Ejercicios de Programación (parte 2)

Profesor: Mariano Fernández López
26 de enero de 2023

Índice

1. Métodos recursivos **3**

2. Elevación de excepciones **4**

3. *ArrayList* y *List* **5**

4. Métodos iterativos **5**

5. Expresiones lambda en Java **5**

6. Maven **7**
6.1. Primer proyecto en Maven 7
6.2. La libreta de contactos 8
6.3. La libreta de contactos ejecutable con *java -jar* 12
6.4. La libreta de contactos con uso de una biblioteca *jar* 12
6.5. Arquetipos 16

7. Prueba de programas **19**

8. Desarrollo dirigido por pruebas **20**

9. Complejidad de algoritmos **21**

10. Recursividad, desarrollo dirigido por pruebas y complejidad **26**

11. Expresiones lambda, desarrollo dirigido por pruebas y complejidad **27**

12. Tipos abstractos de datos y estructuras de datos **28**
12.1. Listas 28
12.2. Pilas 28
12.3. Colas 29
12.4. Ordenación de estructuras lineales 29
12.4.1. Inserción directa 29
12.4.2. Mergesort 30
12.4.3. Timsort 30
12.4.4. Quicksort 30
12.4.5. Ejemplos de ejecución de algoritmos de ordenación . . 31
12.4.6. Límite inferior en la complejidad de la ordenación . . . 31
12.4.7. Los métodos *sort* en Java 31
12.5. Árboles 33
12.6. Hashing 40
12.7. Colas con prioridad 40
12.8. Grafos 41

12.9. Iteradores 43

13.Expresiones lambda y estructuras de datos **43**

14.Técnicas algorítmicas **44**

15.Ejercicios extraídos de pruebas objetivas **44**

1. Métodos recursivos

Ejercicio 1. Explique qué es un método recursivo.

Ejercicio 2. Programe las siguientes funciones mediante métodos recursivos:

1. La suma $0 + 1 + 2 + ... + n$.
2. El factorial de un número.
3. La potencia n-ésima de un número natural.
4. La suma de los elementos de una lista de números.
5. La media aritmética de una lista de números.
6. La desviación típica de una lista de números.
7. La suma de los primeros números pares hasta n asumiendo $n \geq 2$. Por ejemplo, $sumaPares(9) = 8 + 6 + 4 + 2 = 20$.
8. La suma de los elementos pares de una lista de enteros. Por ejemplo, $sumaPares([1, 2, 3, 4]) = 6$.
9. Dada una lista de números naturales mayores o iguales que 2, obtiene otra lista con los números pares de la lista inicial, en el mismo orden y respetando los números repetidos. Por ejemplo,

$$obtenerListaPar([1, 2, 6, 11]) = [2, 6].$$

10. La lista de los primeros números pares hasta n asumiendo $n \geq 2$. Por ejemplo, $listaPar(9) = [8, 6, 4, 2]$.
11. Producto escalar de dos listas de números no vacías y del mismo tamaño. Por ejemplo, $calcularProductoEscalar([1, 2, 3], [2, 4, 6]) = 1 \cdot 2 + 2 \cdot 4 + 3 \cdot 6 = 2 + 8 + 18 = 28$
12. El elemento n-ésimo de la sucesión de Fibonacci.
13. Calcule el cociente entre el decimo tercer y el decimo segundo elemento de la sucesión de Fibonacci, y compare el resultado con $\frac{1+\sqrt{5}}{2}$.
14. Averigüe la relación entre la sucesión de Fibonacci y la razón áurea.

El cociente entre el elemento (n) y (n-1) de la serie de fibonacci es igual al número aureo.

Ejercicio 3. ¿En qué consiste el problema de las Torres de Hanoi? ¿Cómo puede resolverse?

Ejercicio 4. Escriba un método recursivo con la siguiente cabecera:

```
double integralEXCuadrado(double limInf, double limSup, double h)
```

para que calcule la integral de e^{x^2} mediante el método de los rectángulos. Es decir,

$$suma = \sum_{x=inf}^{sup} e^{x^2} h$$

donde inf, el límite inferior, se corresponde con *limInf* en la cabecera del predicado; *sup*, el límite superior, se corresponde con *limSup*; h es el paso; y *suma*, el resultado de la integral definida, se corresponde con valor devuelto por el método.

Por ejemplo, si se desea calcular $\int_0^1 e^{x^2}\, dx$ con $h = 0{,}1$, se realizará la siguiente invocación:

```
Matematicas.integralEXCuadrado(0, 1, 0.1);
```

donde 0 es el límite inferior; 1, el límite superior; y 0,1 el paso, por tanto,

$$suma = \sum_{x=0}^{1} e^{x^2} 0{,}1$$

Ejercicio 5. ¿Por qué la ejecución de un método recursivo puede levantar una excepción de desbordamiento de pila con la misma entrada que la versión no recursiva no provoca tal excepción?

Ejercicio 6. Escriba un método recursivo que reciba una cadena de caracteres y devuelva la suma de sus códigos. La cabecerá será la siguiente:

```
public static int sumaCodigos(String str)
```

Ejercicio 7. Escriba un método recursivo que reciba una cadena de caracteres y la transforme según el cifrado de César. La cabecerá será la que se muestra a continuación:

```
public static String cifrarCadena(String str, int desplazamiento)
```

Ejercicio 8. Escriba un método recursivo que transforme una cadena de caracteres en otra en la que sólo estén presentes los caracteres con código par, y lo estén en el mismo orden en que aparecen en la cadena original. Por ejemplo, si la cadena de entrada es "Llego mañana", la de salida será "Ll n", pues el código de 'L' es 76, el de 'l' es 108, el del espacio, 32, etc. Sin embargo, la letra 'a', por ejemplo, no aparece porque su código es 97. La cabecera del método será la que se muestra a continuación:

```
public static String conCodigoPar(String str)
```

2. Elevación de excepciones

Ejercicio 9. Modifique el código del ejercicio 2 para que se eleve una excepción cuando haya una entrada no válida (por ejemplo, un número menor que 0 en la suma de los números naturales hasta n).

3. *ArrayList* y *List*

Ejercicio 10. Explique qué es *List* en Java y qué relación guarda con *ArrayList*.

4. Métodos iterativos

Ejercicio 11. Resuleva los apartados del 1 al 11 del ejercicio 2, ambos inclusive, mediante bucles.

5. Expresiones lambda en Java

Ejercicio 12. Explique qué es una expresión lambda.

Ejercicio 13. El propósito del presente ejercicio es mostrar un uso básico de funciones lambda con listas. Para ello se pide resolver el ejercicio 4 de tal forma que sea aplicable a diferentes funciones reales de variable real. La cabecera el método ha de ser la que se muestra a continuación:

```
public static double integral(double a, double b, double h,
    Function<Double, Double> funcion)
```

Ejercicio 14. El propósito de este ejercicio es introducir el uso de listas utilizando expresiones lambda. Se pide crear los métodos que se describen a continuación:

1. Dada una lista de números, obtiene otra lista con sus cuadrados.
2. Dada una lista de números, obtiene su suma.
3. Dada una lista de enteros, obtine la suma de los números pares.
4. Dada una lista de enteros, obtine otra lista con los números pares de la lista inicial.
5. La media aritmética de los elementos de una lista dada.
6. La desviación típica de los elementos de una lista dada.
7. El menor de una lista de números.
8. A partir de una lista de *String*, obtiene otra lista con sus elementos transformados a minúscula. La cabecera será la siguiente:

   ```
   public static List<String> aMinusculas(List<String> lista)
   ```

9. Devuelve una cadena que sea la concatenación de todos los elementos de una lista de *String*. La cabecera será la que se muestra a continuación:

```
public static String concatenarCadenas(List<String> al)
```

10. Devuelve una lista con todos los elementos de otra lista de *String* transformados a minúsculas. La cabecera será la que se muestra a continuación:

```
public static List<String> transformarAMinusculas(List<String
    > al)
```

11. Devuelve una lista con todos los elementos de otra lista de *String* que no empiezen por la letra 'A'. La cabecera será la siguiente:

```
public static List<String> noEmpiezaPorA(List<String> al)
```

12. Devuelve una lista con todos los números mayores que seis resultantes de multiplicar por 2 todos los elementos de una lista de entrada. La cabecera será la que se muestra a continuación:

```
public static List<Integer> por2MayorQue6(List<Integer> lista
    )
```

Ejercicio 15. Realice los siguientes apartados:

1. Cree la clase *Persona* con los atributos *nombre* de tipo *String* y *fechaDeNacimiento* de tipo *LocalDate*, y con un método que calcule la edad de la persona en años con la siguiente cabecera:

```
public long calcularEdad()
```

2. Cree la clase *Personas* con un método que permita añadir personas con la cabecera que se muestra a continuación:

```
public void annadirPersona(Persona persona)
```

3. Utilizando expresiones lambda, implemente los siguientes métodos dentro de la clase *Personas*.

   a) Obtención de la persona más joven, de acuerdo con la cabecera que se muestra a continuación:

```
public Persona elMasJoven()
```

   b) Cálculo de la suma de las edades, con la siguiente cabecera:

```
public long calcularSumaEdades()
```

   c) Otención de la edad mínima, con la cabecera que se muestra a continuación:

```
public long calcularEdadMinima()
```

   d) Cálculo de la media de edad, con la siguiente cabecera:

```
public double calcularMediaDeEdad()
```

6. Maven

El propósito de la presente sección es introducir al estudiante de la herramienta de desarrollo en Java Maven.

