Practica 2: Algoritmos de Ordenamiento Parte 2

Adolfo Roman Jimenez

September 26, 2021

1 Objetivo

El estudiante identificará la estructura de los algoritmos de ordenamiento BubbleSort, QuickSort y MergeSort

2 Desarrollo

2.1 Ejercicio 1

2.1.1 QuickSort

Antes de comenzar a analizar el algoritmo *QuickSort* se debe de mencionar que este incluye una funcion auxiliar cuyo nombre es *partition().*

El algortimo *QuickSort* utiliza 3 argumentos, un arreglo de numeros enteros llamado **arr** y dos enteros respectivamente que son **low** y **high**.

Los enteros **low** y **high** en realidad corresponden al primero y ultimo indices del arreglo **arr** insertado, por lo que **low** siempre sera 0 y **high** correspondera al tamano del arreglo - 1 unidad.

Despues de esto, el codigo en *QuickSort* asigna una variable **pi** de tipo entero al resultado de la funcion *partition().*

QuickSort es un algoritmo que usa la estrategia *"divide y venceras"* y lo que hace en realidad, es que genera algo llamado *pivote* que funciona como referencia para ordenar el arreglo, pues en terminos generales, todo elemento que tenga un valor inferior al pivote lo posiciona en el lado izquierdo y todo elemento que contrariamente tenga un valor superior, lo posicionara en el lado derecho del pivote.

En esta version de *QuickSort* el pivote se escoje al final del arreglo, pero existen diferentes versiones del mismo algoritmo donde el pivote puede elegirse al principio, al final o en un punto al azar del arreglo.

El programa despues evalua si **low** es menor a **high**, si esto es verdadero, entonces declara una variable entera **pi** que es el punto en donde el arreglo se particionara. El valor de esta variable, es proporcionado por la funcion *paritition()* que devuelve un valor entero.

La funcion *partition()* de igual forma que *QuickSort* toma 3 argumentos que se componen de un arreglo de numeros enteros llamado **arr** y dos enteros respectivamente que de igual manera son **low** y **high**.

Despues se declara una variable entera **pivot** que se iguala al valor en ultimo indice de **arr**, en seguida se declaran las variables enteras **j** e **i**, a **i** se le asigna el valor del indice menor **low - 1** y a **j** para efectos de un *for-loop* se le asigna el valor en **low** que ira sumando en una unidad a **j** hasta alcanzar el valor en **high** que es cuando el loop se rompera.

Ya dentro del *loop*, en el caso de que el valor de **arr** en el indice **j** sea menor o igual al valor en **pivot** entonces se le sumara 1 a la variable **i** y se hace un intercambio de valores entre el indice **i** y el indice **j** en **arr**.

Al finalizar y salir del *loop* se hace un intercambio entre los valores en los indices **i + 1** y **high** en **arr** y la funcion finaliza retornando el valor en **i + 1**.

Basicamente lo que la funcion *partition()* hace, es que toma el pivote, que es el ultimo elemento en la primera iteracion y coloca todos los elementos menores que el a la izquierda y mayores a la derecha como lo muestra la siguiente imagen.

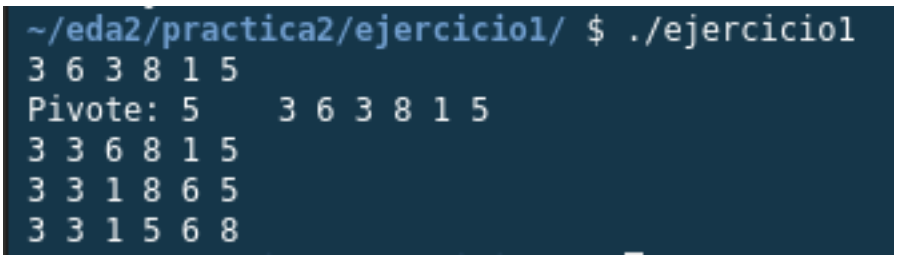

Figure 1: Funcion *partition()* ejecutada sobre arreglo [3, 6, 3, 8, 1, 5]

Una vez que la variable **pi** obtiene un valor, se utiliza la recursion obligando a *QuickSort* a invocarse de nuevo, pero esta vez lo hara modificando el valor a **high** restandole 1, lo que ocasionara que tome como pivote el penultimo digito y repita la operacion recursivamente, lo mismo con una segunda funcion recursiva pero esta vez, el valor en **low** se traslada a un indice despues de el del pivote, ejecutando la funcion recursivamente de igual forma hasta el momento en que se ordena completamente.

