INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

ANÁLISIS DE ALGORITMOS

PRÁCTICA 2: MERGE SORT - DIVIDE Y VENCERÁS

INTEGRANTES

- HERNÁNDEZ PINEDA MIGUEL ANGEL
- SALDAÑA AGUILAR ANDRÉS ARNULFO

PROF. CONSUELO VARINIA GARCÍA MENDOZA

19 DE OCTUBRE DE 2017

Índice general

1. Merge Sort: Divide y Vencerás **2**
1.1. Planteamiento del Problema 2
1.2. Objetivos 2
1.3. Metodología 2
1.4. Materiales 3
1.5. Desarrollo 4
1.6. Resultados 8
1.6.1. Determinación de la complejidad Recursiva 9
1.6.2. Complejidad para división entre 2 9
1.6.3. Complejidad para división entre 3 10
1.6.4. Complejidad para división entre 4 11

Conclusiones **11**

Anexo: Código **12**

Referencias Bibliográficas **16**

Capítulo 1

Merge Sort: Divide y Vencerás

1.1. Planteamiento del Problema

Existen diversos algoritmos de ordenamiento, sin embargo, la mayoría de estos tienen una complejidad computacional con un valor de n^2 por lo que al trabajar con grandes cantidades de datos, el tiempo de ejecución del algoritmo es muy grande. Uno de los algoritmos cuya complejidad es menor, se conoce con el nombre de Merge Sort o Algoritmo de Mezcla, el cual, trabaja bajo la técnica Divide y Vencerás y tiene complejidad determinada por $nlog(n)$, aún así, el algoritmo tiene un tiempo de ejecución elevado debido a que las divisiones que emplea se hacen por 2, por lo que para conjuntos de datos grandes el tiempo de ejecución sigue siendo alto.

1.2. Objetivos

- Reducir los tiempos de ejecución del algoritmo de ordenamiento conocido como Merge Sort.
- Optimizar el Merge Sort sustituyendo las divisiones entre 2, por dividendos de mayor tamaño.

1.3. Metodología

El algoritmo de Mezcla utiliza la técnica Divide y Vencerás en su implementación, pues divide a la mitad el problema inicial y aplica este procedimiento de manera recursiva de tal manera que el problema se reduzca hasta donde ya sea imposible hacer mas divisiones; sin embargo, cuando se manejan conjuntos de datos grandes, se vuelve poco factible ya que el proceso de división se llevará a cabo varias veces, por ese motivo, el algoritmo desarrollado tiene la capacidad de dividir entre un número de veces determinado por el usuario.

Procedimiento

1. El usuario ingresa el tamaño del arreglo.
2. El usuario ingresa el número de divisiones (n).
3. Se genera un arreglo del tamaño establecido con valores aleatorios.
4. La función divide y vencerás recibe el arreglo.
5. La función verifica si se cumple o no la condición establecida: *"La dimensión del arreglo es menor a la división n"* (caso base).
6. En caso de que no se cumpla, se lleva a cabo la división del arreglo en n subarreglos y llama a la función Divide y Vencerás enviando estos subarreglos (llamada recursiva) y regresamos al punto 5.
7. Si se cumple, se lleva a cabo el ordenamiento del contenido del arreglo enviado.
8. Una vez que todos los subarreglos fueron ordenados, se envían a una nueva función para combinarlos.
9. La función retorna el arreglo ordenado al método principal y se le muestra el resultado al usuario.

