

type Set of T = List of T

fun empty_set() ret s : Set of T
    s := empty_list()
end fun
```

➔ Constructor add

```
proc add (in e: T, in/out s: Set of T)
{- IDEA: tengo que lograr que una vez añadido el elemento e al conjunto s se
siga manteniendo el invariante de representación, es decir el conjunto debe
estar representado por una lista ordenada crecientemente.
Para ello, lo que debo hacer es fijarme si el elemento e es mayor, igual o
menor que cada elemento del conjunto:
     Si e es mayor, entonces sigo avanzando hasta encontrar un elemento que
     sea mayor a e.
     Si e es igual, entonces no hago nada, pues en los conjuntos no puede
     haber repeticiones.
     Si e es menor, entonces lo agrego en ese lugar. -}

     var s_aux: Set of T
     var i: nat

     i := 1
     s_aux := copy_list(s)
     while not is_empty_set(s_aux) && head(s_aux) < e do
          tail(s_aux)
          i := i + 1
     od
     if is_empty_set(s_aux) v head(s_aux) > e then
          add_at(s,i,e)
     fi
     destroy(s_aux)
end proc
```

➔ Operación get

```
fun get(s: Set of T) ret e: T
{- La especificación de Set of T dice que la operación get devuelve algún
elemento (cualquiera) de s. En esta implementación, yo elijo que este
elemento que devuelve sea el primero de la lista con la que se representa al
conjunto s. -}
     e := head(s)
end fun
```

➔ Operación inters

```
proc inters (in/out s: Set of T, in s0: Set of T)
     var s_aux: Set of T

     s_aux := copy_list(s)

     while not is_empty_set(s_aux) do
          elem := head(s_aux)
          if not member(elem, s0) then
               elim(s, elem)
          fi
          tail(s_aux)
     od
end proc
```

➔ Operación elim

```
proc elim (in/out s: Set of T, e: T)
     var s_aux: Set of T

     s_aux := copy_list(s)

     while not is_empty_set(s_aux) && head(s_aux) != e do
          tail(s_aux)
     od
     if head(s_aux) = e then
          tail(s_aux)
          inters(s, s_aux) ¿?
     fi
     destroy(s_aux)
end proc
```