Problema 1 (7 puntos)

Una empresa de distribución y venta de productos necesita un sistema para hacer las actualizaciones del fichero de clientes respecto de las compras realizadas. Como siempre la solución consistirá en implementar diferentes métodos de diferentes clases que os describiremos en el enunciado.

La clase Product

Las instancias de esta clase representan los diferentes productos que vende la empresa y está definida como:

```
public class Product {
  private static final int NAME_LIMIT = 20; // In chars
  private long id;       // identificador
  private String name;   // nombre
  private int price;     // precio en € (siempre > 0)

  public static final int SIZE = ...; // In bytes

  public Product(long id, String name, int price) {...}

 // getters

  public byte[] toBytes() { ... }
  public static Product fromBytes(byte[] bytes) { ... }
}
```

Esta clase podéis considerar que os la damos completamente implementada.

La clase ProductsDB

Esta clase representa el fichero de productos y, dado un identificador, permitirá acceder a los datos del producto correspondiente. Los productos se encuentran en el fichero ordenados según el valor del identificador (atributo `id`), con valores consecutivos y sin dejar huecos. El primer producto tiene el identificador `0L`.

```
public class ProductsDB {
   private RandomAccessFile raf;
   public ProductsDB(String fname) throws IOException { ... }
   public void close() throws IOException { ... }
   public Product read(long id) throws IOException { ... }
   public void write(Product product) throws IOException { ??? }
   public long numProducts() throws IOException { ... }
}
```

En este apartado

A. **(0,5 puntos)** Implementad el método `write` de la clase `ProductsDB`.

La clase Client

Esta clase representa a cada uno de los clientes. El atributo que más nos interesará es el que se corresponde con el total de las compras realizadas.

```
public class Client {
  private static final int NAME_LIMIT = 20; // In chars
  private long id;       // identificador
  private String name;   // nombre
  private int total;     // total de compras en €

  public static final int SIZE = ...; // In bytes

  public Client(long id, String name) {
    this.id = id; this.name = name; this.total = 0;
  }

 // getters

 public void add(int quantity, int price) {
    // Precondición: quantity > 0 && price > 0
    this.total += quantity * price;
 }

  public byte[] toBytes() { ... }
  public static Client fromBytes(byte[] bytes) { ??? }
}
```

Para esta clase, se pide:

B. **(0,5 puntos)** Dibujad un registro correspondiente a una instancia de la clase `Client` mostrando la correspondencia entre los bytes y los atributos de la instancia. Calculad también el tamaño (`SIZE`) que tienen los registros de la clase `Client`.
C. **(0,5 puntos)** Implementad el método `fromBytes`

La clase ClientsDB

Esta clase representa el fichero de clientes y, dado un identificador, permitirá acceder a los datos del cliente correspondiente. Los clientes se encuentran en el fichero ordenados según el valor del identificador (atributo `id`), con valores consecutivos y sin dejar huecos. El primer cliente tiene el identificador `1L`.

```
public class ClientsDB {
   private RandomAccessFile raf;
   public ClientsDB(String fname) throws IOException { ... }
   public void close() throws IOException { ... }
   public Client read(long id) throws IOException { ??? }
   public void write(Client client) throws IOException { ... }
   public long numClients() throws IOException { ... }
}
```

En este apartado

D. **(0,5 puntos)** Implementad el método read de la clase `ClientsDB`.

El fichero purchases.txt

Este fichero de texto contiene la información de las compras realizadas por cada cliente. Las compras de un cliente vienen detalladas en una línea con el siguiente formato:

id-cliente;id-producto1;cantidad-producto1;id-producto2;cantidad-producto2;...

Es decir, el identificador del cliente seguido de parejas de identificador de producto y cantidad comprada. Como siempre, supondremos que las líneas no contienen errores de formato, aunque los valores pueden ser inválidos, por ejemplo:

- puede no existir un cliente con ese identificador
- puede no existir un producto con ese identificador
- la cantidad puede no ser positiva

En el primer caso, se ignorará toda la línea y, en los otros, la pareja formada por id-producto y cantidad-producto.
Podéis considerar que como mínimo hay una pareja, aunque no necesariamente válida.