6.1. Primer proyecto en Maven

Ejercicio 16. Responda a las siguientes preguntas:

1. ¿Qué es Maven?
2. ¿Qué es el repositorio central de Maven?, ¿hasta qué punto son fiables las bibliotecas que hay en él?
3. ¿Qué es el repositorio local?

Ejercicio 17. Instale Maven. Por ejemplo, en OpenSuse, se ejecuta:

```
sudo zypper install maven
```

Para comprobar que se ha instalado correctamente:

```
mvn --version
```

Para otros sistemas operativos, seleccione la opción *download* de la página de Maven:

```
https://maven.apache.org/
```

Ejercicio 18. Realice los siguientes apartados:

1. Cree un proyecto en Maven ejecutando la siguiente instrucción[1]:

   ```
   mvn archetype:generate -DgroupId=pr2.org -DartifactId=my-app
       -DarchetypeArtifactId=maven-archetype-quickstart -
       DarchetypeVersion=1.4 -DinteractiveMode=false
   ```

   y explore el árbol de directorios generado.
2. ¿Qué es un arquetipo en Maven?
3. Explique el fichero *pom.xml* que hay en directorio *my-app* que se ha generado.
4. Explore el árbol de directorios.
5. Compile el programa:

   ```
   mvn compile
   ```

[1] `https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html`

6. Ejecute el programa:

```
mvn exec:java -D exec.mainClass=pr2.org.App
```

7. Elimine los artefactos generados anteriormente, vuelva compilar y ejecute de nuevo:

```
mvn clean compile
mvn exec:java -D exec.mainClass=pr2.org.App
```

8. Genere la documentación del proyecto, y explórela:

```
mvn site
cd target/site
firefox index.html &
```

9. Explique en qué consisten las siguientes fases:

   - validate.
   - compile.
   - test.
   - package.
   - install.
   - deploy.
   - clean.
   - site.

6.2. La libreta de contactos

Ejercicio 19. El propósito del presente ejercicio es la realización de una aplicación sencilla (una libreta de contactos) utilizando Maven. Se pide realizar los apartados que se muestran a continuación:

1. Genere el proyecto:

```
mvn archetype:generate -DgroupId=org.pr2 -DartifactId=libreta
    -contactos -DarchetypeArtifactId=maven-archetype-
    quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false
```

2. Explore el árbol de directorios de *src*:

```
cd libreta-contactos
tree
```

3. Explore el código de la clase principal:

```
vi src/main/java/org/pr2/App.java
```

4. Compile:

```
mvn compile
```

5. Compruebe que la apliación ejecuta:

```
mvn exec:java -D exec.mainClass=org.pr2.App
```

6. Cree un directorio para el dominio de la aplicación y otro para la interfaz de usuario:

```
md src/main/java/org/pr2/dominio
md src/main/java/org/pr2/interfaces
```

7. Escriba las clases *Contacto.java* en el directorio *src/main/java/org/pr2/dominio*

```
package org.pr2.dominio;

public class Contacto
{
    private String nombre;
    private String telefono;

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }

    public void setTelefono(String telefono) {
        this.telefono = telefono;
    }

    public String getTelefono() {
        return telefono;
    }

    @Override
    public String toString() {
        return "nombre: " + getNombre() + " " +
               "telefono: " + getTelefono();
    }

    public Contacto(String nombre, String telefono) {
        this.nombre = nombre;
        this.telefono = telefono;
    }

    public Contacto() {
    }

}
```

8. En el mismo directorio, escriba la clase *Libreta.java*

```
package org.pr2.dominio;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

public class Libreta
{
    private String nombreFichero = "contactos.txt";
    private ArrayList<Contacto> contactos = new ArrayList<>()
        ;

    public void addContacto(Contacto contacto)
    {
        contactos.add(contacto);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for(Contacto contacto : contactos)
        {
            sb.append(contacto + "\n");
        }
        return sb.toString();
    }

    public Libreta() {
        try
        {
            File fichero = new File(nombreFichero);
            fichero.createNewFile();
            Scanner sc = new Scanner(fichero);
            while(sc.hasNext())
            {
                Contacto contacto = new Contacto();
                contacto.setNombre(sc.nextLine());
                contacto.setTelefono(sc.nextLine());
                contactos.add(contacto);
            }
        }catch(IOException ex)
        {
            System.err.println(ex);
        }
    }

    private void volcarContactos()
    {
        System.out.println(contactos);
        try
```

```
        {
            FileWriter fw = new FileWriter(nombreFichero);
            for(Contacto contacto : contactos)
            {
                fw.write(contacto.getNombre()+"\n");
                fw.write(contacto.getTelefono()+"\n");
            }
            fw.close();
        }catch(IOException ex)
        {
            System.err.println(ex);
        }
    }

    public void annadirContacto(Contacto contacto)
    {
        contactos.add(contacto);
        this.volcarContactos();
    }

}
```

9. Escriba la clase *Interfaz.java* en el directorio *src/main/java/org/pr2/interfaces*

```
package org.pr2.interfaces;

import org.pr2.dominio.*;

public class Interfaz
{
    public static void iniciar(String args[])
    {
        Libreta libreta = new Libreta();
        if (args[0].equals("add"))
        {
            Contacto contacto = new Contacto(args[1], args
                [2]);
            libreta.annadirContacto(contacto);
        }
        else if (args[0].equals("show")) System.out.println(
            libreta);
        else System.out.println("Opción incorrecta");
    }
}
```

10. Modifique la clase *App.java*

```
package org.pr2;

import org.pr2.interfaces.*;

public class App
{
    public static void main( String[] args )
```

```
    {
        Interfaz.iniciar(args);
    }
}
```

11. compile:

```
mvn compile
```

12. Ejecute:

```
mvn exec:java -D exec.mainClass=org.pr2.App -D exec.args='add
     "Juan García Pérez" 65432145'
mvn exec:java -D exec.mainClass=org.pr2.App -Dexec.args=show
```

6.3. La libreta de contactos ejecutable con *java -jar*

Ejercicio 20. Lleve a cabo los siguientes apartados:

1. Añada el siguiente código al *pom.xml* en la sección de *plugins*:

```
<plugin>
   <artifactId>maven-assembly-plugin</artifactId>
       <configuration>
           <archive>
               <manifest>
                   <mainClass>org.pr2.App</mainClass>
               </manifest>
           </archive>
           <descriptorRefs>
               <descriptorRef>jar-with-dependencies</
                   descriptorRef>
           </descriptorRefs>
       </configuration>
</plugin>
```

2. Genere el *.jar* invocando el *plugin*:

```
mvn compile assembly:single
```

3. Compruebe el funcionamiento de la aplicación:

```
java -jar target/libreta-contactos-1.0-SNAPSHOT-jar-with-
    dependencies.jar show
```

6.4. La libreta de contactos con uso de una biblioteca *jar*

Para que funcione el software generado en el presente ejercicio, es necesario haber incluido el plugin *maven-assembly-plugin* (véase ejercicio 20).