1.4. Materiales

Características del Ordenador

- HP Pavilion 10
- Procesador AMD A10-4655M APU whith Radeon HD Graphics 2GHz
- 12 GB de memoria RAM
- Sistema Operativo de 64 bits
- Windows 10 Home Single Language

Programación

- Lenguaje Java
- NetBeans IDE 8.2

Arreglo de Prueba

24 37 65 83 14 29 96 33 42 28 48 86 41 32 2 57 90 59 9 23 34 24 37 17 2 6 64 55 72 52 62 8 85 68 43 47 73 45 17 81 65 86 31 53 39 58 79 33 39 18 9 70 89 85 51 10 31 68 26 48 48 40 62 17 77 67 33 31 19 76 20 31 78

1.5. Desarrollo

Merge Sort: Es un algoritmo de ordenamiento externo estable basado en la técnica divide y vencerás. Es de complejidad O(n log n) que fue desarrollado en 1945 por John Von Neumann.[1]

Divide y Vencerás: En las ciencias de la computación, el término divide y vencerás (DYV) hace referencia a uno de los más importantes paradigmas de diseño algorítmico. El método está basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original. [2]

La profesora planteo el problema del ordenamiento de un vector haciendo uso de un algoritmo de ordenamiento, posteriormente, la profesora planteó el uso del Merge Sort, mostrándonos por medio del árbol de recursividad, el funcionamiento del mismo.

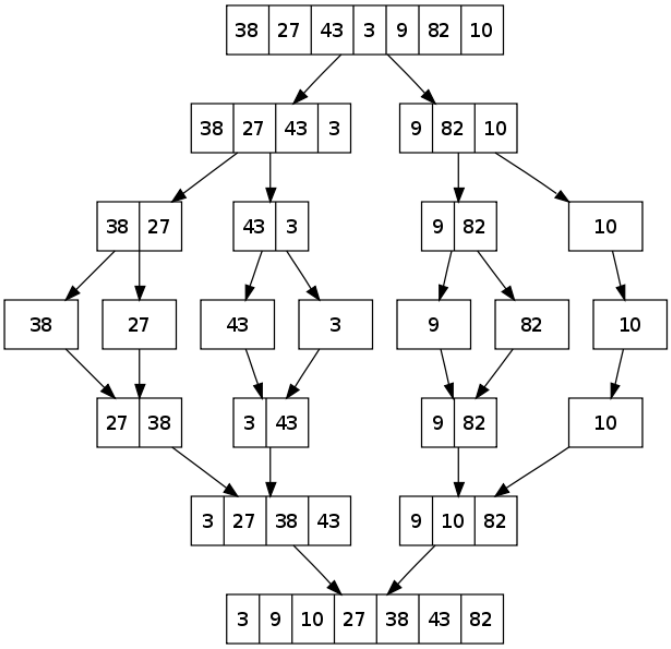

Figura 1.1: Árbol de Recursividad

Se planteó el desarrollo del algoritmo Merge Sort pero sustituyendo la división entre 2, por cualquier otro divisor n elegido por el usuario. Para poder llevar a cabo esto, fue necesario hacer un análisis del pseudo-código del merge sort con divisiones entre 2.

```
1:  if length(m) <= 1 then
2:      return result
3:  else
4:
5:      for each x in m up to middle - 1 do
6:          add x to left
7:      end for
8:      for each x in m at and after middle do
9:          add x to right
10:     end for
11:     left = mergesort(left)
12:     right = mergesort(right)
13:     if last(left) <= first(right) then
14:         append right to left
15:         return left
16:     end if
17:     result = merge(left, right)
18:     return result
19: end if
```

Function: MergeSort [3]

```
1:  while length(left) > 0 and length(right) > 0 do
2:
3:      if first(left) <= first(right) then
4:          append first(left) to result
5:          left = rest(left)
6:      else
7:          append first(right) to result
8:          right = rest(right)
9:      end if
10:     end while
11:     if length(left) > 0  then
12:         append rest(left) to result
13:     end if
14:     if length(right) > 0  then
15:         append rest(right) to result
16:     end if
17: return result
```

Function: Merge [3]

Se analizó y se decidió trabajar en lenguaje Java ya que nos pareció más cómoda la implementación del algoritmo en este lenguaje. Para inciar, se creó la clase **MergeSort** ,la cuál tiene dos atributos, el tamaño del arreglo y la división (**tam** y **div** respectivamente), además tiene un constructor que recibe los dos atributos como parámetros y cuenta con los métodos **creaArreglo**, **ordenaArreglo** y **combinar**.

```
public class MergeSort{

int tam,div;

public MergeSort(int tam, int div){...4 lines }


public void creaArreglo(){...11 lines }

public int[] ordenaArreglo(int[] arreglo){...111 lines }

public int[] combinar(int [][] a, int [] b, int x, int y){...23 lines }
```

