npus de Ourense
32004 Ourense
España
el. 988 387 000
Fax 988 387 001
esei.uvigo.es
sdeuig@uvigo.es

Pregunta 1.- Este año la plataforma Voluntarios Sin Fronteras vuelve a recaudar dinero, en esta ocasión, para ayudar a niños y niñas que viven las consecuencias de conflictos armados. La donación a la plataforma se puede realizar como siempre, llamando gratis al teléfono 900 30 20 10 o bien a través de Bizum al número 11013 (la cantidad máxima que se puede donar es de 500 euros).

A lo largo de su historia, las donaciones a la plataforma han sido muy variadas, pero habitualmente hay ciertas cantidades de dinero que se suelen repetir entre los donantes. Por ese motivo este año se ha decidido analizar más en detalle las donaciones realizadas para buscar cuál es la cantidad de dinero que más se repite. Puede haber más de una donación que cumpla esta condición.

Por ejemplo:
Si las donaciones han sido: 25, 5, 70, 20, 45, 100, 25, 30, 150, 320, 500, 25, 35, 5, 100, 5, las donaciones que más se repiten son: 25 y 5 euros ya que ambas han sido realizadas por 3 donantes.

Se pide:

1. Escribir un método eficiente que, dadas las donaciones efectuadas a la plataforma en un año, devuelva una lista con las donaciones más frecuentes.

public static List<Integer> getMasFrecuentes(int [] donaciones)
//Produce: las donaciones que más repiten

2. Escribir un método que ordene la lista de donaciones más frecuentes de menor a mayor. Un método de ordenación muy simple sería el método de ordenación por Selección eligiendo el elemento mayor en cada momento. Dada una lista de elementos $x_1, x_2, x_3, \ldots, x_n$ la ordenación en orden ascendente sería del siguiente modo:

Paso 1: Localizar el elemento mayor del array x_1 a x_n; intercambiarlo con x_n.
Paso 2: Localizar el elemento mayor del array x_1 a x_{n-1}; intercambiarlo con x_{n-1}.
Paso 3: Localizar el elemento mayor del array x_1 a x_{n-2}; intercambiarlo con x_{n-2}.
….
Último paso: los dos primeros elementos se comparan e intercambian si es necesario y la ordenación termina.

public static void getMasFrecuentes(List<Integer> donacionesMasFrec)
//Produce: ordena la lista de donaciones más frecuentes de menor a mayor

```
import java.util.LinkedList;
import java.util.List;

public class EjerciciosExamenB {

private static final int MAX_CANT = 501;

public static List<Integer> getMasFrecuentes (int[] donaciones){

        int[] contador = new int[MAX_CANT];

        for (int i : donaciones)
```

```
            contador[i]++;

        int numMaxDonaciones = contador[1];
        List<Integer> salida = new LinkedList<>();

        for (int donacion = 2; donacion < contador.length; donacion++){
            if (contador[donacion] > numMaxDonaciones){
                numMaxDonaciones = contador[donacion];
                salida = new LinkedList<>();
                salida.add(donacion);
            }
            else if (contador[donacion] == numMaxDonaciones)
                salida.add(donacion);
        }

      return salida;
   }



public static void OrdSeleccion(List<Integer> donacionesMasFrec) {

  for (int i = 0; i < donacionesMasFrec.size() - 1; i++) {

        int posMayor = i;

        for (int j = 0; j < donacionesMasFrec.size() - i; j++) {

            if (donacionesMasFrec.get(j) > donacionesMasFrec.get(posMayor)) {

                posMayor = j;

            }

        }

        int ultimaPos = donacionesMasFrec.size()  - i -1;


        if (posMayor != ultimaPos) {

            Integer temp = donacionesMasFrec.set(ultimaPos,
donacionesMasFrec.get(posMayor));

            donacionesMasFrec.set(posMayor, temp);

        }

      }

   }

}
```

Pregunta 2. Se quiere implementar un método que ordene los elementos de un array de Cartas. La ordenación es ascendente por el palo y el número.

