PROBLEMA 1

Sea ***P(x)*** un polinomio definido para todo el dominio de los reales. Utilice la técnica de diseño divide y vencerás para implementar un algoritmo que devuelva una lista con todas las raíces de dicho polinomio. *Las raíces se definen en forma de intervalos* y deberán ser calculadas con una precisión menor que ***p*** (para el intervalo ***[a, b]*** la precisión será |***a - b***|)**.** Se dispone de la función **existenRaices(a, b)** que devuelve ***verdadero*** si existe al menos una raíz en el intervalo ***[a, b]*** y ***falso*** en caso contrario. *El algoritmo implementado deberá buscar en el menor número de intervalos posibles.*

Nota: Utilice las siguientes operaciones del tipo Lista para resolver el problema.

```
c: Lista                  // Define una variable de tipo lista
c := <crea Lista>         // Crea una lista vacía
c := <concatena a y b>    // Concatena las listas a y b.
c := <añade e en c>       // Añade un el elemento e a la lista c
```

Ejemplo: Para el polinomio $P(x) = x^2 – 6.2*x + 9.25$ y con precisión 0.001 se obtendría las siguientes raíces:

Raíz 1 -> [2.49951171875, 2.500244140625]
Raíz 2 -> [3.699951171875, 3.70068359375]

Tiempo estimado: 45 minutos. Puntuación: 5 puntos

SOLUCION

Clase CalculaRaices hereda EsquemaDyV

precisión: Real

```
func P (x:Real) dev (y: Real)
  y := x * x –  6.2 * x + 9.25
fin

Clase Intervalo
  a: Real
  b: Real
  proc Intervalo(pa, pb: Entero)
     a := pa
     b := pb
  fin
fClase

Clase Raiz hereda Intervalo
Clase Problema hereda Intervalo

Clase Solucion
   L: Lista
fClase

func dyV (x: Problema) dev (s: Solucion)
var
  subproblemas: Array [] de Problemas
  subsoluciones: Array [] de Soluciones
alg
    si esCasoBase(x):
        s := resuelveCasoBase(x)
   otros |
       subproblemas := divide(x)
       desde i := 1 hasta tamaño(subproblemas) hacer
          subsoluciones[i] := dyV(subproblema)
       fdesde
       s = combina(subsoluciones)
   fsi
fin

func esCasoBase (x: Problema) deve (b: Lógico)
alg
   b : = ((|x.a – x.b| < precisión) O No existenRaices(x.a, x.b))
fin

func resuelveCasoBase (p: Problema) dev (s: Solución)
    raiz: Raiz
alg
    s := <nueva Solución>
    si existenRaices(x.a, x.b) :
      raiz := <nueva Raiz(p.a, p.b)>
      s.l  := <añade raiz en s.l>
   fsi
fin
```

```
func divide (p: Problema) dev (ss: Array [] de Soluciones)
var
   med : Real
   numss :Entero
alg
    ss := <Array [1 .. 2] de Soluciones>
    med = (p.a + p.b) / 2
    ss[1] := <nuevo Problema(p.a, med)>
    ss[2] := <nuevo Problema(med, p.b)>
fin

func  combina (ss: Array [] de Solcuciones) dev (s: Solucion)
    s := <nueva Solución>
    s.l := <concatena ss[1].l y ss[2].l>
fin

func resuelve(a, b: Real) dev (raices: Lista)
var
   p: Problema
   s: Solución
alg
    s := <nueva Solucion()>
    p := <nuevo Problema(a, b)>
    s := dyv(p)
    raices := s.l
fin

fClase
```

PROBLEMA 2

El alcalde de un determinado pueblo dispone de un presupuesto para realizar las diversas actividades que estaban contempladas en su programa electoral. Con el fin de maximizar el número de habitantes que se vean beneficiados por esas medidas, los asesores pretenden diseñar un algoritmo de programación dinámica para escoger aquellas actividades que hagan máximo el número de habitantes que se verían afectados por dicha actividad.