Figura 1.2: Clase MergeSort

El programa crea un nuevo objeto de la clase MergeSort y le envía como parámetros los valores ingresados por el usuario, posteriormente se llama al método creaArreglo y genera un arreglo de la longitud ingresada por el usuario.

```
public void creaArreglo(){
        int[] arreglo=new int[tam];
        for(int i=0; i<tam; i++){
                Random rand=new Random();
                arreglo[i]=rand.nextInt(100)+1;
        }
        ordenaArreglo(arreglo);
}
```

Figura 1.3: Método creaArreglo

Como podemos ver, una vez generado el arreglo, se llama al método ordenaArreglo, aquí se encuentra la implementación del algoritmo Divide y Vencerás, es decir, donde se llevan a cabo todos los procesos del caso base, así como los procesos necesarios para la creación de los subarreglos y las llamadas recursivas.
Uno de los problemas a los que nos enfrentamos, fue que al dividir entre n números, en algunas ocasiones, los subarreglos que se generaban no eran iguales en longitud, es decir, pueden quedar $n-1$ arreglos de una longitud y el último arreglo tener una dimensión mayor en una o más unidades. Debido a esto, se optó por guardar los $n-1$ subarreglos generados en una matriz, y el último subarreglo guardarlo en un vector, posterior a esto, se llevan a cabo las llamadas recursivas, primero para los datos guardados en la matriz, y finalmente para el vector.

Si nos dirigimos al caso base, la condición que debe cumplirse es que el tamaño del vector sea menor al número de divisiones que se deben hacer, aquí se lleva a cabo el ordenamiento de los valores del vector más pequeño que salió de la recursividad. A continuación se muestra la implementación de nuestro caso base y de las llamadas recursivas.

```
if(tam < div){
        for(int i=0;i<(tam-1);i++){
            for(int j=i+1;j<tam;j++){
                if(arreglo[i]>arreglo[j]){
                    //Intercambiamos valores
                    int variableauxiliar=arreglo[i];
                    arreglo[i]=arreglo[j];
                    arreglo[j]=variableauxiliar;
                }
            }
        }
        result=arreglo;
}
```

Figura 1.4: ordenaArreglo: Caso Base

```
for (int i = 0; i < (div-1); i++) {
    for (int k = 0; k < nvotam; k++) {
        arreglo_aux[k]=nvoarreglo[i][k];
    }
    int [] arreglo_aux2;
    arreglo_aux2=ordenaArreglo(arreglo_aux);
```

Figura 1.5: ordenaArreglo: Llamada Recursiva Matriz

```
int [] arregloFinal=new int[tam_aux];
arregloFinal=ordenaArreglo(ultimoarreglo);
System.out.println("");
result=combinar(arreglosFinales,arregloFinal,div-1,nvotam);
```

Figura 1.6: ordenaArreglo: Llamada Recursiva Vector

Como podemos ver, las llamadas recursivas para la matriz, en realidad se envian vectores, puesto que nuestra función ordenaArreglo solo recibe vectores, por ese motivo se crea un arreglo auxiliar donde con ayuda de un ciclo, se va guardando cada una de las filas de la matriz para poder llevar a cabo la llamada recursiva. Una vez terminada la recursión de la función ordenaArreglo se invoca el método combinar donde se llevará a cabo la combinación de los vectores ordenados para generar un nuevo vector con todos sus valores ordenados de menor a mayor.