2.1.2 BubbleSort

El algoritmo de *BubbleSort*, como su nombre lo indica, es un algoritmo que va empujando a los numeros mas grandes al final del arreglo donde se encuentran alojados, como si se tratara de una serie de burbujas que se van inflando.

El algoritmo de *BubbleSort* es uno muy sencillo, que tiene tambien una implementacion facil, pero que lamentablemente, su tiempo de ejecucion puede llegar a ser de $O(n^2)$ en el peor de los casos.

BubbleSort comienza tomando 2 argumentos, el primero un arreglo de enteros **a** y el segundo una variable entera **size** que corresponde al tamano del arreglo.

La funcion declara las variables enteras **i**, **j** y **n**. **i** y **j** serviran para iterar en los *loops* subsecuentes mientras que a **n** se le deposita el valor de **size**.

Cuando comienza a iterar el primer *for-loop* lo hace desde el ultimo indice del arreglo hacia el primero en **i** y en **j** inicia desde el 0 hasta el valor en **i**.

En cada iteracion dentro del *for-loop* en **j** se va evaluando si el numero que se esta revisando es mayor que el siguiente. Si esto es verdadero, entonces el programa intercambia de posicion los numeros y de esta manera el numero mas grande va abriendose paso dentro del arreglo hasta quedar posicionado al final de este, mientras que con cada iteracion los numeros mas chicos, de igual manera se van alejando de la parte derecha del arreglo para llegar a la parte izquierda.

El algoritmo finaliza cuando el *for-loop* principal, ha llegado al 0 y es para este tiempo que el arreglo queda ordenado.

2.1.3 BubbleSort - Stop

Para este ejemplo, lo unico que se me ocurrio fue agregar una especie de *switch* al codigo (y no me refiero a una estructura de control tipo *switch*) en el que creo una variable entera **x** y le asigno un valor de 1.

En el momento en que el algoritmo *BubbleSort* comienza a trabajar, entonces si por lo menos en una ocasion, un numero dentro del arreglo es mayor que el siguiente numero, entonces el valor de **x** cambia a 0 y el algoritmo continua trabajando mientras.

AL finaliar el segundo *for-loop* entonces se encuentra un codigo *if* que evalua si el valor en **x** es igual o no a 1 y si es igual a 1, entonces quiere decir que el arreglo nunca se modifico, por lo que necesariamente los elementos deben de estar ordenados en forma ascendecente.

Si esto sucede entonces la estructura de control rompe el ciclo con un *break* y el programa termina.

De esta manera podeos notar que *BubbleSort* cuando el codigo se activa, en una lista ordenada, solamente recorre la lista 1 sola vez, mientras sin el codigo, recorre a lista **n - 1** veces su longitud aun cuando no existen elementos a modificarse.

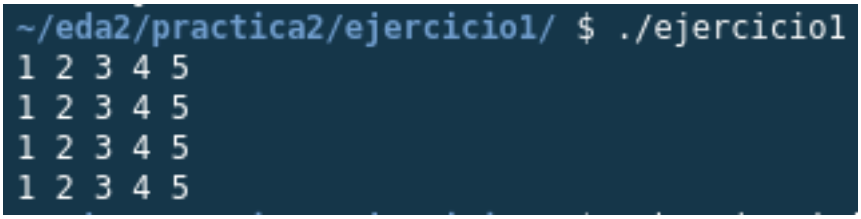

Figure 2: BubbleSort NO modificado, iterando sobre arrelo [1, 2, 3, 4, 5]