Ejercicio 21. Se pide realizar los apartados que se muestran a continuación:

1. Acceda al repositorio central de Maven:

   ```
   https://mvnrepository.com/repos/central
   ```

2. Busque la biblioteca *jopendocument*.

3. Incluya el código de la dependencia en el *pom.xml*.

4. Modifique el código de la clase *Libreta.java*

```
package org.pr2.dominio;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;

import org.jopendocument.dom.spreadsheet.SpreadSheet;

/**
 * Clase responsable de mantener la libreta de contactos.
 */
public class Libreta
{
        private String nombreFichero = "contactos.txt";
        private ArrayList<Contacto> contactos = new ArrayList
            <>();

        /**
         * Genera una cadena de caracteres a partir de la
             libreta.
         * @return la cadena de caracteres.
         */
        @Override
        public String toString() {
                StringBuilder sb = new StringBuilder();
                for(Contacto contacto : contactos)
                {
                        sb.append(contacto + "\n");
                }
                return sb.toString();
        }

        /**
         * Lee la libreta con sus contactos del fichero <i>
             contactos.txt</i>.
         */
        public Libreta() {
                try
                {
```

```
                File fichero = new File(nombreFichero
                    );
                fichero.createNewFile();
                Scanner sc = new Scanner(fichero);
                while(sc.hasNext())
                {
                        Contacto contacto = new
                            Contacto();
                        contacto.setNombre(sc.
                            nextLine());
                        contacto.setTelefono(sc.
                            nextLine());
                        contactos.add(contacto);
                }
        }catch(IOException ex)
        {
                System.err.println(ex);
        }
}

private void volcarContactos()
{
        System.out.println(contactos);
        try
        {
                FileWriter fw = new FileWriter(
                    nombreFichero);
                for(Contacto contacto : contactos)
                {
                        fw.write(contacto.getNombre()
                            +"\n");
                        fw.write(contacto.getTelefono
                            ()+"\n");
                }
                fw.close();
        }catch(IOException ex)
        {
                System.err.println(ex);
        }
}

/**
 * Añade el <i>contacto</i> a la libreta.
 */
public void annadirContacto(String nombre, String
    telefono)
{
        Contacto contacto = new Contacto(nombre,
            telefono);
        contactos.add(contacto);
        this.volcarContactos();
}

/**
```

```
         * Borra el contacto cuyo nombre es el indicado en el
             parámetro
         *
         * @param nombre del contacto a borrar.
         */
        public void borrarContacto(String nombre)
        {
                Contacto contacto = new Contacto();
                contacto.setNombre(nombre);
                contactos.remove(contacto);
                this.volcarContactos();
        }

    /**
     * Genera una hoja de cálculo
     */
    public void generarHojaDeCalculo()
    {
        final Object[][] datos = new Object[contactos.size()
            ][2];
        int i = 0;
        for(Contacto contacto : contactos)
        {
            datos[i][0] = contacto.getNombre();
            datos[i++][1] = contacto.getTelefono();
        }

        String[] columnas = new String[] { "Nombre", "Telé
            fono" };

        TableModel modelo = new DefaultTableModel(datos,
            columnas);

        final File file = new File("output/contactos.ods");
        try
        {
            SpreadSheet.createEmpty(modelo).saveAs(file);
        }catch(IOException ex)
        {
            System.out.println(ex);
        }
    }

}
```

5. Modifique el código de la clase *Interfaz.java*

```
package org.pr2.interfaces;

import org.pr2.dominio.*;

/**
 * Implementa una interfaz texto para la libreta de contactos
     .
```

```
 */
public class Interfaz
{
    /**
     * Inicia la interfaz con parámetros.
     * @param args puede ser <i>add nombre contacto</i> (p.ej
         . <i>add Juan
     * 653421367</i>) para añadir contacto, o <i>show</i>
         para mostrar todos
     * los contactos de la libreta.  */
    public static void iniciar(String args[])
    {
        Libreta libreta = new Libreta();
        if (args[0].equals("add"))
        {
            libreta.annadirContacto(args[1], args[2]);
        }
        else if (args[0].equals("rm"))
        {
            libreta.borrarContacto(args[1]);
        }
        else if (args[0].equals("show")) System.out.println(
            libreta);
        else if (args[0].equals("hoja"))
        {
            libreta.generarHojaDeCalculo();
            System.out.println("Hoja de cálculo generada en
                output/contactos.ods");
        }
        else System.out.println("Opción incorrecta");
    }
}
```

6. Compruebe que funciona correctamente:

```
mvn clean compile assembly:single
java -jar target/libreta-contactos-1.0-SNAPSHOT-jar-with-
    dependencies.jar hoja
```

7. Genere el *javadoc* de la aplicación:

```
mvn javadoc:javadoc
```

6.5. Arquetipos

Ejercicio 22. Acceda a la siguiente página Web y lea el inventario de arquetipos disponibles para Maven:

`https://maven.apache.org/archetypes/index.html`

Ejercicio 23. Cree un proyecto para una aplicación Web y compruebe su funcionamient:

```
mvn archetype:generate -DgroupId=org.pr2 -DartifactId=web-simple -
    DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion
    =1.4 -DinteractiveMode=false
cd web-simple/
mvn package
firefox target/web-simple/index.jsp &
```

Ejercicio 24. El propósito de este ejercicio es que el estudiante aprenda a elaborar su propio arquetipo. Se pide llevar a cabo los siguientes pasos:

1. Genere el arquetipo[2]:

   ```
   mvn archetype:generate -DarchetypeGroupId=org.apache.maven.
       archetypes -DarchetypeArtifactId=maven-archetype-
       archetype -DarchetypeVersion=1.4
   ```

   La respuestas a las preguntas que le formule Maven son las siguientes:

   - $groupId = org.pr2$
   - $artifactId = mi - arquetipo$
   - $version = 1.0$
   - $'package' org.pr2$: Se deja vacío.
   - $Y : Y$

2. Entre el directorio que se acaba de crear:

   ```
   cd mi-arquetipo
   ```

3. Abra el fichero

   ```
   src/main/resources/archetype-resources/pom.xml
   ```

4. Añada el siguiente *plugin*

   ```
   <plugin>
       <artifactId>maven-assembly-plugin</artifactId>
          <configuration>
              <archive>
                  <manifest>
                     <mainClass>org.pr2.App</mainClass>
                  </manifest>
              </archive>
                  <descriptorRefs>
                      <descriptorRef>jar-with-dependencies</
                          descriptorRef>
                  </descriptorRefs>
        </configuration>
   </plugin>
   ```

[2] https://maven.apache.org/archetypes/maven-archetype-archetype/

5. Valide el código:

   ```
   mvn validate
   ```

6. Instale el arquetipo en el repositorio local:

   ```
   mvn install
   ```

7. Explore el siguiente directorio a partir del $HOME:

   ```
   .m2/repository/org/pr2/mi-arquetipo/1.0/
   ```

8. Con el propósito de que pueda tener disponible el arquetipo creado en otros ordenadores o, en caso de que trabaje en modo kiosco en el laboratorio, pueda tenerlo disponible de una clase para otra, se recomienda también mantener los fuentes del arquetipo en un repositorio Git. Para ello, puede seguir los pasos que se muestran a continuación:

   a) Cree el repositorio remoto en Bitbucket.

   b) Clone el repositorio en el directorio donde ha creado el arquetipo.

   c) Cree un *.gitignore* para descartar el contenido del directorio *target* así como de los ficheros *.swp* temporales de Vi:

   ```
   /target/
   .*
   !/.gitignore
   ```

   d) Añada todos los ficheros y directorios del arquetipo:

   ```
   git add .
   ```

   e) Realice el *commit* y el *push*:

   ```
   git commit -am "Versión 1.0 de mi arquetipo"
   git push -u origin master
   ```

9. Use el arquetipo. Para ello, en primer lugar, vaya a otro directorio y ejecute:

   ```
   mvn archetype:generate -DgroupId=org.pr2 -DartifactId=prueba
       -DarchetypeGroupId=org.pr2 -DarchetypeArtifactId=mi-
       arquetipo -DarchetypeVersion=1.0 -DinteractiveMode=false
   ```

10. Compruebe el resultado de

    ```
    cd prueba
    tree
    ```

11. Luego, genere el *.jar*:

    ```
    mvn clean compile assembly:single
    ```

12. Compruebe que ejecuta el programa resultante:

    ```
    java -jar target/prueba-1.0-jar-with-dependencies.jar
    ```

7. Prueba de programas

Ejercicio 25. Explique los siguientes conceptos:

1. Prueba de caja blanca.
2. Prueba de caja negra.
3. Prueba unitaria.
4. Prueba de integración.
5. Prueba de sistema.
6. Prueba de aceptación.