En el siguiente extracto de código se puede apreciar el contenido del método combinar, y como podemos observar el proceso es una comparación de los valores de los vectores para que al final, queden ordenados como se desea. Esta función regresa el arreglo final ordenado para posteriormente este sea impreso en pantalla y el usuario observe que se llevó a cabo el ordenamiento de manera correcta.

```
public int[] combinar(int [][] a, int [] b, int x, int y){
        int []al= new int[x*y];
        int n=0;
        for(int i=0; i<x; i++){
                for(int j=0; j<y; j++){
                al[n]=a[i][j];
                n++;
                }
        }
        int []c = new int[al.length+b.length];
        int k;
        for(k=0; k<al.length; k++)
                c[k] = al[k];
    for(int j=0; j<b.length; j++)
        c[k++]=b[j];
    Arrays.sort(c);
    return c;
}
```

Figura 1.7: Método combinar

Se optó por mostrar en pantalla el proceso que se realiza de manera que el usuario pueda observar como se van dividiendo los arreglos y subarreglos.

```
Arreglo despues de combinar
19 20 31 76 78
Arreglo antes de combinar
26 31 68  | 40 48 48  | 17 62 77  | 31 33 67  | 19 20 31 76 78
Arreglo despues de combinar
17 19 20 26 31 31 31 33 40 48 48 62 67 68 76 77 78
Arreglo antes de combinar
14 24 28 29 32 33 37 41 42 48 65 83 86 96  | 2 2 6 9 17 23 24 34 37 55 57 59 64 90  | 8 17 43 45 47 52 62 65 68 72 73 81 85 86  |
Arreglo despues de combinar
2 2 6 8 9 9 10 14 17 17 17 18 19 20 23 24 24 26 28 29 31 31 31 31 32 33 33 33 34 37 37 39 39 40 41 42 43 45 47 48 48 48 51 52 53
```

Figura 1.8: Extracto del proceso mostrado en pantalla

1.6. Resultados

Se llevaron a cabo 10 pruebas para divisiones entre 2, 3 y 4 con un arreglo de longitud 73 con los mismos valores, esto con el fin de obtener los distintos tiempos que tarda el programa en ejecutarse. El arreglo utilizado para las pruebas fue el siguiente:

24 37 65 83 14 29 96 33 42 28 48 86 41 32 2 57 90 59 9 23 34 24 37 17 2 6 64 55 72 52 62 8 85 68 43 47 73 45 17 81 65 86 31 53 39 58 79 33 39 18 9 70 89 85 51 10 31 68 26 48 48 40 62 17 77 67 33 31 19 76 20 31 78

Los resultados obtenidos se muestran en la siguiente tabla:

Prueba	División por 2	División por 3	División por 4
1	110 ms	77 ms	87 ms
2	130 ms	62 ms	50 ms
3	194 ms	72 ms	50 ms
4	120 ms	100 ms	49 ms
5	101 ms	82 ms	49 ms
6	101 ms	50 ms	60 ms
7	110 ms	70 ms	58 ms
8	111 ms	50 ms	70 ms
9	199 ms	59 ms	68 ms
10	131 ms	74 ms	90 ms
Promedio	130.7 ms	69.6 ms	63.1 ms

Cuadro 1.1: Tabla de Resultados

Como podemos observar los tiempos de ejecución se reducen cuando aumentamos el número por el que dividiremos nuestros arreglos y subarreglos, por tanto, como era de esperarse, el llevar a cabo divisiones entre 4 reduce el tiempo de ejecución en un 51.78 %.

1.6.1. Determinación de la complejidad Recursiva

A continuación se muestra la complejidad recursiva de nuestro algoritmo para n divisiones.

$$MS(x) = \begin{array}{ll} W1 & x < n \\ n * MS(x/n) + W2 & x >= n \end{array} \tag{1.1}$$

Se buscará la complejidad no recursiva del algoritmo para 2, 3 y 4 divisiones usando el método de las no homogéneas.