La clase Carta se define como:

```
public class Carta {
   private int numero;
   private String palo;

   public Carta(int numero, String  palo){
      this.numero = numero;
      this.palo = palo;
   }
   public int getNumero(){
      return this.numero;
   }
   public String getPalo(){
      return this.palo;
   }

   public boolean soyMayor(Carta c){

      ------------------------------

   }

}
```

Para ello se pide:

a) Implementa un método en Carta que indique si el objeto es mayor que otra carta.

public boolean soyMayor (Carta c)
// Produce: si this es mayor que c devuelve true, false en caso contrario. Se considera que una carta es mayor que otra si su palo es mayor, o si es del mismo palo si su número es mayor.

b) Haciendo uso de los métodos de Carta, implementa el método ordenar de un array de cartas. Una implementación sencilla consiste en utilizar el método burbuja, el cual en cada pasada va colocando el elemento mayor en su posición correcta, es decir, en la primera pasada el elemento mayor quedará en la última posición del array, para ello, cada elemento del array se va comparando con su adyacente, de tal forma que si es mayor que su adyacente se intercambian. Se van comparando todos los elementos del array hasta llegar al penúltimo, en ese momento el elemento mayor estará colocado en la última posición. Es necesario realizar esta acción, tantas veces como elementos queremos ordenar. Implementa dicho método.

public void ordenar (Carta [] mano)
// Modifica: mano
// Produce: ordena las cartas del array ascendetemente por palo y número.

```
public boolean soyMayor(Carta c){

      if (palo.compareTo(c.palo)>0) return true;

      else if (palo.compareTo(c.palo) == 0 && numero > c.numero) return true;
```

```
    else return false;
}

public void ordenar (Carta [] mano){
    for (int pasada = 0; pasada < mano.length-1; pasada++){
        for (int j=0; j<mano.length - pasada-1; j++){
            if (mano.get(j).soyMayor(mano.get(j+1))) {
                Carta c = mano [j+1];
                mano[j+1] = mano[j];
                mano [j] = c;
            }
        }
    }
}
```

Pregunta 3. Una matriz dispersa es una matriz de gran tamaño en la que la mayor parte de sus elementos son cero. Para su implementación se va a utilizar un array cuyas posiciones se corresponden con las filas de la matriz, y donde cada posición almacena una lista que contiene los pares número de columna y el valor distinto de cero almacenado en la matriz (**los pares puede que no estén ordenados por columna**). Consideramos que tanto filas como columnas comienzan en el valor 0.

Por ejemplo, dada la matriz:

$$\begin{bmatrix} 0 & -3 & 4 \\ 4 & -3 & 0 \\ -3 & 0 & 4 \end{bmatrix}$$

Se guardaría como:

0 → {(2,4), (1,-3)}

1 → {(0,4), (1,-3)}

2 → {(2,4), (0,-3)}

Por lo tanto, una representación adecuada de una matriz dispersa consiste en un array estático en el que para cada posición (fila) se almacena una lista de objetos ValueCol (List<ValueCol> []). La clase ValueCol se define como:

```
public class ValueCol {
  private final int column;
  private int value;

public ValueCol(int column, int value){
    this.column = column;
    this.value = value;
  }
public int getColumn() {
    return this.column;
}
public int getValue() {
    return this.value;
}
public void setValue(int value) {
        this.value = value;
}
}
```

Se desea implementar un método que realice la operación suma de dos matrices dispersas **cuadradas** de igual tamaño.

```
public static List<ValueCol>[] sum(List<ValueCol>[] matrix1, List<ValueCol>[] matrix2);
```

// *Modifica*: matrix1, matrix2

// *Produce*: devuelve una matriz dispera donde cada posición del array se corresponde con la fila y contiene una lista con objetos ValueCol (columna y valor) que representan la suma de los elementos de matrix1 y matrix2.

El algoritmo para implementar el método sum, se detalla a continuación:

1. Ordenar las dos matrices. Para cada fila de la matriz, ordenar los elementos de menor a mayor atendiendo a la columna.
    a. Implementa el método de ordenación por Selección que se detalla a continuación:

Este método se basa en que cada vez que se mueve un elemento, se coloca en su posición correcta. Se comienza examinando todos los elementos, se localiza el que tiene el valor de columna más pequeño y se sitúa en la primera posición de la lista. A continuación, se localiza el menor de los restantes y se sitúa en la segunda posición. Se procede de manera similar sucesivamente hasta que quedan dos elementos. Entonces se localiza el menor y se sitúa en la penúltima posición y el último elemento, que será el mayor de todos, ya queda automáticamente colocado en su posición correcta.

