PEC2: Segunda Prueba de Evaluación Continuada
Por: Álvaro Giménez Gorrís

Ejercicio 1: Conceptos básicos de recursividad y tipos abstractos de datos (20%)

Tarea: Responde las preguntas siguientes justificando las respuestas:

i) . ¿En qué principio se basa la recursividad? Explica como se demuestra este principio.

La recursividad se basa en el principio de inducción. El principio de inducción consiste en demostrar que si una propiedad es válida para el caso más simple (por ejemplo, n=1), también lo será para n + 1. Para poder demostrarlo necesitamos un caso base para el que ya sabemos el resultado (n = 0).

En un algoritmo recursivo, la función se llamará a sí misma hasta que se dé la condición del caso base (para el que ya sabemos el resultado), que hará que el algoritmo se pare.

ii) Cuando utilizamos memoria dinámica, ¿cómo se encadenan los diferentes nodos del TAD?

Para encadenar los diferentes nodos del TAD se utilizan punteros. Cada nodo guarda en su interior un puntero que guarda la dirección de memoria del siguiente nodo del TAD.

iii) Los punteros p1 y p2 apuntan a 2 objetos. ¿Por qué motivo no es suficiente para compararlos, utilizar p1=p2? Puedes ayudarte con un ejemplo.

La comparación de dos objetos apuntados por dos punteros no se puede reducir a la comparación de espacios de memoria, ya que, si un puntero contiene punteros a otra estructura, este se quedaría fuera de la comparación.

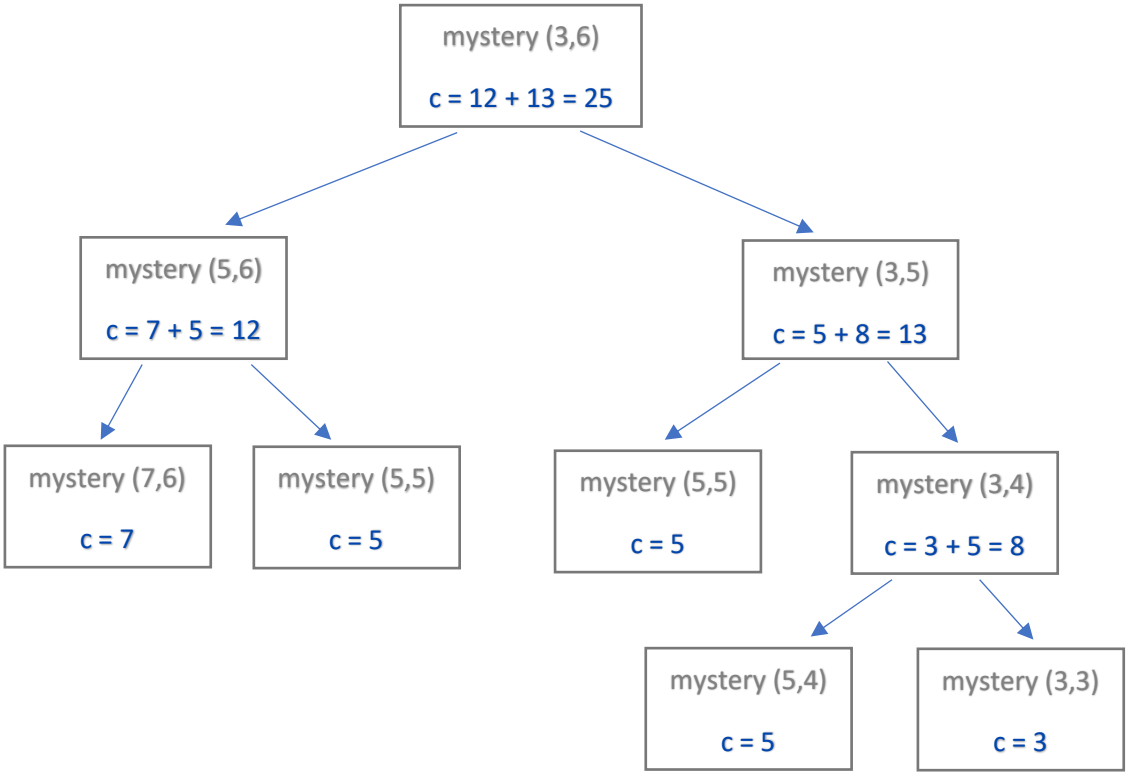
Para poder comparar dos punteros tendremos que hacer la comparación recursiva de manera completa a todos los objetos apuntados por dichos punteros.