1.6.2. Complejidad para división entre 2

Tomando la ecuación 1.1 sustituimos el valor n por 2, por lo que tenemos:

$$MS(x) = \begin{array}{ll} W1 & x < 2 \\ 2MS(x/2) + W2 & 2 >= n \end{array} \tag{1.2}$$

De la ecuación 1.2 tomamos el término recursivo $MS(x) = MS(x/2) + W2$ y separamos los términos homogéneos y no homogéneos de la ecuación.

$$MS(x) - 2MS(x/2) = W2 \tag{1.3}$$

Hacemos el cambio de variable $x = 2^k$ y sustituimos en 1.3

$$MS(2^k) - 2MS(2^{k-1}) = W2 \tag{1.4}$$

Ahora, sustituimos en 1.4 los términos homogéneos y no homogéneos de acuerdo a $MS(2^k) = S$ para los homogéneos y $(S-b)^{g+1}$ para los no homogéneos donde $b = 1$ y $g = 0$ (para este caso); tenemos entonces:

$$(S-2) = (S-1) \tag{1.5}$$

Multiplicamos ambos términos e igualamos a 0 1.5 y posteriormente obtenemos las raíces de la ecuación:

$$(S-2)(S-1) = 0 \tag{1.6}$$

Entonces tenemos que $s1 = 2$ y $s2 = 1$; ahora con las raíces tenemos una nueva ecuación para $MS(2^k)$

$$MS(2^k) = C1k^0(2)^k + C2k^0(1)^k \tag{1.7}$$

$$MS(2^k) = C1(2)^k + C2 \tag{1.8}$$

Ahora para volver a nuestra variable original despejamos de $x = 2^k$ el valor de k y obtenemos $k = log_2(x)$, sustituimos en 1.8

$$MS(x) = C1x + C2 \tag{1.9}$$

Tenemos las condiciones iniciales $MS(1) = W1$ y $MS(2) = 2W1 + W2$ por lo que será necesario evaluar la ecuación 1.9 para obtener los valores de C1 y C2.

$$W1 = C1 + C2 \tag{1.10}$$

$$2W1 + W2 = 2C1 + C2 \tag{1.11}$$

Resolvemos el sistema de ecuaciones y tenemos que:

- $C1 = W1 + W2$
- $C2 = -W2$

Finalmente sustituimos estos valores en la ecuación 1.9 y obtenemos la Ecuación Caracteristica de la complejidad del algoritmo:

MS(x)=(W1+W2) x - W2 (1.12)

1.6.3. Complejidad para división entre 3

Tomando la ecuación 1.1 sustituimos el valor n por 3, por lo que tenemos:

$$MS(x) = \begin{array}{ll} W1 & x < 3 \\ 3MS(x/3) + W2 & x >= 3 \end{array} \tag{1.13}$$

Ahora, repetimos el proceso hecho en el punto *1.6.2*.

$$MS(x) - 3MS(x/3) = W2 \tag{1.14}$$
$$MS(3^k) - 3MS(3^{k-1}) = W2 \tag{1.15}$$
$$(S-3) = (S-1) \tag{1.16}$$
$$(S-3)(S-1) = 0 \tag{1.17}$$
$$MS(3^k) = C1k^0(3)^k + C2k^0(1)^k \tag{1.18}$$
$$MS(3^k) = C1(3)^k + C2 \tag{1.19}$$
$$MS(x) = C1x + C2 \tag{1.20}$$
$$W1 = C1 + C2 \tag{1.21}$$
$$3W1 + W2 = 3C1 + C2 \tag{1.22}$$

- $C1 = W1 + W2$
- $C2 = -W2$

$$\textbf{MS(x)=(W1+W2) x - W2} \tag{1.23}$$

1.6.4. Complejidad para división entre 4

Tomando la ecuación 1.1 sustituimos el valor n por 4, por lo que tenemos:

$$MS(x) = \begin{matrix} W1 & x < 4 \\ 4MS(x/4) + W2 & x >= 4 \end{matrix} \tag{1.24}$$

Ahora, repetimos el proceso hecho en el punto *1.6.2*.

