PEC3
Prácticas de Programación
PEC3 - 20222

Fecha límite de entrega: **23 / 04 / 2023**

Formato y fecha de entrega

La PEC debe entregarse antes del día **23 de Abril de 2023 a las 23:59.**

Es necesario entregar un fichero en formato ZIP, que contenga:

- Un documento en formato **pdf** con las respuestas de los ejercicios. El documento debe indicar en su primera página el **nombre y apellidos** del estudiante que hace la entrega.

Es necesario hacer la entrega en el apartado de entregas de EC del aula de teoría antes de la fecha de entrega. **Únicamente el último envío** dentro del período establecido será evaluado.

Objetivos

- Saber interpretar y seguir el código de terceras personas.
- Adquirir los conceptos teóricos explicados sobre las técnicas de análisis de algoritmos y recursividad.
- Diseñar funciones recursivas, identificando los casos base y recursivos, siendo capaces de simular la secuencia de llamadas dada una entrada.

Criterios de corrección:

Cada ejercicio tiene asociada su puntuación sobre el total de la actividad. Se valorará tanto que las respuestas sean correctas como que también sean completas.

- No seguir el **formato de entrega**, tanto por lo que se refiere al **tipo y nombre de los ficheros** como al contenido solicitado, comportará una **penalización importante** o la cualificación con una **D de la actividad**.
- En los ejercicios en que se pide lenguaje algorítmico, se debe respetar el **formato.**
- En el caso de ejercicios en lenguaje C, estos **deben compilar para ser evaluados**. Si compilan, se valorará:
  - Que **funcionen** tal como se describe en el enunciado.
  - Que se respeten los **criterios de estilo y** que el código esté **debidamente comentado**.
  - Que las **estructuras** utilizadas sean las correctas.

Aviso

Aprovechamos para recordar que **está totalmente prohibido copiar en las PECs** de la asignatura. Se entiende que puede haber un trabajo o comunicación entre los estudiantes durante la realización de la actividad, pero la entrega de esta debe que ser individual y diferenciada del resto. Las entregas serán analizadas con **herramientas de detección de plagio.**

Así pues, las entregas que contengan alguna parte idéntica respecto a entregas de otros estudiantes serán consideradas copias y todos los implicados (sin que sea relevante el vínculo existente entre ellos) suspenderán la actividad entregada.

Guía citación:
https://biblioteca.uoc.edu/es/contenidos/Como-citar/index.html

Monográfico sobre plagio:
http://biblioteca.uoc.edu/es/biblioguias/biblioguia/Plagio-academico/

Observaciones

Esta PEC avanza en el proyecto presentado en la PEC1.
En este documento se utilizan los siguientes símbolos para hacer referencia a los bloques de diseño y programación:


Indica que el código mostrado es en **lenguaje** algorítmico


Indica que el código mostrado es en **lenguaje** C.


Muestra la ejecución de un programa en **lenguaje** C.

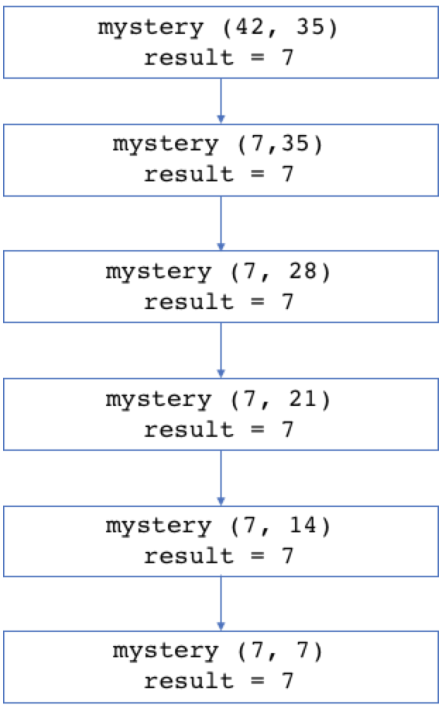
Ejercicio 1: Conceptos básicos de recursividad [40%]