Para *i* desde la primera posición hasta la penúltima
    localizar menor desde *i* hasta el final
    se elimina el elemento menor según su columna
    se inserta el elemento en la posición *i*

```
private static void selectionSort(List<ValueCol> list) {


}
```

b. Haciendo uso del método anterior, implementa el siguiente método que ordena todas las listas contenidas en un array.

```
private static void allSort(List<ValueCol>[] aux) {


}
```

2. Crea una matriz resultado, que inicialmente estará vacía.

3. Recorre de forma consecutiva todas las listas de cada una de las matrices. Para cada lista de cada una de las matrices accede a cada uno de los elementos. Para cada columna realiza la suma de sus valores y almacena el dato en la matriz resultado.

Por ejemplo, (matrix1 + matrix2 = matrixResult):

Siendo:

matrix1:

0 → {(2,4), (0,-3)}
1 → {(0,-4), (1,-3)}
2 → {(1,5), (0,3)}

matrix2:

0 → {(2,4), (1,3)}
1 → {(0,4), (1,-3)}
2 → {(2,-4), (0,-3)}

1. Ordenar matrices

matrix1:

0 → {(0,-3), (2,4)}
1 → {(0,-4), (1,-3)}
2 → {(0,3), (1,5)}

matrix2:

0 → {(1,3), (2,4)}
1 → {(0,4), (1,-3)}
2 → {(0,-3), (2,-4)}

npus de Ourense
32004 Ourense
España
el. 988 387 000
Fax 988 387 001
esei.uvigo.es
sdeuig@uvigo.es

2. **Crear matriz resultado**

   0 → {}

   1 → {}

   2 → {}

3. **Realizar suma de las matrices**

matrixResult:

   0 → (0,-3),(1,3),(2,8)

   1 → (1,-6)

   2 → (1,5),(2,-4)

Por lo tanto, implementa los tres métodos indicados:

Método 1.
```
private static void selectionSort(List<ValueCol> list)
```

Método 2.
```
private static void allSort(List<ValueCol>[] aux)
```

Método 3.
```
public static List<ValueCol>[] sum(List<ValueCol>[] matrix1,
List<ValueCol>[] matrix2)
```
.
```
public class MatrixSum {


   private static void selectionSort(List<ValueCol> list){
      for (int j = 0; j < list.size() - 1; j++) {
            int minorPosition = j;
            ValueCol valueColMinor = list.get(j);


            //buscar la columna menor
            for (int k = j + 1; k < list.size(); k++) {
               if (list.get(k).getColumn() < valueColMinor.getColumn()) {
                  minorPosition = k;
                  valueColMinor = list.get(k);
```

```
            }
        }

        if (minorPosition != j) {
            list.removeValue(valueColMinor);
            list.add(j, valueColMinor);
        }
    }
}

private static void allSort(List<ValueCol>[] aux) {
    //ordeno todas las listas seleccionando el menor en cada momento
    for (int i = 0; i < aux.length; i++) {
        selectionSort(aux[i]);
    }
}

//matrices cuadradas
public static List<ValueCol>[] sum(List<ValueCol>[] matrix1, List<ValueCol>[] matrix2){
    allSort(matrix1);
    allSort(matrix2);

    List<ValueCol> [] result = new List[matrix1.length];

    //Añadimos la matriz 1 y luego modificamos
    for (int i= 0; i<result.length; i++){
        result[i] = new LinkedList<>();
    }

    for (int i=0; i<result.length; i++){
        List<ValueCol> list1 = matrix1[i];
```

```
List<ValueCol> list2 = matrix2[i];
int index1 = 0;
int index2 = 0;

while (index1 < list1.size() && index2 < list2.size()){
    if (list1.get(index1).getColumn() < list2.get(index2).getColumn()){
        result[i].addLast(list1.get(index1));
        index1++;
    }else if (list1.get(index1).getColumn() > list2.get(index2).getColumn()){
        result[i].addLast(list2.get(index2));
        index2++;
    }
    else {//son iguales
        int sum = list1.get(index1).getValue()+list2.get(index2).getValue();
        if (sum != 0)
            result[i].addLast(new ValueCol(list1.get(index1).getColumn(),sum));
        index1++;
        index2++;
    }
}

while (index1 < list1.size()){
    result[i].addLast(list1.get(index1));
    index1++;
}

 while (index2 < list2.size()){
    result[i].addLast(list2.get(index2));
    index2++;
}
```

```
        }
        return result;
    }
}
```