Ejercicio 26. Diseñe pruebas para cada uno los programas del ejercicio 2.

Ejercicio 27. El propósito del presente ejercicio es aprender a utilizar la biblioteca JUnit para implementar pruebas unitarias de software. Se pide llevar a cabo los siguientes apartados:

1. Cree un proyecto Maven.

```
mvn archetype:generate -DgroupId=org.pr2 -DartifactId=
    matematicas -DarchetypeArtifactId=maven-archetype-
    quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false
```

2. Entre el directorio *matematicas*.

```
cd matematicas
```

3. Realice la siguiente creación de directorio

```
md src/main/java/org/pr2/matematicas
```

4. Dentro del directorio recién creado cree el fichero *Matematicas.java* con el siguiente contenido:

```
package org.pr2.matematicas;

public class Matematicas{
    public static int sumaNumeros(int n){
        if (n < 0) throw new ArithmeticException("La suma
            hasta el n-ésimo número natural no es aplicable a
             un n menor que 0");
        return n == 0?0:n + sumaNumeros(n - 1);
    }
}
```

5. Cree un directorio para las pruebas para el paquete *matematicas*:

```
md src/test/java/org/pr2/matematicas
```

6. Dentro del directorio creado, cree el fichero *MatematicasTest.java* con el siguiente contenido, y explíquelo:

```
package org.pr2.matematicas;

import org.junit.Test;
import static org.junit.Assert.*;

public class MatematicasTest{

    @Test
    public void sumaCero(){
        assertEquals(0, Matematicas.sumaNumeros(0));
    }

    @Test
    public void sumaPositivo(){
        assertEquals(6, Matematicas.sumaNumeros(3));
    }

    @Test(expected = ArithmeticException.class)
    public void sumaNegativo(){
        Matematicas.sumaNumeros(-1);
    }

}
```

7. Ejecute los tests:

```
mvn test
```

   Para que sólo ejecute los tests de la clase `Matematicas.java`, puede ejecutar `mvn -Dtest=MatematicasTest test`

Ejercicio 28. Implemente, utilizando JUnit, las pruebas del ejercicio 26.

8. Desarrollo dirigido por pruebas

Ejercicio 29. El propósito del presente ejercicio es mostrar cómo se lleva a cabo el desarrollo dirigido por pruebas (TDD) mediante un ejemplo sencillo. Se pide realizar los siguientes apartados:

1. Elabore un test para comprobar que la suma de los n primeros enteros positivos siendo $n = 1$ es 1, y **véalo fallar**.
2. Escriba el código **mínimo** del método de sumar los n primeros números con el propósito de que pase el test del apartado 1.
3. Elabore un test para la suma de los n primeros enteros positivos siendo $n > 1$, y véalo fallar.

4. Modifique el código del método de sumar los n primeros números enteros con el propósito de que pase el test del apartado 3.

5. Añada los tests que considere oportunos de tal forma que, para cada uno ellos, primero lo vea fallar, y luego lo haga funcionar realizando los cambios oportunos sobre el código del método de la suma de los n primeros enteros.

Ejercicio 30. Realice el ejercicio 2 utilizando la técnica de TDD.

Ejercicio 31. Una progresión geométrica es una sucesión que se caracteriza por que el cociente entre dos términos consecutivos es una cantidad constante a la que se llama razón (`http://www.ematematicas.net/sucesiones.php?a=4&tipo=geometrica`). Por ejemplo, la sucesión 1, 2, 4, 8, 16, ... es una progresión geométrica de razón 2, pues $2/1 = 2$, $4/2 = 2$, $8/4 = 2$, etc.

Se pretende codificar, de acuerdo con la técnica de desarrollo dirigido por pruebas (TDD), un método que calcule la suma de los n primeros términos de una progresión geométrica. Por ejemplo, la suma de los 3 primeros términos de la sucesión 1, 2, 4, 8, 16, ... es 7.

Se pide la secuencia de tests y de versiones del método a desarrollar según TDD que culmine con el método implementado que pase todos los tests. No se considerará válida una respuesta que aporte el código correcto del programa, pero que no siga correctamente los pasos del TDD.

9. Complejidad de algoritmos

A lo largo de los siguientes ejercicios puede ayudarse, cuando lo crea conveniente, de la instrucción `time` de Linux, por ejemplo, ejecutando:

```
time java Principal
```

Ejercicio 32. ¿Qué es un algoritmo?

Ejercicio 33. Escriba dos algoritmos en Java y esos mismos dos algoritmos en C para resolver el mismo problema. Por ejemplo, puede escribir un algoritmo recursivo y otro iterativo (con un bucle) para resolver el problema de la suma de los n primeros números naturales.

Ejercicio 34. La fórmula para calcular el espacio recorrido por un móvil que se deja caer al vacío (suponiendo $v_0 = 0$) es $e = \frac{1}{2}gt^2$, donde g es la aceleración de la gravedad en la superficie de la tierra, y t el tiempo que está cayendo el móvil. ¿Cuál es la complejidad temporal de este cálculo en función de t?

Ejercicio 35. Indique la complejidad temporal asintótica de los siguientes métodos:

```
public static String primero(ArrayList<String> lista)
{
    return lista.get(0);
}

public static String nEsimo(ArrayList<String> lista, int n)
{
    return lista.get(n);
}
```

Ejercicio 36. Calcule la complejidad temporal de los algoritmos del ejercicio 33.

Ejercicio 37. Resuelva cualquiera de los apartados del ejercicio 11 y calcule su complejidad temporal.

Ejercicio 38. Calcule la complejidad temporal y espacial de cualquiera de los algoritmos (recursivos) del ejercicio 2 (salvo los referentes a la serie de Fibonacci). Compare dichas complejidades con el algoritmo iterativo para resolver el mismo problema.

Ejercicio 39. Sea un conjunto A con cardinalidad n, y sea l un algoritmo que ejecuta una instrucción para cada elemento del producto cartesiano de $A \times A$. Calcule la complejidad temporal de l en función de n.

Ejercicio 40. Calcule la complejidad temporal del siguiente método:

```
public static double sumaEltosMatriz(double matriz[][])
{
    double suma = 0;
    for(int i = 0; i < matriz.length; i++)
    {
        for(int j = 0; j < matriz[i].length; j++)
        {
            suma+=matriz[i][j];
        }
    }

    return suma;

}
```

Ejercicio 41. Escriba un algoritmo que busque un número en un *array* de enteros. Calcule su complejidad temporal en el caso peor, en el caso mejor y en el caso promedio. Su cabecera será la siguiente:

```
public static boolean buscar(int e, int[] array)
```

Ejercicio 42. Escriba un algoritmo recursivo para buscar un número en un *array* ordenado de enteros. Su cabecerá será la misma que la del ejercicio 41. Calcule su complejidad en el caso peor.

Ejercicio 43. Calcule las complejidades temporal y espacial del algoritmo recursivo para calcular el elemento n-ésimo de la sucesión de Fibonacci.

Ejercicio 44. Se tiene el siguiente método:

```
public static int sumaNPrimeros(int n) {
    int suma = 0;
    for (int i = 1; i <= n; i++) {
        suma += i;
    }

    return suma;
}
```

Utilizando el *profiler* de Netbeans se han medido los tiempos de ejecución de diferentes llamadas al método (véase el cuadro 1). Explique los resultados.

Cuadro 1: Tiempos de ejecución del método del ejercicio 44

Llamada	Tiempo de ejecución
`sumaNPrimeros(100)`	0,003 ms
`sumaNPrimeros(1000)`	0,013 ms
`sumaNPrimeros(10000)`	0,131 ms
`sumaNPrimeros(100000)`	1,20 ms
`sumaNPrimeros(1000000)`	3,27 ms
`sumaNPrimeros(10000000)`	6,19 ms
`sumaNPrimeros(100000000)`	36 ms
`sumaNPrimeros(1000000000)`	325 ms

Ejercicio 45. Se tiene el siguiente método:

```
public static int sumaNMPrimeros(int n) {
    int suma = 0;
    for (int i = 1; i <= n; i++) {
        for(int j = 1; j <= i; j++)
        suma += j;
    }

    return suma;
}
```

Utilizando el *profiler* de Netbeans se han medido los tiempos de ejecución de diferentes llamadas al método (véase el cuadro 2). Explique los resultados.