$$MS(x) - 4MS(x/4) = W2 \tag{1.25}$$
$$MS(4^k) - 4MS(4^{k-1}) = W2 \tag{1.26}$$
$$(S-4) = (S-1) \tag{1.27}$$
$$(S-4)(S-1) = 0 \tag{1.28}$$
$$MS(4^k) = C1k^0(4)^k + C2k^0(1)^k \tag{1.29}$$
$$MS(4^k) = C1(4)^k + C2 \tag{1.30}$$
$$MS(x) = C1x + C2 \tag{1.31}$$
$$W1 = C1 + C2 \tag{1.32}$$
$$4W1 + W2 = 4C1 + C2 \tag{1.33}$$

- $C1 = W1 + W2$
- $C2 = -W2$

$$\textbf{MS(x)=(W1+W2) x - W2} \tag{1.34}$$

Conclusiones

Los algoritmos de ordenamiento nos dan un claro ejemplo de que al momento de trabajar con cantidades de datos mayores, la cantidad de trabajo que requiere hacer el ordenador aumenta en proporciones no lineales, aún asi, existen algunas técnicas que nos permiten reducir la complejidad de los algoritmos computacionales, de tal manera que su uso se factible incluso en el manejo conjuntos de datos grandes, la técnica Divide y Vencerás es un claro ejemplo de esto. Como su nombre lo indica, está técnica divide el problema general en subproblemas que son más fáciles de resolver, y esto permite que el trabajo que requiere el ordenador disminuya. Uno de los algoritmos de ordenamiento que implementa esta técnica es el conocido como MergeSort, sin embargo, este algoritmo divide el problema a la mitad por lo que aún así, los tiempos de ejecución que tiene no son muy convenientes.
Se llevó a cabo una implementación que permite al usuario ingresar el numero de divisiones que desea que se hagan,lo que mejora de manera considerable los tiempos de ejecución del algoritmo. Para poder comprobar esto, se realizaron 10 pruebas con diferentes valores para la división que nos permitieron encontrar un tiempo de ejecución promedio para cada uno de estos valores y así identificar cuales se llevaban a cabo más rápido; en efecto, mientras el valor del divisor era mayor, los tiempos de ejecución se reducían. Además, es importante tener en cuenta que la complejidad del algoritmo no aumenta ni disminuye haciendo estos cambios, pues siempre se conserva como una ecuación lineal determinada por los valores que se le asignen a x que en realidad representa la longitud de los vectores ingresados.