Ejemplo: La siguiente tabla muestra un ejemplo de datos de entrada. Para un presupuesto de 2500 unidades la mejor opción es realizar las actividades 2 y 3, que agruparían 50 habitantes y tendría un coste de 2500 unidades.

Actividad	Habitantes afectados	Coste de la actividad
1	35	1400
2	25	1300
3	25	1200
4	45	1400

Presupuesto: 2500

Teniendo en cuenta que:

- `habit[i]` representa los habitantes afectados para la actividad `i`.
- `coste[i]` representa el coste de realizar la actividad `i`.
- `act` representa el número total de actividades que se pueden realizar
- `pres` representa el presupuesto (en unidades) que dispone el alcalde.
- Para calcular los habitantes afectados por las actividades simplemente se deben sumar los habitantes afectados por cada actividad, es decir, no hace falta contemplar si varias actividades afectan al mismo habitante.
- Las actividades no se pueden fraccionar.

Se pide:

a) Escriba la ecuación en recurrencia que resuelva este problema.

b) Realizar un algoritmo de programación dinámica que permita calcular cual es el número máximo de habitantes que pueden beneficiarse para un presupuesto concreto.

c) Realizar las modificaciones, y añada el código necesario para mostrar por la pantalla la relación de actividades que deben realizarse para alcanzar ese máximo número de habitantes.

a) Suponemos que al menos existe una actividad a realizar.

$$PD(i,j) = \begin{cases} 0 & \text{si } j=0 \\ 0 & \text{si } i=1 \text{ y } j < coste[i] \\ habit[i] & \text{si } i=1 \text{ y } j >= coste[i] \\ PD(i-1, j) & \text{si } i>1 \text{ y } j < coste[i] \\ maximo(PD(i-1, j), habit[i] + PD(i-1, j-coste[i])) & \text{si } i>1 \text{ y } j >= coste[i] \end{cases}$$

Donde "i" representa el número de actividades disponibles y "j" representa la cantidad maxíma que se puede invertir en las actividades disponibles.

b)

```
clase PD_Feb0708
      ...

      funcion PD_Feb0708() dev (max: Entero)
      var
            i,j : Entero
            PD: Array [1..act][0..pres]
      alg
            desde i = 1 hasta act
                  desde j = 0 hasta pres
                        si j = 0 :
                              PD[i][j] := 0
                        | i = 1 y j < coste[i] :
                              PD[i][j] := 0
                        | i = 1 y j >= coste[i] :
                              PD[i][j] := habit[i]
                        | i > 1 y j < coste[i] :
                              PD[i][j] := PD[i-1][j]
                        | otras :
                              PD[i][j] := maximo(PD[i-1, j],
                                          habit[i] + PD[i-1, j-coste[i]])
                        fsi
                  fdesde
            fdesde
c)          // Comienzo apartado c
            i := act
            j := pres
            mientras i >= 1 y j > 0
                  si i = 1 :
                        si PD[i][j] > 0 :
                              escribe "Actividad: " + i
                        fsi
                  | otras :
                        si PD[i][j] != PD[i - 1][j] :
                              j := j - coste[i]
                              escribe "Actividad: " + i
                        fsi
                  fsi
                  i := i - 1
            fmientras
            // fin apartado c
            max := PD[act, pres]
      fin
```

Asignación de registros en un compilador

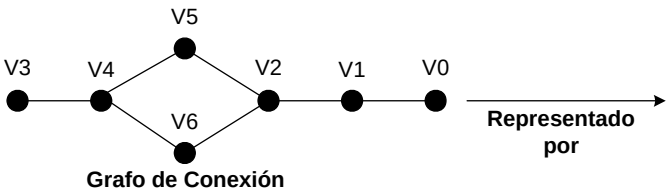
Cuando se desea implementar un programa en alto nivel hace falta un compilador que realice la traducción a código máquina. En la mayoría de los casos, en primer lugar se realiza la traducción a una representación intermedia. Una de las cosas fundamentales para llevar a cabo esta traducción de forma óptima es utilizar el menor número de registros posibles en función de las variables que aparezcan en el código y las relaciones existentes entre las mismas. En cada momento cada una de las variables "vivas" debe estar asignada a un registro determinado. El ciclo de vida de una variable comienza cuando es definida y finaliza cuando es referenciada por última vez. Dos o más variables diferentes podrán tener asignado el mismo registro si no coinciden en ningún momento en su ciclo de vida.