Cuadro 2: Tiempos de ejecución del método del ejercicio 45

Llamada	Tiempo de ejecución
`sumaNPrimeros(100)`	0,085 ms
`sumaNPrimeros(1000)`	2,16 ms
`sumaNPrimeros(10000)`	18,1 ms
`sumaNPrimeros(100000)`	1531 ms

Ejercicio 46. Explique la definición que se muestra a continuación
Sean dos funciones $T : \mathbb{N} \to \mathbb{N}$ y $f : \mathbb{N} \to \mathbb{N}$. Se dice que $T(n)$ es de orden $f(n)$, y se escribe $T(n) \in O(f(n))$, si y sólo si existen dos números naturales k y n_0 tales que, para todo m, también natural, que cumpla $m > n_0$, entonces $T(m) \leq k \cdot f(m)$.

Ejercicio 47. Asumiendo la definición del ejercicio 46, se pide:

1. Encontrar k y n_0 que muestren que la siguiente función, $T : \mathbb{N} \to \mathbb{N}$, es de orden $O(log_2(n))$.

$$T(n) = 3 \cdot log_2(n) + 2.$$

2. ¿Si $T(n) \in O(log_2(n))$, entonces $T(n) \in O(n)$? Justifique la respuesta.
3. ¿Si $T(n) \in O(log_3(n))$, entonces $T(n) \in O(log_2(n))$? Justifique la respuesta.

OBSERVACIÓN: en este ejercicio no es necesario que utilice la calculadora.

Ejercicio 48. Estudie de forma comparativa entre ellas el crecimiento de las siguientes funciones reales de variable real:

1. $f_0(x) = 1$,
2. $f_1(x) = x$,
3. $f_2(x) = x^2$,
4. $f_3(x) = log_2(x)$, y
5. $f_4(x) = 2^x$.

Ejercicio 49. Calcule la complejidad temporal asintótica del método f:

```
public static int f(int n) {
    if (n == 0) return 1;
    else if (n < 0) return -1;
    else{
        int m = 1/f(n/2) + f(n/2);
        return sumaNPrimeros(m);
    }
}
```

Para los cálculos, suponga que no hay en ningún momento desbordamiento de pila, y no tenga en cuenta los efectos sobre la ejecución que pueda suponer el desbordamiento de un entero.

Ejercicio 50. La complejidad en el caso peor de la inserción en un *array list* es diferente si el *array list* está ordenado de si no lo está. ¿Es cierta esta afirmación? Justifique la respuesta.

Ejercicio 51. Suponga que una instrucción tarda en ejecutarse 10 ns, y que el tamaño de la entrada es $n = 100$, se pide calcular el tiempo requerido para los siguientes números de ejecuciones:

1. $log(n)$,
2. n,
3. $nlog(n)$,
4. n^2,
5. n^8 y
6. 10^n.

Realice los cálculos anteriores, pero ahora bajo los siguientes supuestos:

1. $n = 100.000$.
2. $n = 100.000$ y el tiempo de instrucción (o bloque de instrucciones) 1 ms.

Ejercicio 52. Explique por qué el problema del ajedrez todavía no está resuelto.

Ejercicio 53. Calcule la complejidad espacial de los algoritmos que ha programado para los apartados 1 al 12 del ejercicio 2, ambos inclusive.

Ejercicio 54. Explique los siguientes conceptos:

- Problema P.
- Problema NP.
- Problema NP-completo.

10. Recursividad, desarrollo dirigido por pruebas y complejidad

Ejercicio 55. Una forma de aproximar la función e^x es mediante polinomios de Taylor utilizando el siguiente sumatorio:

$$e^x = \sum_{k=0}^{n} \frac{x^k}{k!},$$

donde x es el valor del exponente para el que se desea realizar la aproximación, y n el número de sumandos que se desea calcular.

Se pide:

1. Implementar un método iterativo a partir de la aproximación anterior para $n = 10$ utilizando la técnica de desarrollo dirigido por pruebas (TDD). No se considerará válida ninguna solución en que se obtenga el método sin haber aplicado correctamente TDD.

2. Implementar un método recursivo a partir de la aproximación anterior. No es necesario que aplique TDD. En lo referente al tratamiento de n, se considerará válida la solución tanto si se deja variable como si se fija $n = 10$.

3. ¿Cuál es la relación entre la exactitud de la aproximación implementada en Java y el valor de n? Razone la respuesta.

Ejercicio 56. El siguiente código calcula la suma de los elementos de una pila de *big decimals*:

```
        package ejercicios;

        import java.math.BigDecimal;
        import java.util.Stack;

        public class Ejercicio {

    static public BigDecimal sumaIterativaElementosPila(Stack pila)
    {
        BigDecimal suma = BigDecimal.ZERO;
        while(!pila.empty())
        {
            //pila.peek() devuelve la cima de la pila.
            suma = suma.add((BigDecimal) pila.peek());
            pila.pop();
        }

        return suma;
    }
}
```

Se pretende codificar, de acuerdo con la técnica de desarrollo dirigido por pruebas (TDD), el método *static public BigDecimal sumaRecursivaElementosPila(Stack pila)* de tal forma que realice la misma función que el anterior, pero de manera recursiva.

Se pide:

1. La secuencia de tests y de versiones del método a desarrollar según TDD que culmine con el método implementado que pase todos los tests. En la respuesta del estudiante debe quedar claro qué versión del software hace pasar cada test.

2. El análisis de la complejidad temporal asintótica tanto del método dado en el enunciado como del método que sea la versión final del estudiante.

11. Expresiones lambda, desarrollo dirigido por pruebas y complejidad

Ejercicio 57. La media armónica de una serie de datos $x_1, x_2, ..., x_n$ viene definida por:

$$H = \frac{n}{\sum_{k=1}^{n} \frac{1}{x_k}},$$

No está definida si alguno de los x_i es cero, o si $\sum_{k=1}^{n} \frac{1}{x_k} = 0$.
Se pide:

1. Implementar mediante expresiones lambda un método para calcular la media armónica de un conjunto de datos dados en una lista utilizando la técnica de desarrollo dirigido por pruebas (TDD). La cabecera del método será la siguiente:

```
double mediaArmonica(List<Double> datos)
```

   Debe tener en cuenta que se debe elevar una excepción *ArithmeticException* si la lista cumple alguna de las siguientes propiedades:

   a) está vacía;

   b) contiene algún elemento que sea cero; o

   c) la suma de sus inversos es cero.

2. Calcular la complejidad asintótica temporal del algoritmo que se muestra a continuación. Su pseudocódigo sólo es válido para conjuntos de datos no vacíos que no contienen el cero ni valores próximos a cero, y tal que la suma de sus inversos tampoco es cero.

```
function MediaArmónica(datos)
    suma ← 0
    for dato in datos do
        suma ← suma + 1/dato
    end for
     return datos.length/suma
end function
```

12. Tipos abstractos de datos y estructuras de datos

Ejercicio 58. Explique la diferencia entre tipo abstracto de datos (TAD) y estructura de datos.

12.1. Listas

Ejercicio 59. Explique qué es una lista y cuáles son sus operaciones típicas.

Ejercicio 60. Explique las diferentes formas de implementar una lista.

Ejercicio 61. Explique las ventajas e inconvenientes del *ArrayList* frente al *LinkedList*.

Ejercicio 62. Examine los siguientes códigos fuente e identifique la relación entre ellos:

1. *Collection.java*.
2. *List.java*.
3. *ArrayList.java*.
4. *LinkedList.java*.

12.2. Pilas

Ejercicio 63. Explique qué es una pila y cuáles son sus operaciones típicas.

Ejercicio 64. Indique dos aplicaciones en que se utilzan pilas.

Ejercicio 65. Explique la relación existente entre el TAD lista y el TAD pila.

Ejercicio 66. Examine el código fuente de las clases *Stack.java* y *Vector.java*. Identifique la relación entre ellos y con *ArrayList.java*.