Pregunta 4. **Bucket Sort** es un algoritmo de ordenación que distribuye una colección de elementos en varios **grupos** (cubetas o "buckets"), luego **ordena cada grupo individualmente**, quedando todos los elementos ordenados de menor a mayor, recorriendo en orden cada cubeta. Se basa en la idea de *distribuir uniformemente los datos*, por eso es más eficiente cuando los datos están *uniformemente distribuidos* en un rango conocido (por ejemplo, números entre 0 y 1).

Se desea utilizar este algoritmo para calcular la mediana de la combinación de dos colecciones de números **float** con valores entre 0 y 1. La cantidad de elementos de las dos colecciones es el mismo (es decir, son de igual tamaño). Implementa el método **findMedium**, dadas dos colecciones de números representados como un array de números *float* que no están ordenadas.

```
public static float findMedium(float[] collection1, float[] collection2);
```

El algoritmo para implementar el método findMedium, se detalla a continuación:

1. Ordenar las dos colecciones de elementos. El algoritmo de ordenación que se va a utilizar es el **Bucket Sort**. Se indica a continuación los pasos a seguir:
   a. Implementa el método de ordenación por Selección que se detalla a continuación:

   Este método se basa en que cada vez que se mueve un elemento, se coloca en su posición correcta. Se comienza examinando todos los elementos, se localiza el más pequeño y se sitúa en la primera posición de la lista. A continuación, se localiza el menor de los restantes y se sitúa en la segunda posición. Se procede de manera similar sucesivamente hasta que quedan dos elementos. Entonces se localiza el menor y se sitúa en la penúltima posición y el último elemento, que será el mayor de todos, ya queda automáticamente colocado en su posición correcta.

   Para *i* desde la primera posición hasta la penúltima
   localizar menor desde *i* hasta el final
   se elimina el elemento menor
   se inserta el elemento en la posición *i*

   ```
   private static void selectionSort(List<Float> list) {

   }
   ```
   b. Haciendo uso del método anterior, implementa el método bucketSort, tal y como se describe a continuación:

   - Crea tantas cubetas (bucket) como números a ordenar, se representarán como un array de listas de números Float. Inicialmente estarán vacías.

   - Distribuye todos los elementos en las cubetas, para saber en qué cubeta debes añadir el elemento (asumiendo que los números están entre 0 y 1) puedes utilizar la siguiente fórmula:
     ```
     int index = (int) (num * numElementos);
     ```

     siendo *num* el elemento a almacenar, y *numElementos* el número de elementos a ordenar.

- Haciendo uso del método anterior, ordena cada cubeta.
- Devuelve el array de listas ordenadas.

```
private static List<Float>[] bucketSort(float [] elemCollection) {

}
```

2. Una vez aplicado el método **bucketSort** a las dos colecciones, recorre de forma consecutiva todas las listas ordenadas de cada uno de los arrays, mezclando sus elementos para poder calcular la mediana. La mediana es el valor medio cuando un conjunto de datos se ordena de menor a mayor.