Este problema se puede representar mediante un grafo de forma que cada vértice representa una variable. Dos vértices están conectados si los ciclos de vida de las variables que representan se solapan, es decir, si no pueden compartir el mismo registro.

Se cuenta con *n* variables, cada una de ellas representada por un número entero comprendido entre 0 y *n*-1. Para ubicar *n* variables, en el caso peor se necesitan *n* registros, representados por un entero comprendido entre 0 y *n*-1.

Se desea resolver el problema de asignación de registros a variables de forma que se utilice el menor número de registros posibles, teniendo en cuenta que dos variables que se encuentren conectadas en el grafo no pueden tener asignado el mismo registro.

Se cuenta con la clase *ProblemaRegistros* que contiene el grafo con la información acerca de las conexiones entre las variables, de forma que *grafo[i][j] = true* indica que las variables *i* y *j* están conectadas, es decir, no pueden estar ubicadas en el mismo registro, y *false* en caso contrario. Un ejemplo de la equivalencia entre el grafo y la mostrada en la siguiente figura.



	0	1	2	3	4	5	6
0	false	true	false	false	false	false	false
1	true	false	true	false	false	false	false
2	false	true	false	false	false	true	true
3	false	false	false	false	true	false	false
4	false	false	false	true	false	true	true
5	false	false	true	false	true	false	false
6	false	false	true	false	true	false	false

Matriz del grafo

Problema 1: Asignación de registros en un compilador mediante voraz

Para resolver este problema mediante **voraz** se va a implementar el siguiente algoritmo**: a cada variable (comenzando por la variable representada por 0) se le asigna un registro válido representado por el menor número posible (se comienza desde el registro 0). Un registro válido para una variable *v* es aquél que no esté ya asignado a ninguna de las variables conectadas a *v*.**

Se va a diseñar la clase *RegistrosVoraz*, la cual posee el siguiente conjunto de atributos:

- **`boolean`** `[][] grafo:` grafo que representa el problema y proporciona información acerca de las conexiones entre las variables.
- **`int`** `numVariables:` número de variables del problema.
- **`int`** `variableActual:` variable que se estudia en cada paso del algoritmo voraz para proceder a su ubicación en uno de los registros.
- **`int`** `registroSeleccionado:` registro seleccionado para ubicar la variable actual en cada paso del algoritmo voraz.
- **`int`**`[] asignacionRegistros:` array donde se almacena la solución, es decir, la ubicación de cada variable en un registro, de forma que *asignacionRegistros[var] = reg* indica que la variable *var* se ubicará en el registro *reg*. Se va completando en cada paso del algoritmo voraz, de forma que al final de la ejecución tendrá la solución completa.

A continuación se muestra la solución al problema anterior utilizando la técnica voraz:

`asignacionRegistros`
con la técnica Voraz

0	1	0	0	1	2	2
0	1	2	3	4	5	6

Se pide completar la clase *RegistrosVoraz* teniendo en cuenta las indicaciones proporcionadas en el enunciado y en la plantilla.

Problema 2: Asignación de registros en un compilador mediante backtracking

Para obtener una solución donde el número de registro sea mínimo, se propone utiliza el esquema de BtOptimo, donde en cada etapa se asignará un registro a cada una de las variables. Se pide completar los cuadros para obtener una solución que haga uso del mínimo número de registros posibles para el tratamiento de las variables. Utilizando dicha técnica, se obtendrá la asignación mostrada en la siguiente figura:

`asignacionRegistros`
con la técnica de BT

0	1	0	1	0	1	1
0	1	2	3	4	5	6

Consideraciones Generales:

- Las soluciones a cada uno de los métodos se escribirán obligatoriamente en los espacios de la plantilla proporcionados para tal fin, y por ello los alumnos no entregarán ningún folio adicional.
- El examen debe escribirse preferiblemente a bolígrafo, pero se aceptarán exámenes escritos a lápiz dejando bajo la responsabilidad del alumno cualquier deterioro que pueda sufrir el texto escrito de esta forma.