Ejercicio 67. Ejecute y explique el siguiente código:

```
    import java.util.Stack;

    public class DemoPilas
    {
        public static void main(String[] args)
    {
        Stack<Integer> pila = new Stack<>();

        pila.push(1);
        pila.push(2);
        System.out.println(pila);
        System.out.println(pila.peek());
        pila.pop();
        System.out.println(pila);
    }
}
```

12.3. Colas

Ejercicio 68. Explique qué es una cola y cuáles son sus operaciones típicas.

Ejercicio 69. Indique tipos de aplicaciones en que se utilicen colas.

Ejercicio 70. Examine el código de la clase *ArrayDequeue.java* y explique cómo aplica la aritmética modular.

Ejercicio 71. Ejecute y explique el siguiente código, y compare su comportamiento con el programa del ejercicio 67:

```
import java.util.ArrayDeque;

public class DemoColas
{
    public static void main(String[] args)
    {
        ArrayDeque<Integer> cola = new ArrayDeque<>();
        cola.add(1);
        cola.add(2);
        System.out.println(cola);
        System.out.println(cola.peek());
        System.out.println(cola.poll());
        System.out.println(cola);
    }
}
```

12.4. Ordenación de estructuras lineales

12.4.1. Inserción directa

Ejercicio 72. Explique apoyándose en un ejemplo cómo funciona el método de inserción directa.

Ejercicio 73. Calcule la complejidad asintótica temporal para el peor caso del método de inserción directa.

Ejercicio 74. ¿Cuándo se usa el método de inserción directa? Justifique la respuesta.

Ejercicio 75. Explique el código del algoritmo de inserción directa proporcionado por el libro que aparece en la bibliografía básica de la asignatura.

12.4.2. Mergesort

Ejercicio 76. Explique apoyándose en un ejemplo cómo funciona el método *mergesort*. ¿A quién se atribuye la invención del método?

Ejercicio 77. Calcule la complejidad asintótica temporal para el peor caso del método *mergesort*.

Ejercicio 78. Explique el código del algoritmo *mergesort* proporcionado por el libro que aparece en la bibliografía básica de la asignatura.

12.4.3. Timsort

Ejercicio 79. Realice los siguientes apartados sobre el algoritmo *timsort*:

1. Explique qué es un *run*.
2. Explique qué es el *minrun*.
3. Explique a partir de un ejemplo cómo funciona el algoritmo de inserción binaria.
4. Explique con ejemplos cómo funciona el *timsort*.
5. ¿Quién inventó el algoritmo?
6. ¿Qué complejidad tiene?

12.4.4. Quicksort

Ejercicio 80. Explique apoyándose en un ejemplo cómo funciona el método *quicksort*. ¿Quién inventó el método?

Ejercicio 81. Explique el código del algoritmo *quicksort* proporcionado por el libro que aparece en la bibliografía básica de la asignatura.

Ejercicio 82. Calcule las siguientes complejidades del método *quicksort*:

1. Caso mejor.
2. Caso peor.
3. Caso promedio.

Ejercicio 83. ¿Qué significa que el *quicksort* no es estable?

12.4.5. Ejemplos de ejecución de algoritmos de ordenación

Ejercicio 84. Para cada uno de los siguientes *arrays*, muestre su contenido a través de los diferentes pasos para los algoritmos de inserción directa, Mergesort y Quicksort.

1. (5, 3, 4, 8, 1, 9).
2. (1, 5, 2, 4, 7, 12, 10, 9).
3. (1, 4, 7, 9, 2, 5, 8, 10).

12.4.6. Límite inferior en la complejidad de la ordenación

Ejercicio 85. Se pide calcular el límite inferior de la complejidad de los algoritmos de ordenación basados en comparaciones.

12.4.7. Los métodos *sort* en Java

Ejercicio 86. Explique qué algoritmos de ordenación utiliza Java y en qué casos.

Ejercicio 87. Escriba, compile y ejecute el siguiente código y explíquelo:

```
    import java.util.Arrays;
    import java.util.Collections;
    import java.util.List;

    public class CompararEnteros
    {
        public static void main(String[] args)
    {
        List<Integer> lista = Arrays.asList(23, -2, 14, 3, 12, 8);
        Collections.sort(lista);
        System.out.println(lista);
    }
}
```

Ejercicio 88. Escriba el siguiente programa y ejecútelo:

```
import java.time.LocalDate;
import java.time.Period;
import java.time.format.DateTimeFormatter;

public class Persona implements Comparable<Persona>{
    private String nombre;
    private LocalDate fechaNacimiento;

    public String getNombre() {
```

```
        return nombre;
    }

    public int compareTo(Persona persona){
        return this.nombre.compareTo(persona.nombre);
    }

    @Override
    public String toString() {
        DateTimeFormatter formatter = DateTimeFormatter.
            ofPattern("dd/MM/yyyy");
        return "Persona{" +
            "nombre = " + getNombre() +
            "fecha nacimiento = " +  fechaNacimiento.format(
                formatter);
    }

    public Persona(String nombre, int dia, int mes, int anno) {
        this.nombre = nombre;
        this.fechaNacimiento = LocalDate.of(anno, mes, dia);
    }

    public LocalDate getFechaNacimiento() {
        return fechaNacimiento;
    }

    public int calcularEdad(){
        return Period.between(fechaNacimiento, LocalDate.now())
            .getYears();
    }
}
```

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Principal{
    public static void main(String[] args)
    {
        Persona juan = new Persona("Juan García Rodríguez", 13,
             2, 1980);
        Persona pepe = new Persona("José López Pérez", 4, 10,
            1950);
        Persona maria = new Persona("María Sánchez Martínez",
            5, 7, 2004);
        List<Persona> lista = Arrays.asList(juan, pepe, maria);
        Collections.sort(lista);
        System.out.println("Ordenación por nombres: " + lista);
    }
```

```
}
```

Ejercicio 89. Complemente el ejercicio 89 con el código de la siguiente clase:

```
import java.util.Comparator;

public class ComparadorEdades implements Comparator<Persona>{
    public int compare(Persona persona1, Persona persona2){
        return persona1.calcularEdad() - persona2.calcularEdad
            ();
    }
}
```

Y modifique la clase `Principal.java` para que quede tal y como se muestra a continuación:

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Principal{
    public static void main(String[] args)
    {
        Persona juan = new Persona("Juan García Rodríguez", 13,
             2, 1980);
        Persona pepe = new Persona("José López Pérez", 4, 10,
            1950);
        Persona maria = new Persona("María Sánchez Martínez",
            5, 7, 2004);
        List<Persona> lista = Arrays.asList(juan, pepe, maria);
        Collections.sort(lista);
        System.out.println("Ordenación por nombres: " + lista);
        Collections.sort(lista, new ComparadorEdades());
        System.out.println("Ordenación por edades: " + lista);
    }
}
```

12.5. Árboles

Ejercicio 90. Defina los términos *árbol* y *árbol binario*, y explique aplicaciones de ambas estructuras.

Ejercicio 91. Simule la inserción en un árbol binario de los siguientes elementos: 2, 1, 3, 8, 5, 2.

Ejercicio 92. Explique qué reglas se siguen para borrar un elemento de un árbol binario.