Por ejemplo, calcular la mediana de la combinación de collection1 y collection2:

collection1: {0.47421017, 0.1362304, 0.50045824}

collection2: {0.5516443, 0.3896347, 0.6084934}

1. **Ordenar utilizando el método bucketSort a los elementos de cada colección (ejecutar dos veces)**

   a. **Crear las cubetas**

   collection1Buckets:

   0 → {}

   1 → {}

   2 → {}

   collection2Buckets:

   0 → {}

   1 → {}

   2 → {}

   b. **Distribuir los elementos en las cubetas**

   collection1Buckets:

   0 → {0.1362304}

   1 → {0.47421017, 0.50045824}

   2 → {}

   collection2Buckets:

   0 → {}

   1 → {0.5516443, 0.3896347, 0.6084934}

   2 → {}

   c. **Ordenar cada lista**

   collection1Buckets:

   0 → {0.1362304}

   1 → {0.47421017, 0.50045824}

   2 → {}

   collection2Buckets:

   0 → {}

   1 → {0.3896347, 0.5516443, 0.6084934}

   2 → {}

   d. **Devolver el array de listas ordenadas (en una llamada devuelve collection1Buckets, y en la otra collection2Buckets)**

2. **Calcular la mediana, de los dos arrays de listas ordenadas**

Mediana: 0.4873342

Por lo tanto, implementa los tres métodos indicados:

Método 1.
```
private static void selectionSort(List<Float> list)
```

Método 2.
```
private static List<Float>[] bucketSort(float [] elemCollection)
```

Método 3.
```
public static float findMedium(float[] collection1, float[] collection2)
```

```
public class Medium {
    private static void selectionSort(List<Float> list){
        for (int j = 0; j < list.size() - 1; j++) {
            int minorPosition = j;
            Float valueMinor = list.get(j);

            //buscar la columna menor
            for (int k = j + 1; k < list.size(); k++) {
                if (list.get(k).compareTo(valueMinor) < 0) {
                    minorPosition = k;
                    valueMinor = list.get(k);
                }
            }

            if (minorPosition != j) {
                list.removeValue(valueMinor);
                list.add(j, valueMinor);
            }
        }
    }
```

```
public static List<Float>[] bucketSort(float[] arr) {
    // 1. Crear un array de listas (cubetas)
    List<Float>[] buckets = new LinkedList[arr.length];
    for (int i = 0; i < arr.length; i++) {
        buckets[i] = new LinkedList<>();
    }
    // 2. Distribuir los elementos en cubetas
    for (float num : arr) {
        int index = (int)(num * arr.length);  // Asumiendo que num está entre 0 y 1
        buckets[index].addLast(num);
    }
    // 3. Ordenar cada cubeta individualmente
    for (int i = 0; i < arr.length; i++) {
        selectionSort(buckets[i]);
    }
    return buckets;  // Devolver array de listas ordenadas
}

public static float findMedium(float[] collection1, float[] collection2) {
    List<Float>[] arr1 = bucketSort(collection1);
    List<Float>[] arr2 = bucketSort(collection2);
    int i = 0, k = 0;
    int tam = collection1.length+1;
    float[] merged = new float[tam];

    while (k < tam) {
        List<Float> list1 = arr1[i];
        List<Float> list2 = arr2[i];

        int index1 = 0;
        int index2 = 0;
```

```
        // Fusionar los dos arrays en O(n)
        while (k < tam && index1 < list1.size() && index2 < list2.size()) {
            if (list1.get(index1) < list2.get(index2)) {
                merged[k++] = list1.get(index1);
                index1++;
            } else {
                merged[k++] = list2.get(index2);
                index2++;
            }
        }

        // Agregar elementos restantes del primer array (si los hay)
        while (k < tam && index1 < list1.size()) {
            merged[k++] = list1.get(index1);
            index1++;
        }

        // Agregar elementos restantes del segundo array (si los hay)
        while (k < tam && index2 < list2.size()) {
            merged[k++] = list2.get(index2);
            index2++;
        }
        i++;
    }

    // Calcular la mediana, el tamaño de la suma de los dos arrays siempre será par
    return (merged[tam - 2] + merged[tam - 1]) / 2; // Promedio de los dos centrales
  }
}
```

Pregunta 5. **Bucket Sort** es un algoritmo de ordenación que distribuye una colección de elementos en varios **grupos** (cubetas o "buckets"), luego **ordena cada grupo individualmente**, quedando todos los elementos ordenados de mayor a menor, recorriendo en orden cada cubeta los elementos están ordenados. Se basa en la idea de *distribuir uniformemente los datos*, por eso es más eficiente cuando los datos están *uniformemente distribuidos* en un rango conocido (por ejemplo, números entre 0 y 1). Implementa los dos métodos que se indican a continuación:

a. Implementa el método de ordenación por Selección que se detalla a continuación:

   Este método se basa en que cada vez que se mueve un elemento, se coloca en su posición correcta. Se comienza examinando todos los elementos, se localiza el mayor y se sitúa en la primera posición de la lista. A continuación, se localiza el mayor de los restantes y se sitúa en la segunda posición. Se procede de manera similar sucesivamente hasta que quedan dos elementos. Entonces se localiza el mayor y se sitúa en la penúltima posición y el último elemento, que será el menor de todos, ya queda automáticamente colocado en su posición correcta.