APELLIDOS:
NOMBRE:
TITULACIÓN: ***DNI:***

```
public class RegistrosVoraz extends EsquemaVZ implements EstrategiaSolucion {
    // grafo[i][j] = true --> variables i y j están conectadas (no se
    // pueden ubicar en el mismo registro)
    boolean [][] grafo;
    // número de variables
    int numVariables;

    // Variable que se estudia en cada paso para proceder a su ubicación
    // en un registro
    int variableActual;

    // Registro seleccionado para ubicar la variable actual
    int registroSeleccionado;

    // Array donde se almacena la solución, es decir, la ubicación de
    // cada variable, de forma que "asignacionRegistros[var] = reg"
    // indica que la variable "var" se ubicará en el registro "reg".
    public int[] asignacionRegistros;

    // Se toman los datos del problema
    public RegistrosVoraz(ProblemaRegistros p) {
        this.grafo = p.getGrafo();
        this.numVariables = this.grafo.length;
    }
    // Se inicializa variableActual y se crea asignacionRegistros con el
    // tamaño adecuado
    public void inicializa(){
        variableActual = 0;
        asignacionRegistros = new int[numVariables];
    }
    // Devuelve cierto en caso de que todas las variables hayan sido
    // ubicadas en un registro
    public boolean fin(){
        return variableActual == numVariables;
    }
    public boolean prometedor(){
        return true;
    }
    public void procesamientoInicial(){
    }
    // Lanza la ejecución del algoritmo voraz
    public void solucion(){
        voraz();
    }
    public void procesamientoFinal(){
    }
```

```
// Se selecciona un registro para la variableActual teniendo en
// cuenta lo indicado en el enunciado. Dicho registro se almacena
// en el atributo registroSeleccionado para poder acceder a él en
// el método anotaEnSolucion.
public void seleccionaYElimina(){
    boolean enc = false;
    int registro = 0;
    while(!enc){
        boolean esPosible = true;
        int j = 0;
        while(esPosible && j < variableActual){
            // Si están conectadas
            if(grafo[variableActual][j]){
            // Si alguna variable vecina (j) está ubicada en el
            // registro analizado, ya no es posible
                if(registro == asignacionRegistros[j])
                    esPosible = false;
            }
            j++;
        }
        if(!esPosible)
            registro++;
        else
            enc = true;
    }
    registroSeleccionado = registro;
}
// Se actualiza asignacionRegistros y variableActual
public void anotaEnSolucion(){
    asignacionRegistros[variableActual] = registroSeleccionado;
    variableActual++;
}}
```