La classe Updater

Esta clase representa el programa principal y su cometido es actualizar la información de los clientes en función de las ventas realizadas.

```
public class Updater extends ConsoleProgram {
  private static final String CLIENTS   = “clients.dat”;
  private static final String PRODUCTS  = “products.dat”;
  private static final String PURCHASES = “purchases.txt”;

  private ClientsDB  clientsDB;
  private ProductsDB productsDB;
  private BufferedReader purchases;

  public void run() {
    try {
      openFiles();
      processPurchases();
      closeFiles();
    } catch (IOException ex) {
       println(“Ha habido un error”);
    }
  }

  private void openFiles() throws IOException {
    clientsDB = new ClientsDB(CLIENTS);
    productsDB = new ProductsDB(PRODUCTS);
    purchases = new BufferedReader(new FileReader(PURCHASES));
  }
```

```
    private void closeFiles() throws IOException {
      clientsDB.close(); productsDB.close(); purchases.close();
    }

    private void updateClient(Client client, long idProduct, int quantity)
      throws IOException {
       // Precondición: client != null
       ???
    }

    private void processPurchases() throws IOException { ??? }

  }
```

Es demana:

E. **(1 punto)** Implementad el método `updateClient` que, dado un cliente (que sabemos no es null), actualiza el valor del total de compras del cliente en función del precio del producto cuyo identificador se pasa como parámetro y de la cantidad de producto comprado. Si el `idProduct` no se corresponde con algún producto o la cantidad no es positiva, la operación no hace nada.

F. **(4 puntos)** Implementad, **descomponiéndolo en métodos auxiliares**, el método `processPurchases`, que ejecuta todas las actualizaciones de clientes correspondientes a las líneas del fichero.

Problema 2 (3 puntos)

Se desea diseñar una función recursiva tal que, dado un vector de enteros **no vacío**, devuelva la posición que ocupa el elemento con valor máximo en el vector. En caso de que haya varias posiciones que contengan el máximo, devolved la posición más hacia la derecha..

Es decir, dicha función tendrá la forma:

```
public static int posOfMax(int[] v) {
  // 1 <= v.length
  ???
}
```

Como los vectores no permiten hacer recursividad de forma directa, lo que haremos es diseñar una función que trabaje sobre el prefijo del vector (parte izquierda) formado por los elementos con posiciones anteriores a pos.

```
private static int posOfMax(int[] v, int pos) {
  // 1 <= pos <= v.length
  ???
}
```

Tened en cuenta que **el prefijo indicado por pos tampoco puede ser vacío**.

Se pide que:

A. **(0,5 puntos)** Definid y justificad con un **diagrama** cómo la primera función llama a la segunda para resolver el problema sobre todo el vector.
B. **(1,5 puntos)** Definid y justificad con un **diagrama** el caso recursivo de la segunda función. **Razonad** cómo se ha de adaptar el resultado de la llamada recursiva para encontrar el resultado de la llamada actual. Tened en cuenta que no podéis llamar nunca a la función con un prefijo vacío.
C. **(0,5 puntos)** Definid y justificad con un **diagrama** el caso simple de la segunda función.
D. **(0,5 puntos) Implementad** en Java las funciones diseñadas en los apartados a, b y c.

Apéndice

byte	char	int	long	float	double
8	16	32	64	32	64

- PackUtils
  - static void packDouble(double d, byte[] buffer, int offset)
  - static void packInt(int n, byte[] buffer, int offset)
  - static void packLimitedString(String s, int maxLength, byte[] buffer, int offset)
  - static void packLong(long l, byte[] buffer, int offset)
  - static double unpackDouble(byte[] buffer, int offset)
  - static int unpackInt(byte[] buffer, int offset)
  - static long unpackLong(byte[] buffer, int offset)
  - static String unpackLimitedString(int maxLength, byte[] buffer, int offset)
- class RandomAccessFile (totes llencen IOException)
  - new RandomAccessFile(String fileName, String mode)
  - long getFilePointer()
  - long length()
  - int read(byte[] buff)
  - void seek(long pos)
  - void setLength(long newLength)
  - void write(byte[] buff)
- class BufferedReader (totes llencen IOException)
  - new BufferedReader(FileReader reader)
  - String readLine()
- class FileReader (totes llencen IOException)
  - new FileReader(String fileName)
  - int read()
- class StringTokenizer:
  - new StringTokenizer(String str)
  - new StringTokenizer(String str, String delims)
  - boolean hasMoreTokens()
  - String nextToken()
- class String:
  - static String valueOf(int i)
  - boolean equals(String other)
- class Double
  - static double parseDouble(String s)
- class Integer
  - static int parseInt(String s)
- class Long
  - static long parseLong(String s)