   Para *i* desde la primera posición hasta la penúltima
   localizar menor desde *i* hasta el final
   se elimina el elemento mayor
   se inserta el elemento en la posición *i*

```
private static void selectionSortMax(List<Float> list) {
                    }
```

b. Haciendo uso del método anterior, implementa el método **bucketSort**. A continuación se detalla cómo es el algoritmo:
   - Crea tantas cubetas (bucket) como números a ordenar, se representarán como un array de listas de números Float. Inicialmente estarán vacías.
   - Distribuye todos los elementos en las cubetas, para saber en qué cubeta debes añadir el elemento (asumiendo que los números están entre 0 y 1) puedes utilizar la siguiente fórmula:

     ```
     int index = (int) (numElementos - (num * numElementos));
     ```

     siendo *num* el elemento a almacenar, y *numElementos* el número de elementos a ordenar.
   - Haciendo uso del método anterior, ordena cada lista del array.
   - Recorre las listas en orden y almacena los elementos en el array a ordenar, almacenándolos en su posición correcta.

```
private static void bucketSortMax(float [] elemCollection) {

    }
```

Por ejemplo, se desea ordenar los elementos del array collection1, bucketSort realizaría lo siguiente:

collection1: {0.8311797, 0.74408805, 0.11596672, 0.8684007, 0.2236345, 0.7979076}

a. Crear las cubetas

collection1Buckets:

0 → {}

1 → {}
2 → {}

3 → {}

4 → {}
5 → {}

b. Distribuir los elementos en las cubetas

collection1Buckets:

0 → {0.8684007}

1 → {0.8311797, 0.74408805, 0.7979076}

2 → {}

3 → {}

4 → {0.2236345}
5 → {0.11596672}

c. Ordenar cada lista, utilizando el método selectionSort.

collection1Buckets:

0 → {0.8684007}

1 → {0.8311797, 0.7979076, 0.74408805}

2 → {}

3 → {}

4 → {0.2236345}

5 → {0.11596672}

d. Recorrer las listas en desde la primera a la última y almacenar los elementos en el array, de tal forma que quedan ordenados de mayor a menor.

collection1: {0.8684007, 0.8311797, 0.7979076, 0.74408805, 0.2236345, 0.11596672}

```
public class bucketSortMax {


   private static void selectionSortMax(List<Double> list){
      for (int j = 0; j < list.size() - 1; j++) {
            int maxPosition = j;
            double valueMax = list.get(j);


            //buscar el elemento mayor
```

```
            for (int k = j + 1; k < list.size(); k++) {
                if (list.get(k).compareTo(valueMax) > 0) {
                    maxPosition = k;
                    valueMax = list.get(k);
                }
            }

            if (maxPosition != j) {
                list.removeValue(valueMax);
                list.add(j, valueMax);
            }
        }
    }
public static void bucketSortMax(double[] arr) {
        // 1. Crear un array de listas (cubetas)
        List<Double>[] buckets = new LinkedList[arr.length];
        for (int i = 0; i < arr.length; i++) {
            buckets[i] = new LinkedList<>();
        }

        // 2. Distribuir los elementos en cubetas
        for (double num : arr) {
            int index = (int)(arr.length - (num * arr.length));  // Asumiendo que num está entre 0 y 1
            buckets[index].addLast(num);
        }

        // 3. Ordenar cada cubeta individualmente
        for (int i = 0; i < arr.length; i++) {
            selectionSortMax(buckets[i]);
        }
```

```
// ordenar el array
        int i = 0;
        for (List<Double> l: buckets){
            for (Double elem: l)
                arr[i++] = elem;
        }
    }
}
```