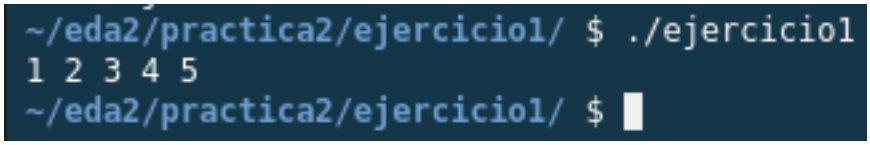

Figure 3: BubbleSort modificado, iterando sobre arrelo [1, 2, 3, 4, 5]

2.2 Ejercicio 2

Agregando la funcion *MergeSort* como estaba en el video e implementandola en *C*, no tuve muchas dificultades para hacerlo. Simplemente tuve que modificar la palabra *list* por un **int list[]** al momento de escribir los argumentos de las funciones, tanto en *merge()* como en *mergeSort()* y en el caso de *merge()* al momento de crear un segundo arreglo, lo unico que hice fue implementar **int lista2[r + 1];** que en la nueva version de *C* si es permitido para crear un arreglo con ese numero de posiciones dentro de los barckets cuadrados.

Al final hice una prueba y ordena correctamente un arreglo que corri con numeros aleatorios.

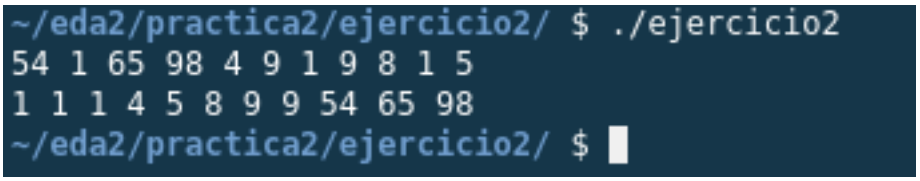

Figure 4: Arreglo ordenado con MergeSort

2.3 Ejercicio 3

2.3.1 Menu

El menu, como el ejercicio lo indica, es el mismo que el de la practica pasada, pero con los nuevos algoritmos integrados a este, de esta forma, tenemos como nuevos elementos del menu a los algoritmos:

- *Quick Sort*
- *Bubble Sort*
- *Merge Sort*

De igual forma, el programa contiene una funcion que realiza un arreglo al azar de los elementos que se deseen y el cual se muestra funcionando en la siguiente imagen:

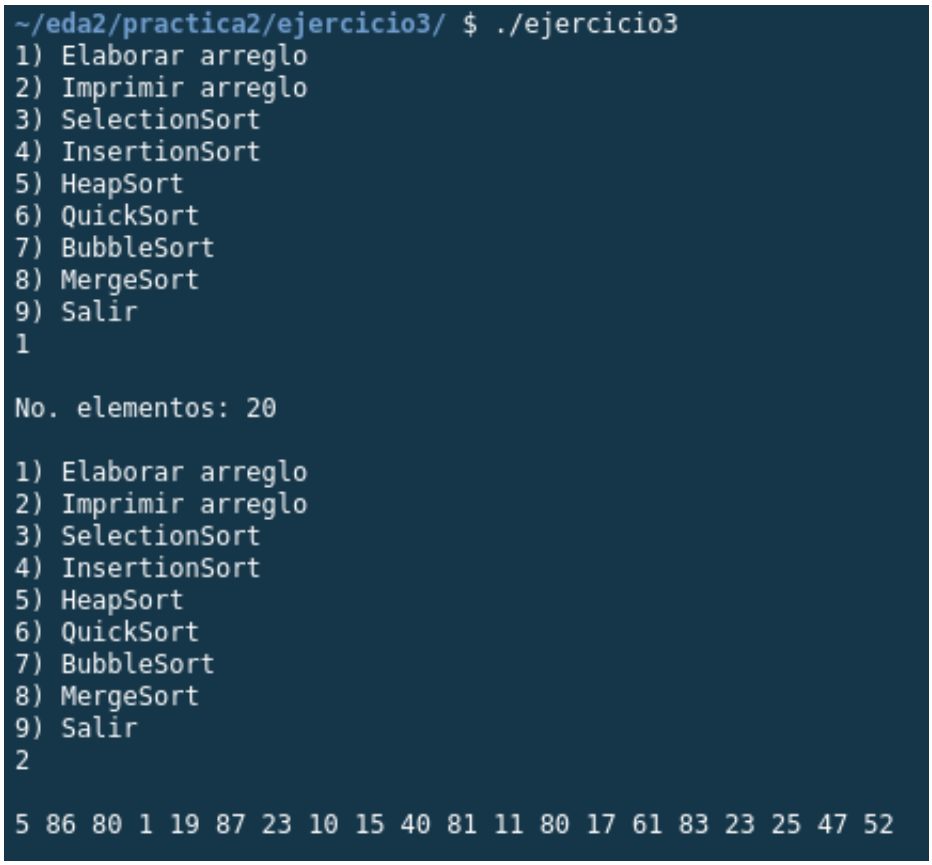

Figure 5: Menu Principal

2.3.2 Quick Sort

La opcion numero 6 corresponde a *QuickSort()* que fue transcrito directamente como venia en la practica. El programa imprime el pivote y los subarreglos que *QuickSort* va generando en cada iteracion.

Las imagenes muestran el funcionamiento del programa, primeramete se crear el arreglo al azar de 20 elementos y despues se escoje la opcion numero 6 para que corra el programa.

```
~/eda2/practica2/ejercicio3/ $ ./ejercicio3
1) Elaborar arreglo
2) Imprimir arreglo
3) SelectionSort
4) InsertionSort
5) HeapSort
6) QuickSort
7) BubbleSort
8) MergeSort
9) Salir
1

No. elementos: 20

1) Elaborar arreglo
2) Imprimir arreglo
3) SelectionSort
4) InsertionSort
5) HeapSort
6) QuickSort
7) BubbleSort
8) MergeSort
9) Salir
6

Arreglo sin ordenar: 89 92 49 8 56 25 7 61 38 59 10 91 32 86 20 83 17 3 47 90
```

Figure 6: Funcinamiento QuickSort Imagen 1

```
Arreglo sin ordenar: 89 92 49 8 56 25 7 61 38 59 10 91 32 86 20 83 17 3 47 90

Pivote: 90    Sub array :  89 49 8 56 25 7 61 38 59 10 32 86 20 83 17 3 47
Pivote: 47    Sub array :  8 25 7 38 10 32 20 17 3
Pivote: 3    Sub array :
Sub array :  25 7 38 10 32 20 17 8
Pivote: 8    Sub array :  7
Sub array :  38 10 32 20 17 25
Pivote: 25    Sub array :  10 20 17
Pivote: 17    Sub array :  10
Sub array :  20
Sub array :  32 38
Pivote: 38    Sub array :  32
Sub array :
Sub array :  89 86 61 83 56 59 49
Pivote: 49    Sub array :
Sub array :  86 61 83 56 59 89
Pivote: 89    Sub array :  86 61 83 56 59
Pivote: 59    Sub array :  56
Sub array :  83 86 61
Pivote: 61    Sub array :
Sub array :  86 83
Pivote: 83    Sub array :
Sub array :  86
Sub array :
Sub array :  92 91
Pivote: 91    Sub array :
Sub array :  92

Arreglo ordenado: 3 7 8 10 17 20 25 32 38 47 49 56 59 61 83 86 89 90 91 92
```

Figure 7: Funcinamiento QuickSort Imagen 2

2.3.3 Bubble Sort

La opcion numero 7 pertenece a *BubbleSort()*. Este programa me gusto mucho como quedo, porque despues de investigar un poco, escribi una funcion alterna *ssswap()* la cual agrega color a los elementos que estan siendo modificados en su posicion en cada iteracion.