iv) Dada la función recursiva **mystery**, calcula que valor devuelve la llamada **mystery(3, 6)** y completa el **modelo de las copias** para ver como has llegado al resultado:

```
function mystery (a : integer, b: integer ) : integer

var

        c : integer;

end var


if a ≥ b then
```

```
        c := a;
else
        c := mystery (a+2, b) + mystery (a, b-1);
end if
return c;
end function
```

El resultado es 25, que resulta de la secuencia de llamadas de la siguiente figura:



Ejercicio 2: Diseño de algoritmos recursivos (20%)

Tarea: Dada la descripción de los problemas siguientes, diseña los algoritmos recursivos que los resuelven. *Consejo: Antes de empezar a escribir cada algoritmo, tienes que identificar los casos base y recursivos.*

i) Diseña la función recursiva **count_char** que devuelve el número de apariciones de un carácter **c** en las **n** primeras posiciones de un vector de caracteres.

Ejemplo: count_char ({'h','e','l','l','o',' ','w','o','r','l','d'}, 'l', 11) devuelve 3.

```
function count_char ( v: vector[MAX] of character, c: character, n: integer) : integer
Pre: { 0 ≤ n ≤ MAX }
var
        res: integer
end var
        if n = 0 then
                res := 0;
        else
                res := count_char (v, c, n – 1);
                if v[n] = c then
                        res := res + 1;
                end if
        end if
        return res;
end function
```

ii) Diseña la acción recursiva **stack_analyzer** que dada una pila **p** de enteros, elimina de esta pila los elementos menores o iguales que 0. Los menores que 0 los coloca, manteniendo el orden original, en una pila nueva **q**. Además, la acción devuelve el número de elementos que son 0.

Ejemplo: **stack_analyzer** ([1, 0, -8, 2, 0, -9, 5, 3>, q, res) , devuelve las pilas modificadas de la siguiente forma: la pila p (primer parámetro): [1, 2, 5, 3> y la pila q: [-8, -9>. El valor de res es 2.

```
action stack_analyzer ( inout p: stack(integer), out q: stack(integer), out res:
integer)
var
        a : integer;
end var
        if emptyStack(p) then
                q := createStack();
                res := 0;
        else
                a := top(p);
                pop(p);
                stack_analyzer(p, q, res);
                if (a <= 0) then
                        if (a < 0) then
```

```
                push(a, q);
            else
                res := res + 1;
            end if
        else
            push(a, p);
        end if

    end if
end action
```

Ejercicio 3: Convertir algoritmos recursivos en iterativos (20%)

Tarea: Dada una acción recursiva transfórmala en iterativa

i) Rellena los recuadros para finalizar el diseño de la función recursiva **sum_squares** que calcula la suma de los n primeros números al cuadrado a partir de un numero d. Ejemplo: **sum_squares** (4,3) devuelve $3^2+4^2+5^2+6^2=86$

```
function sum_squares (n: integer, d: integer): integer
Pre: { 0 < n, 0 < d }
var
    res: integer;
fvar
    if n = 0 then
        res := 0;
    else
        res := (d * d) + sum_squares (n - 1, d + 1);
    end if
    return res;
end function
```

ii) Transforma la función recursiva *sum_squares* en una función iterativa.

```
function sum_squares (n: integer, d: integer): integer
Pre: { 0 < n, 0 < d }
var
    res: integer;
fvar
```

```
        res := 0;
        while n ≠ 0 do
                res := res + d * d;
                d := d + 1;
                n := n – 1;
        end while
        return res;
end function
```

Ejercicio 4: Modificación de TAD básicos (20%)

Tarea: Dada la siguiente implementación del TAD cola (**queue**) con punteros (también explicada en los apuntes):

```
type
        node = record
                e : elem;
        next : pointer to node;
        end record

        queue = record
                first, last : pointer to node;
        end record
end type
```