Ejercicio 93. Sea el siguiente código que implementa un árbol binario (la mayor parte del código ha sido tomado de `http://cslibrary.stanford.edu/110/BinaryTrees.html#java`):

```
    // BinaryTree.java
    public class BinaryTree {

// Root node pointer. Will be null for an empty tree.
private Node root;


/*
 --Node--
 The binary tree is built using this nested node class.
 Each node stores one data element, and has left and right
 sub-tree pointer which may be null.
 The node is a "dumb" nested class -- we just use it for
 storage; it does not have any methods.
 */
private static class Node {

    Node left;
    Node right;
    int data;

    Node(int newData) {
        left = null;
        right = null;
        data = newData;
    }
}

/**
 * Creates an empty binary tree -- a null root pointer.
 */
public void BinaryTree() {
    root = null;
}

/**
 * Returns true if the given target is in the binary tree. Uses a recursive
 * helper.
 *
 * @param data
 * @return
 */
public boolean lookup(int data) {
    return (lookup(root, data));
}
```

```
/**
 * Recursive lookup -- given a node, recur down searching for the given
 * data.
 */
private boolean lookup(Node node, int data) {
    if (node == null) {
        return (false);
    }

    if (data == node.data) {
        return (true);
    } else if (data < node.data) {
        return (lookup(node.left, data));
    } else {
        return (lookup(node.right, data));
    }
}

/**
 * Inserts the given data into the binary tree. Uses a recursive helper.
 *
 * @param data
 */
public void insert(int data) {
    root = insert(root, data);
}

/**
 * Recursive insert -- given a node pointer, recur down and insert the given
 * data into the tree. Returns the new node pointer (the standard way to
 * communicate a changed pointer back to the caller).
 */
private Node insert(Node node, int data) {
    if (node == null) {
        node = new Node(data);
    } else {
        if (data <= node.data) {
            node.left = insert(node.left, data);
        } else {
            node.right = insert(node.right, data);
        }
    }

    return (node); // in any case, return the new pointer to the caller
}

public int size() {
    return (size(root));
}

private int size(Node node) {
    if (node == null) {
        return (0);
    } else {
```

```
        return (size(node.left) + 1 + size(node.right));
    }
}

/**
 * Returns the max root-to-leaf depth of the tree. Uses a recursive helper
 * that recurs down to find the max depth.
 *
 * @return
 */
public int maxDepth() {
    return (maxDepth(root));
}

private int maxDepth(Node node) {
    if (node == null) {
        return (0);
    } else {
        int lDepth = maxDepth(node.left);
        int rDepth = maxDepth(node.right);

        // use the larger + 1
        return (Math.max(lDepth, rDepth) + 1);
    }
}

/**
 * Returns the min value in a non-empty binary search tree. Uses a helper
 * method that iterates to the left to find the min value.
 *
 * @return
 */
public int minValue() {
    return (minValue(root));
}

/**
 * Finds the min value in a non-empty binary search tree.
 */
private int minValue(Node node) {
    Node current = node;
    while (current.left != null) {
        current = current.left;
    }

    return (current.data);
}

/**
 * Prints the node values in the "inorder" order. Uses a recursive helper to
 * do the traversal.
 *
 * @return
 */
```

```
@Override
public String toString() {
    return aStringBuilder(root).toString();
}

private StringBuilder aStringBuilder(Node node) {
    StringBuilder sb = new StringBuilder("");
    if (node == null) {
        return sb;
    }

    // left, node itself, right
    return aStringBuilder(node.left).
            append(new StringBuilder(" " + node.data + "  ")).
            append(aStringBuilder(node.right));
}

@Override
public boolean equals(Object other) {
    /*
     Compares the receiver to another tree to
     see if they are structurally identical.
     */
    return (sameTree(root, ((BinaryTree) other).root));
}

/**
 * Recursive helper -- recurs down two trees in parallel, checking to see if
 * they are identical.
 */
boolean sameTree(Node a, Node b) {
    // 1. both empty -> true
    if (a == null && b == null) {
        return (true);
    } // 2. both non-empty -> compare them
    else if (a != null && b != null) {
        return (a.data == b.data
                && sameTree(a.left, b.left)
                && sameTree(a.right, b.right));
    } // 3. one empty, one not -> false
    else {
        return (false);
    }
}

/**
 * Prints the node values in the "postorder" order. Uses a recursive helper
 * to do the traversal.
 */
public void printPostorder() {
    printPostorder(root);
    System.out.println();
}
```

```
    private void printPostorder(Node node) {
        if (node == null) {
            return;
        }

        // first recur on both subtrees
        printPostorder(node.left);
        printPostorder(node.right);

        // then deal with the node
        System.out.print(node.data + "  ");
    }
}
```

Se pide:

1. Mostrar el resultado del siguiente método y dibujar cómo está estructurado el árbol:

```
static void mostrarFuncionamientoArbolesBinarios() {
    BinaryTree bt = new BinaryTree();
    bt.insert(2);
    bt.insert(1);
    bt.insert(3);
    bt.insert(8);
    bt.insert(5);
    bt.insert(2);

    System.out.println(bt);
    bt.printPostorder();
}
```

2. El análisis de complejidad temporal asintótico de cada uno de los métodos públicos que aparecen en el ejercicio.

Ejercicio 94. Explique qué es un árbol rojo-negro y qué propiedades debe satisfacer.

Ejercicio 95. Explique mediante ejemplos cómo se realiza la inserción de un elemento en un árbol rojo-negro.

Ejercicio 96. Explique cómo se realiza el borrado de un nudo rojo en árbol rojo-negro.

Ejercicio 97. Explique qué es un *set* y cómo puede implementarse mediante un árbol binario.

Ejercicio 98. Explique qué es un *map* y cómo puede implementarse mediante un árbol binario.

Ejercicio 99. El propósito del presente ejercicio es mostrar la aplicación de los *set* y los *map* a un problema real. Se pide implementar un validador de doble máster siguiendo los pasos que se muestran a continuación (cree los paquetes que considere oportunos para organizar las diferentes clases):

1. Cree la clase *Asignatura* de tal forma que permita que cualquier asignatura tenga un identificador, un nombre, el número de semestre en que se imparte y el número de créditos ECTS.

2. Cree la clase *Master* de tal forma que permita que cualquier máster tenga un identificador, un nombre y un conjunto de asignaturas.

3. Heredando de la clase *Master*, cree la clase *DobleMaster* de tal forma que permita que cualquier doble máster tenga un identificador, un nombre, un conjunto de asignaturas, una relación con los dos máster que engloba, y una serie de *mappings* entre sus asignaturas y las asignaturas de los dos máster simples.

4. Añada a la clase *DobleMaster* una tabla de convalidaciones entre asignaturas de los máster simples.

5. Utilizando la técnica de TDD, complete la clase *DobleMaster* con un método que permita validar cualquier doble máster según las reglas que se muestran a continuación:

   a) Cobertura del máster 1: para toda asignatura del máster 1, bien pertence el conjunto de asignaturas del doble máster, bien tiene una equivalencia con alguna asignatura del doble máster.

   b) Cobertura del máster 2 (análogo al paso anterior).

   c) Precisión del doble máster: para toda asignatura del doble máster, bien pertenece al máster 1 bien pertenece al máster 2.

   d) Secuenciación correcta: no se da el caso de que una asignatura aparezca en distinto semestre en un máster simple y en el máster doble.

Ejercicio 100. Explique cómo implementaría un árbol en que cada nudo puede tener más de dos hijos.

Ejercicio 101. Averigüe cómo JDOM estructuraría en forma de árbol el siguiente fragmento de HTML:

```
<TABLE>
<TBODY>
<TR>
<TD>Shady Grove</TD>
<TD>Aeolian</TD>
</TR>
<TR>
```

```
<TD>Over the River, Charlie</TD>
<TD>Dorian</TD>
</TR>
</TBODY>
</TABLE>
```

12.6. Hashing

Ejercicio 102. Explique qué es una función *hash*.

Ejercicio 103. Explique qué función *hash* utiliza Java para objetos de la clase *Integer*.

Ejercicio 104. Explique qué función *hash* utiliza Java para objetos de la clase *String*.

Ejercicio 105. Explique cómo puede implementarse un *map* mediante *hashing*.

Ejercicio 106. Explique cómo puede implementarse un conjunto mediante *hashing*.

Ejercicio 107. Explique por qué Java aumenta el tamaño máximo de una tabla *hash* antes de que se llene.

Ejercicio 108. Explique por qué Java recoloca los elementos de una tabla *hash* cuando ésta aumenta de tamaño.

Ejercicio 109. Explique por qué es conveniente especificar el tamaño inicial de una *HashSet* o de un *HashMap* cuando se espera que éste sea grande.

Ejercicio 110. ¿Qué consecuencias tiene el no reescribir el método *hashCode* cuando se define un tipo de elemento para un *HashSet* o *HashMap*?

12.7. Colas con prioridad

Ejercicio 111. Ponga algún ejemplo en que sea conveniente utilizar una cola con prioridad.

Ejercicio 112. Muestre el proceso de inserción de los valores: 5, 7, 8, 1, 4, 3 en una cola con prioridad implementada mediante un *heap*. A continuación, muestre el proceso de sacar el primer elemento de la cola.

Ejercicio 113. Se ha implementado una cola con prioridad mediante un *heap*. En un momento dado, su contenido es el que se muestra en la figura 1. Se pide mostrar el proceso de sacar el primer elemento de la cola.