Las imagenes muestran el funcionamiento del programa, se crea un nuevo arreglo de 20 elementos y al momento de ejecutar el algoritmo, en amarillo se muestra el elemento menor y en verde el elemento que va "arrastrandose" a traves del arreglo hasta que ya no encuentra ningun otro elemento mayor que este, entonces comienza nuevamente al principio del arreglo hasta que termina de ejecutarse.

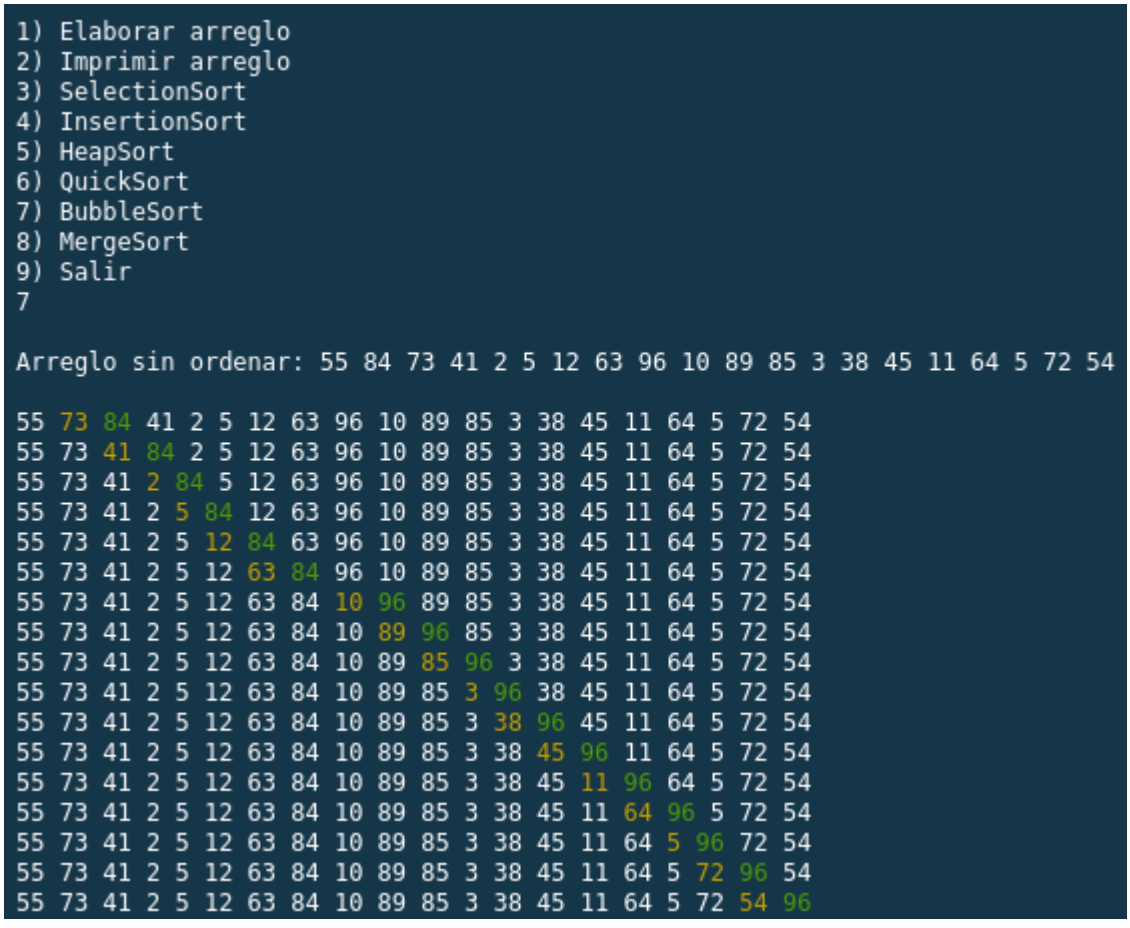