Dada la función recursiva **mystery**, calcula qué valor devuelve la invocación **mystery(42,35)** y completa el **modelo de las copias** o **grafo de invocaciones** para ver cómo has llegado al resultado:

```
function mystery (a, b: integer): integer
var
  result: integer;
end var
  if a = b then
      result := a;
  else
      if a > b then
           result := mystery (a-b, b);
      else
           result := mystery (a, b-a);
      end if
  end if
  return result;
end function
```

Solución:

El resultado de mystery (42, 35) es 7, que resulta de ejecutar la secuencia de invocaciones representadas en el modelo de copias de la figura siguiente:


mystery (42, 35)
result = 7
mystery (7,35)
result = 7
mystery (7, 28)
result = 7
mystery (7, 21)
result = 7
mystery (7, 14)
result = 7
mystery (7, 7)
result = 7

Exercici 2: Diseño de algoritmos recursivos [60%]

En la PR1 hemos trabajado agrupando los proyectos en una lista y además creando una lista de ONGs. En esta PEC continuaremos trabajando con estas estructuras.

Se agregan estructuras nuevas que nos permitirán planificar las llamadas de agradecimiento a los benefactores. Existirá una cola de llamadas (**tCallQueue**) para cada ONG. A pesar de que en una aplicación real de Contact Center la gestión de las llamadas es muy compleja, aquí se muestra una versión simplificada en la que existe una única cola por ONG. Para acceder con más rapidez a la información se modifica ligeramente el tipo **tBenefactor**. En lugar de guardar el documento de la persona almacenaremos directamente el puntero a la persona; de esta forma podremos obtener sus datos fácilmente sin necesidad de buscar dentro de la lista de personas.

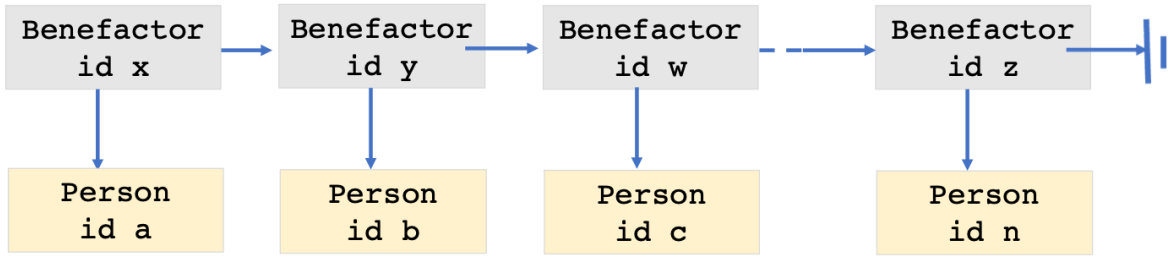
Los tipos de datos con los cuales trabajaremos se facilitan a continuación:

```
type
      tProject = record
            code: string;
            ngoCode : string;
            budget: real;
      end record

      tProjectNode = record
            project: pointer to tProject;
            next: pointer to tProjectNode;
      end record

      tProjectList = record
            first: pointer to tProjectNode;
            count: integer;
      end record

      tBenefactor = record
            person: pointer to tPerson;
            totatAmount: real;
      end record

      tBenefactorData = record
            elems: pointer to tBenefactor;
            count: integer;
      end record

      tCallQueue = Queue (tBenefactor);

end type
```

Entrando más en detalle, las llamadas de agradecimiento a los benefactores se guardan en una cola de llamadas **tCallQueue** (de la cual desconocemos su implementación). Esta estructura tipo FIFO nos permite realizar llamadas ordenadas por el criterio que consideremos más oportuno. De manera que al planificar una llamada nueva la añadimos a la cola y esta nueva llamada se realizará después de haber llamado a todas las personas planificadas con anterioridad. De igual manera, una vez realizada la llamada, esta sale de la cola y da paso a la siguiente llamada.