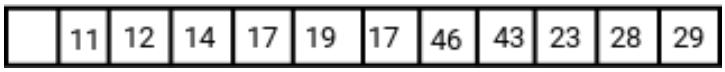

Figura 1: Contenido del *heap* del ejercicio 113

Ejercicio 114. Realice el análisis de complejidad temporal asintótica de las operaciones de inserción y de sacar primero de la cola en la implementación con un *heap* de una cola con prioridad.

Ejercicio 115. ¿Qué clase de Java implementa colas con prioridad con *heap*?

Ejercicio 116. Con la clase identificada en el ejercicio 115 compruebe cómo quedan almacenados los valores del ejercicio 112.

Ejercicio 117. Implemente un programa en Java que inserte los enteros 3, 1 y 5 en un objeto de la clase *PriorityQueue* y que, a continuación, los vaya extrayendo y mostrando por pantalla.

Ejercicio 118. Implemente un programa en Java que inserte las cadenas de carateres "short", "medium" y "very long indeed" en un objeto de la clase *PriorityQueue* y que, a continuación, las vaya extrayendo y mostrando por pantalla. La prioridad se establecerá según la longitud de la cadena. Cuanto más corta sea la cadena, más prioridad debe tener[3].

12.8. Grafos

Ejercicio 119. Defina los términos: *grafo*, *grafo dirigido* (o *digrafo*), cuándo un vértice es *adyacente* a otro, *peso* (o *coste*) de un grafo, *camino*, *longitud* de un camino, *bucle* en un grafo, *ciclo*, *camino simple*, *grafo acíclico* (o *DAG*), *grafo conexo*, *grafo fuertemente conexo*, *grafo débilmente conexo* y *grafo completo*.

Ejercicio 120. Explique diferentes implementaciones que se pueden utilizar para la estructura de grafo.

Ejercicio 121. Sea el grafo de la figura 2. Se pide:

1. Realizar la traza de la búsqueda en profundidad de una ruta que vaya de A a I. Muestre cómo van evolucionando la pila, la tabla con la solución provisional y el árbol. ¿Cuándo se realiza vuelta atrás (*backtracking*)?

2. Realizar la traza la búsqueda en amplitud de una ruta que vaya de A a I. Muestre cómo van evolucionando la cola, la tabla con la solución provisional y el árbol.

3. ¿Qué estructuras de las mencionadas en los apartados 1 y 2 se implementan realmente y cuál es puramente conceptual en este ejercicio?

[3]Ejercicio inspirado en http://stackoverflow.com/questions/683041/java-how-do-i-use-a-priorityqueue

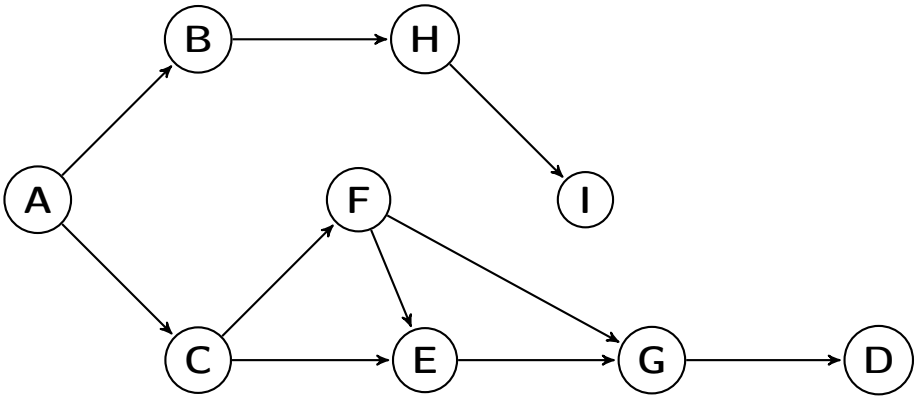

Figura 2: Grafo del ejercicio 121

Ejercicio 122. Sea un grafo con los siguientes vértices: "Albacete", "Almería", "Madrid", "Murcia", "Segovia" y "Toledo", y con los siguientes arcos:

- Madrid-Segovia: 92 km
- Segovia-Toledo: 165 km
- Madrid-Toledo: 72 km
- Madrid-Albacete: 257 km
- Albacete-Toledo: 245 km
- Albacete-Murcia: 146 km
- Murcia-Almería: 218 km
- Toledo-Almería: 500 km

Se pide ejecutar en papel el algoritmo de Dijkstra para obtener el mejor camino desde Madrid a cualquiera de las demás ciudades.

Ejercicio 123. Analice la complejidad temporal asintótica de los siguientes algoritmos:

1. Cálculo del camino entre dos vértices mediante búsqueda en profundidad.
2. Cálculo del camino entre dos vértices mediante búsqueda en anchura.
3. Algoritmo de Dijkstra.

12.9. Iteradores

Ejercicio 124. Escriba, compile, ejecute y explique el siguiente código:

```
    import java.util.HashSet;
    import java.util.Iterator;
    import java.util.Set;

    public class EjemploIterador
    {
        public static void main(String[] args)
    {
        Set<Integer> conjunto = new HashSet<>();
        conjunto.add(1);
        conjunto.add(40);
        conjunto.add(16);
        conjunto.add(3);
        System.out.println("Resultado con forEach()");
        System.out.println("-----------------------");
        conjunto.forEach(e -> System.out.println("Escribiendo " +
            e));
        System.out.println("\nResultado con iteradores");
        System.out.println("\n-----------------------");
        Iterator iterador = conjunto.iterator();
        while(iterador.hasNext())
        {
            System.out.println("Escribiendo " + iterador.next());
        }
    }

}
```

13. Expresiones lambda y estructuras de datos

Ejercicio 125. ¿Qué se muestra por pantalla al ejecutar el método *main* en el código que se presenta a continuación? Justifique la respuesta.

```
        import java.util.HashMap;
        import java.util.HashSet;
        import java.util.Map;
        import java.util.Set;

        public class Main {

            public static int m(Set<Integer> cjto, Map<Integer,
                Integer> map)
        {
            return cjto.stream().filter(e -> map.get(e) != null).
                reduce(0, (a, b) -> a + map.get(b));
        }

        public static void main(String[] args) {
            HashSet<Integer> cjto = new HashSet<>();
```

```
        HashMap<Integer, Integer> map = new HashMap<>();
        cjto.add(1);
        cjto.add(3);
        cjto.add(5);
        cjto.add(2);
        cjto.add(6);
        map.put(3, 4);
        map.put(5, 2);
        map.put(6, 20);
        map.put(7, 34);
        System.out.println(m(cjto, map));
    }

}
```

14. Técnicas algorítmicas

Ejercicio 126. Explique el significado de los siguientes términos en el contexto de la algoritmia:

1. Fuerza bruta.
2. Programación voraz.
3. Divide y vencerás.
4. Programación dinámica.

Ejercicio 127. Implemente en Java el problema de la mochila.

Ejercicio 128. Implemente en Java el cálculo del número del Fibonacci mediante la técnica de programación dinámica. ¿Este algoritmo es óptimo desde el punto de vista de la complejidad espacial? Justifique la respuesta.

Ejercicio 129. Implemente en Java la distancia de edición entre dos palabras según lo expuesto las diapositivas de Dan Jurafsky[4]. No es necesario que almacena la traza para transformar una palabra en otra.

15. Ejercicios extraídos de pruebas objetivas

Ejercicio 130. Recuadre las frases de las siguientes que sean verdaderas:

1. El algoritmo *quicksort* fue elaborado por von Neumann.
2. Un árbol binario no balanceado y una tabla *hash* con un número de colisiones alto tienen la misma complejidad asintótica temporal.

[4] https://web.stanford.edu/class/cs124/lec/med.pdf

3. El método *mergesort* es no estable.

4. Cuando se programa una estructura de datos mediante *polimorfismo paramétrico*, los métodos se escriben abstrayendo los detalles particulares de las clases de los elementos de la estructura.

5. En la implementación del *ArrayList*, el *array* de elementos no es público.

6. `T[] array = new T[tamMax];` en el constructor de un *array list* provoca un error de ejecución, pero no de compilación.

7. La consulta de un árbol rojo-negro tiene una complejidad asintótica temporal logarítmica.

8. Dentro del código *contains* de un *array list*, `array[i].equals(e)` es equivalente a `array[i]==e`.