Extiende el tipo añadiendo las operaciones siguientes:

i) **count_ocurrences**: función que devuelve el número de elementos almacenados en la cola que contienen el elemento m en un node. Para diseñar estas acciones **no puedes utilizar** las operaciones del tipo queue (head, enqueue, dequeue, ...). Así pues, tienes que trabajar directamente con la implementación del tipo que os hemos facilitado en el enunciado.

```
function count_ocurrences (c: queue(elem), m: elem) : integer

        var
                count1 : pointer to node;
                count2 : pointer to node;
                amount : integer;
        end var
```

```
        amount := 0;
        count1 := q.first;
        count2 := q.last;

        while(count1 ≠ count2) do
                count1 := count1^.next;
                amount := amount + 1;
        end while
        return amount;
end function
```

ii) **intersect_queue**: función que, dadas dos colas, compara si el valor de las dos colas en una cierta posición son iguales. Es decir, realiza la intersección de los elementos que están en la misma posición. La función devuelve una nueva cola con los elementos que coinciden, respetando el orden de comprobación. Para diseñar esta función **debes utilizar** las operaciones del tipo queue. Es decir, esta vez desconoces como se han implementado internamente este tipo.

```
function intersect_queue (q1: queue(elem), q2: queue(elem)): queue (elem)
        var
                q3 : queue;
                head1 : elem;
                head2 : elem;
                e1 : elem;
                e2 : elem;
        end var

        q3 := createQueue();
        head1 := head (q1);
        head2 := head (q2);

        if (head1 = head2) then
                enqueue (q3, head1);
        end if

        dequeue (q1);
        dequeue (q2);
        enqueue (q1, head1);
        enqueue (q2, head2);

        e1 := head (q1);
        e2 := head (q2);
```

```
        while head1 ≠ e1 and head2 ≠ e2 do
                if (e1 = e2) then
                        enqueue (q3, e1);
                end if

                dequeue (q1);
                dequeue (q2);
                enqueue (q1, e1);
                enqueue (q2, e2);

                e1 := head (q1);
                e2 := head (q2);

        end while

        return q3;
end function;
```

Ejercicio 5: Diseño de un tipo con punteros (20%)

Tarea: Hasta ahora hemos trabajado con los tipos de datos abstractos pila, cola y lista, pero a veces estos no nos permiten reflejar la realidad y necesitamos crear tipos más complejos (como, por ejemplo, una lista ordenada). En este ejercicio definimos el TAD **tGraph** que implementa un grafo dirigido. Un grafo es un conjunto de elementos (nodos), en nuestro caso de tipo string, conectados por aristas. Dados dos nodos A y B, decimos que A esta conectado a B si existe una arista que va desde el nodo A hasta el nodo B. Un ejemplo en el cual se pueden utilizar grafos es para almacenar las distancias de las carreteras entre dos localidades. Los nodos representarían las localidades y las aristas, las distancias. Para facilitar su comprensión, os proporcionamos un ejemplo de un grafo dirigido con 5 nodos y las aristas que los conectan: La implementación de este grafo utilizando el tipo de datos tGraph quedaría como se muestra en la figura siguiente:

i) A partir de esta explicación, completa la definición del TAD **tGraph** utilizando punteros:

```
type
        tEdge = record
                city : string;
                distance: integer;
                nextEdge : pointer to tEdge;
        end record


        tNode = record
                nextNode : pointer to tNode;
                tag: string;
                firstEdge : pointer to tEdge;
```

```
        end record

        tGraph = record
                firstNode : pointer to tNode;
        end record
end type
```

ii) Este nuevo TAD ofrecerá varias operaciones, entre ellas te pedimos que implementes la operación que a partir de un node que tiene la etiqueta t, devuelva el valor distance mayor de todas las aristas que tiene encadenadas.

Ejemplo: **max_distance** (g, “C”) devuelve 5.

```
function max_distance (g: tGraph, t: string ): integer
var
        founded : boolean;
        node : pointer to tNode;
        edge : pointer to tEdge;
        distance : integer;
end var
        node := g^.firstNode;
        founded := false;
        distance := 0;

        while (node ≠ NULL) and not found do
                if node^.tag = t then
                        founded := true;
                else
                        node := node^.nextNode;
                end if
        end while
        if founded then
                edge := node^.firstEdge;
                distance := edge^.distance;

                while (edge ≠ NULL) do
```

```
                edge := edge^.nextEdge;

                if (edge^.distance > distance) then
                    distance := edge^.distance;
                end if
            end while
        end if

        return distance;
end function
```