Para crear la cola o para añadir nuevos registros simplemente se debe decidir qué criterio seguir y en base a éste ir llenando la cola según la prioridad preestablecida. Además, como en la estructura tenemos el apuntador a la persona podremos acceder fácilmente a todos sus datos. Con ello, además de tener acceso al teléfono tenemos acceso a su correo electrónico, que puede ser una vía de contacto alternativa si no conseguimos contactar telefónicamente.

A continuación se puede ver una representación gráfica simplificada de la cola de llamadas de una ONG y su relación con los benefactores. También se representa la relación benefactor/persona, ya que el primero no deja de ser una extensión/ampliación de la información asociada a una persona:



Donde los nodos grises representan la cola de llamadas de agradecimiento a cada benefactor (**tBenefactor**) y los nodos amarillos los datos de las personas (**tPerson**).

Se pide definir en lenguaje algorítmico los siguientes métodos recursivos:

1. La función **fillCallQueue** que dada una cola de llamadas (**tCallQueue**) y el puntero a una tabla de benefactores (**tBenefactorData**) de una ONG, rellena la cola de llamadas con el mismo orden que en la tabla de benefactores. Únicamente deseamos llamar a los benefactores más generosos para darle las gracias así que solo se incluirán en la cola de llamadas aquellos que hayan aportado un mínimo de dinero que también recibimos como parámetro (threshold). La cabecera de la función es la siguiente:

```
action fillCallQueue (out calls: tCallQueue, in benefactors: tBenefactorData, in threshold: real)
```

2. La acción **greatBenefactor** que dada una cola de llamadas **tCallQueue** de una ONG, nos devuelve el número de benefactores que han aportado la cantidad de dinero mayor en la ONG (valor que también se retorna). La cabecera de la función es la siguiente:

```
action greatBenefactor (in calls: tCallQueue, out bestAmount: real,
out num: integer)
```

Nota: tenéis que implementar todas las funciones auxiliares necesarias para resolver el ejercicio excepto las funciones del TAD cola. Es decir, podéis usar las funciones **createQueue**, **enqueue**, **dequeue**, **head** y **emptyQueue** pero no será necesario implementarlas.

Solución

```
action fillCallQueue (out calls: tCallQueue,
                      in benefactors: tBenefactorData,
                      in threshold: real)

     fillCallQueueRec (calls, benefactors.elems,
                       benefactors.count, threshold);

end action
```

```
action fillCallQueueRec(out calls: tCallQueue,
                        in elems: pointer to tBenefactor,
                        in count: integer, in threshold: real)

     if count = 0 then
        calls := createQueue (tCallQueue);
     else
        fillCallQueueRec (calls, elems, count-1, threshold);

        if elems[count].totalAmount ≥ threshold then
           enqueue (calls, elems[count]);
        end if
     end if

end action
```

```
action greatBenefactors (in calls: tCallQueue,
                         out bestAmount: real,
                         out num: integer)

  var
     calls_copy : tCallQueue;
  end var

  duplicate (calls_copy, calls);

  greatBenefactorsRec (calls_copy, bestAmount, num);

  destroy (calls_copy);

end action

action greatBenefactorsRec (inout calls: tCallQueue,
                            out bestAmount: real,
                            out num: integer)
var
  benefactor : tBenefactor;
end var

  if emptyQueue (calls) then
     bestAmount := 0.0;
     num := 0;
  else
     benefactor := head (calls);
     dequeue (calls);
     greatBenefactorsRec (calls, bestAmount, num);

     if benefactor.totalAmount > bestAmount then
        bestAmount := benefactor.totalAmount;
        num := 1;
     else if benefactor.totalAmount = bestAmount then
            num := num + 1;
     end if

  end if

end action
```