Anexo: Código

```
import java.util.Random;
import java.util.Arrays;

public class MergeSort{

int tam,div;

public MergeSort(int tam, int div){
  this.tam=tam;
  this.div=div;
}


public void creaArreglo(){
    int[] arreglo=new int[tam];
    for(int i=0; i<tam; i++){
      Random rand=new Random();
      arreglo[i]=rand.nextInt(100)+1;
      System.out.print(""+arreglo[i]);
      System.out.print(" ");
    }
    System.out.println();
    ordenaArreglo(arreglo);
}

public int[] ordenaArreglo(int[] arreglo){

  int tam=arreglo.length;
  System.out.println("Tamanio: "+tam);
  int []result;
  //Caso base, ordenamos el arreglo
  if(tam < div){
    for(int i=0;i<(tam-1);i++){
        for(int j=i+1;j<tam;j++){
            if(arreglo[i]>arreglo[j]){
                //Intercambiamos valores
                int variableauxiliar=arreglo[i];
                arreglo[i]=arreglo[j];
                arreglo[j]=variableauxiliar;
            }
        }
    }
    System.out.println("Arreglo ordenado:");
    for(int i=0; i<tam; i++){
```

```
45        System.out.print(""+arreglo[i]);
46        System.out.print(" ");
47      }
48      System.out.println();
49
50      result=arreglo;
51    }
52    //caso recursivo
53    else{
54      System.out.println("Dividiendo arreglos");
55      /*crea los subarreglos*/
56      /*Los primeros o primer arreglo, que obedece a la division y funcion
           piso.*/
57        int nvotam=(int)(Math.floor(tam/div));
58        int[][] nvoarreglo=new int[div-1][nvotam]; //numero de arreglo,
             tamanio
59        int n=0, j=0;
60
61        while(n!=(div-1) && j!=tam){
62            for(int i=0; i<nvotam; i++){
63              nvoarreglo[n][i]=arreglo[j];
64              System.out.print(""+nvoarreglo[n][i]);
65              System.out.print(" ");
66              j++;
67            }
68          System.out.println();
69          n++;
70        }
71      /*puede que el ultimo arreglo sea igual o no.*/
72        int tam_aux=tam-j; //nos da el tama o del arreglo restante
73        int[] ultimoarreglo=new int[tam_aux];
74
75        for(int i=0; i < tam_aux ; i++){
76              ultimoarreglo[i]=arreglo[j];
77              System.out.print(""+ultimoarreglo[i]);
78              System.out.print(" ");
79              j++;
80          }
81          System.out.println();
82
83
84      System.out.println("Ordenando arreglos");
85      //mandamos los arreglos a recursi n
86      //orimero, los arreglos que son iguales
87      int [][] arreglosFinales= new int[div-1][nvotam];
88      int [] arreglo_aux=new int[nvotam];
89
90      int m=0;
91
92      for (int i = 0; i < (div-1); i++) {
93          for (int k = 0; k < nvotam; k++) {
94              arreglo_aux[k]=nvoarreglo[i][k];
95              System.out.print(""+arreglo_aux[k]);
96              System.out.print(" ");
```

```
97          }
98          System.out.println("");
99          int [] arreglo_aux2;
100         arreglo_aux2=ordenaArreglo(arreglo_aux);
101
102         for (int k = 0; k < nvotam; k++) {
103             arreglosFinales[i][k]=arreglo_aux2[k];
104         }
105     }
106     //ahora, el arreglo que es diferente
107     int [] arregloFinal=new int[tam_aux];
108     arregloFinal=ordenaArreglo(ultimoarreglo);
109
110     //los imprimimos ordenados por bloques antes de combinarlos
111     System.out.println("Arreglo antes de combinar");
112     for (int i = 0; i < (div-1); i++) {
113         for (int k = 0; k < nvotam; k++) {
114             System.out.print(""+arreglosFinales[i][k]);
115             System.out.print(" ");
116         }
117         System.out.print(" | ");
118     }
119
120     for (int k = 0; k < arregloFinal.length; k++) {
121             System.out.print(""+arregloFinal[k]);
122             System.out.print(" ");
123         }
124
125     System.out.println("");
126     result=combinar(arreglosFinales,arregloFinal,div-1,nvotam);
127     System.out.println("Arreglo despues de combinar");
128         for (int k = 0; k < result.length; k++) {
129             System.out.print(""+result[k]);
130             System.out.print(" ");
131         }
132     System.out.println("");
133   }
134
135   return result;
136 }
137
138 public int[] combinar(int [][] a, int [] b, int x, int y){
139
140   int []a1= new int[x*y];
141   int n=0;
142
143   for(int i=0; i<x; i++){
144     for(int j=0; j<y; j++){
145     a1[n]=a[i][j];
146     n++;
147     }
148   }
149
150   int []c = new int[a1.length+b.length];
```

```
151     int k;
152     for(k=0; k<a1.length; k++)
153         c[k] = a1[k];
154     for(int j=0; j<b.length; j++)
155         c[k++]=b[j];
156 
157     Arrays.sort(c);
158     return c;
159 
160 }
161 
162 public static void main(String[] args) {
163     MergeSort ms=new MergeSort(73,4);
164     ms.creaArreglo();
165   }
166 }
```

Referencias Bibliográficas

[1] «Ordenamiento por Mezcla» Recuperado de: https://es.wikipedia.org/wiki/Ordenamiento_por_mezcla

[2] «Algoritmo Divide y Vencerás» Recuperado de: https://es.wikipedia.org/wiki/Algoritmo_divide_y_vencer%C3%A1s

[3] «Merge Sort: Pseudocode» Recuperado de: https://es.coursera.org/learn/algorithms-divide-conquer/lecture/NtFU9/merge-sort-pseudocode