APELLIDOS:
NOMBRE:
TITULACIÓN: ***DNI:***

```
public class RegistroBtOptimo extends EsquemaBtOptimo implements EstrategiaSolucion
{
	Solucion sol;
	Solucion solucionOptima;
	boolean [][] grafo;
	int numVariables;


	public RegistroBtOptimo(ProblemaRegistros p){
		grafo = p.getGrafo();
		numVariables = grafo.length;
	}
	public void procesamientoInicial() { }

	public void solucion() {
	 solucionOptima = new SolucionRegistros(numVariables);
	            ((SolucionRegistros)solucionOptima).registrosUtilizados =
	 numVariables;
	 //creamos la solución normal
	 sol = new SolucionRegistros(numVariables);
	 ((SolucionRegistros)sol).registrosUtilizados = 0;
	 //creamos la primera etapa
	 EtapaRegistros etapaReg = new EtapaRegistros();
	 etapaReg.k = -1;
	 btOptimo(etapaReg);
	}

	public void procesamientoFinal() { }
	protected boolean esSolucion(Etapa x) {

	 boolean es = ((EtapaRegistros)x).k==numVariables-1;
	 return es;

	}


	protected boolean esMejor() {


	 return (((SolucionRegistros)sol).registrosUtilizados<
	 ((SolucionRegistros)solucionOptima).registrosUtilizados);


	}

	protected void actualizaSolucion() {
```

```
SolucionRegistros solReg = (SolucionRegistros)sol;
SolucionRegistros solOptReg = (SolucionRegistros)solucionOptima;
for(int i=0;i<solOptReg. asignacionRegistros.length;i++){
      solOptReg.asignacionRegistros[i]=solReg.asignacionRegistros[i];
}
solOptReg.registrosUtilizados = solReg.registrosUtilizados;
```

```
}

protected Candidatos calculaCandidatos(Etapa x) {
EtapaRegistros etapaReg = (EtapaRegistros)x;
CandidatosRegistros cand = new CandidatosRegistros();
if(etapaReg.k<numVariables-1){
      cand.indice = -1;
}
else{
      cand.indice = numVariables;
}

return cand;
```

```
}
protected boolean quedanCandidatos(Candidatos cand) {
CandidatosRegistros candReg = (CandidatosRegistros)cand;
return (candReg.indice<numVariables-1);
```

```
}
protected Etapa seleccionaCandidato(Candidatos cand, Etapa x) {
CandidatosRegistros candReg = (CandidatosRegistros)candEtapaRegistros
etapaReg = (EtapaRegistros)x;
candReg.indice++;
EtapaRegistros etapaSig = new EtapaRegistros();
etapaSig.k = etapaReg.k+1;
etapaSig.recurso = candReg.indice;
return etapaSig;
```

```
}
protected boolean esPrometedor(Candidatos cand, Etapa x, Etapa xsig) {
```

```
            CandidatosRegistros candReg = (CandidatosRegistros)cand;
            EtapaRegistros etapaReg = (EtapaRegistros)xsig;
            SolucionRegistros solReg = (SolucionRegistros)sol;
            boolean comp = true;
            for(int i=0;i<etapaReg.k&&comp;i++){
                  if(grafo[etapaReg.k][i]){
                        //no pueden tener el mismo recurso
                  comp = !(solReg. asignacionRegistros[i]==etapaReg.recurso);
                  }
            }

            return comp;
      }
      protected void anotaSolucion(Candidatos cand, Etapa x, Etapa xsig) {
            SolucionRegistros solReg = (SolucionRegistros)sol;
            EtapaRegistros etapaReg = (EtapaRegistros)xsig;
            CandidatosRegistros candReg = (CandidatosRegistros)cand;
            solReg.asignacionRegistros[etapaReg.k]=candReg.indice;
            boolean esta = false;
            for(int i=0;i<etapaReg.k && !esta;i++){
                  esta = (solReg.asignacionRegistros[i]==candReg.indice);
            }
            if(!esta){
                  solReg.registrosUtilizados++;
            }
      }

      protected void cancelaAnotacion(Candidatos cand, Etapa x, Etapa xsig) {
            SolucionRegistros solReg = (SolucionRegistros)sol;
            EtapaRegistros etapaReg = (EtapaRegistros)xsig;
            CandidatosRegistros candReg = (CandidatosRegistros)cand;

            boolean esta = false;
            for(int i=0;i<etapaReg.k && !esta;i++){
                  esta = (solReg.asignacionRegistros[i]==candReg.indice);
            }
            if(!esta){
                  solReg.registrosUtilizados--;
            }
      }


      class EtapaRegistros extends Etapa{
            int k;
            int recurso;
      }
      class CandidatosRegistros extends Candidatos{
            int indice;
      }
      class SolucionRegistros extends Solucion{
            int[]asignacionRegistros;
            int registrosUtilizados;
            public SolucionRegistros(int tam){
                  asignacionRegistros= new int[tam];
                  registrosUtilizados = 0;
            }
      }


}
```