Figure 8: Algoritmo Bubble Sort Imagen 1

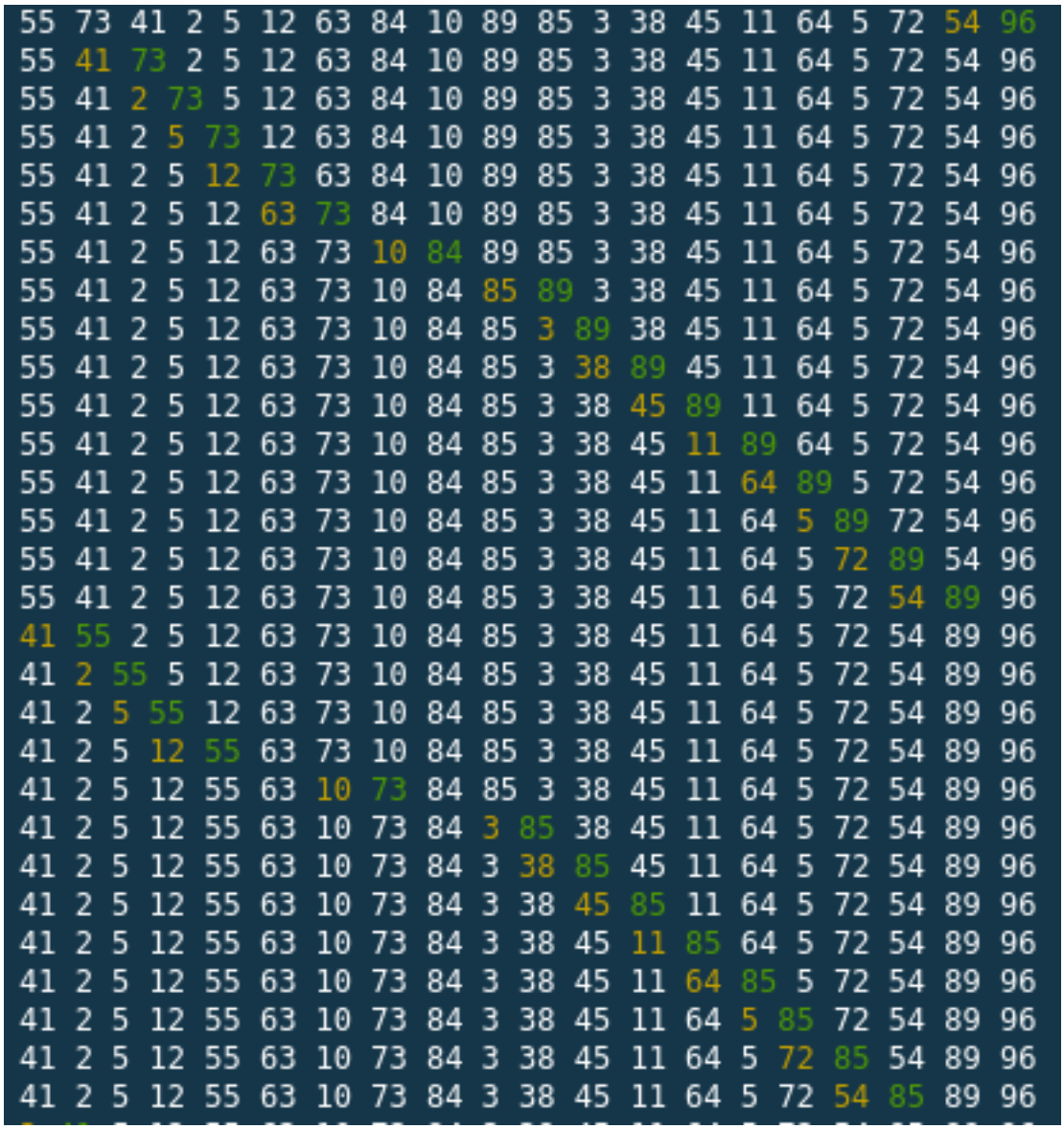

Figure 9: Algoritmo Bubble Sort Imagen 2

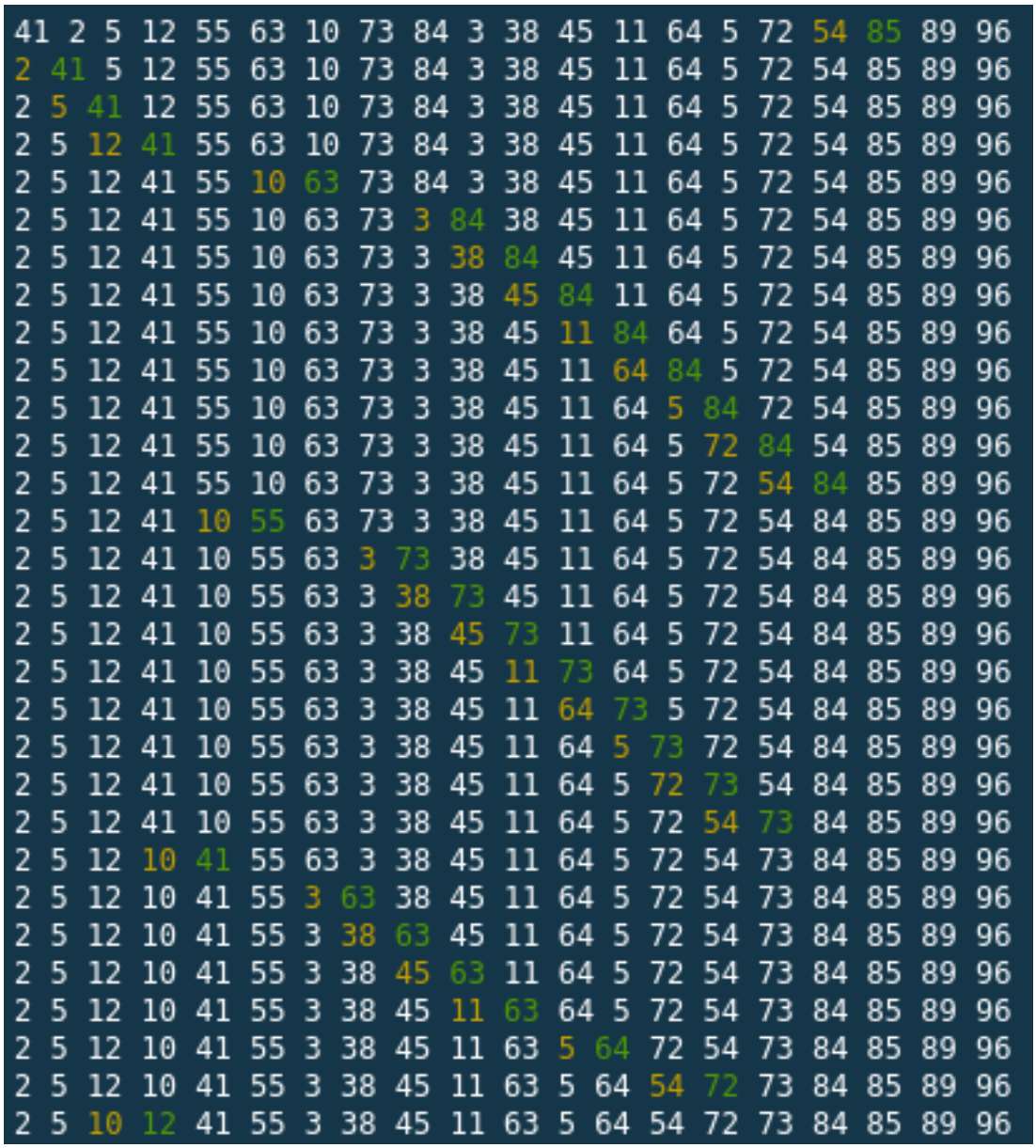

Figure 10: Algoritmo Bubble Sort Imagen 3

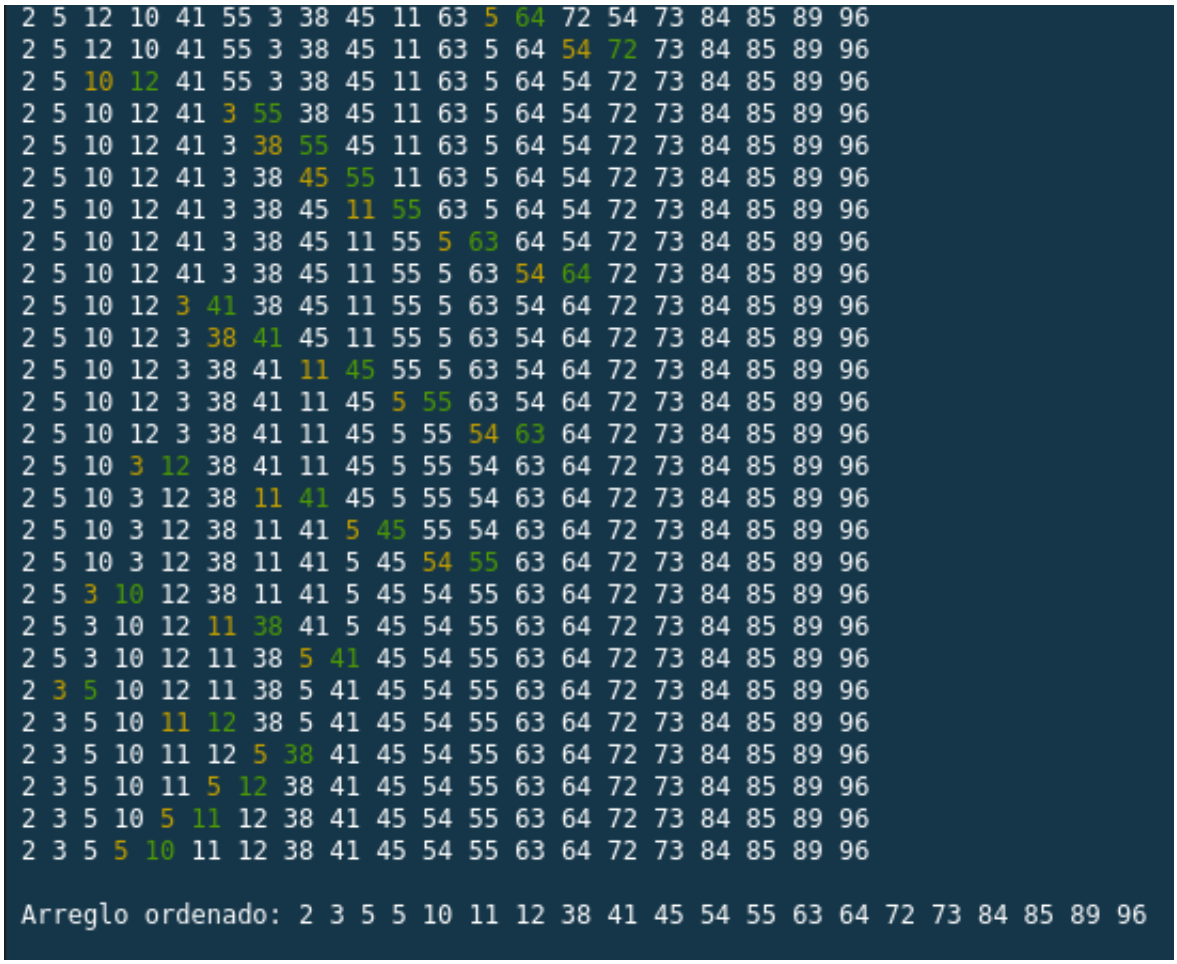

Figure 11: Algoritmo Bubble Sort Imagen 4

2.3.4 Merge Sort

El ultimo algortimo ingresado es *MergeSort()* y sinceramente es un algoritmo **MUY DIFICIL** de representar a traves de *printf()* como se ejecuta.

Una de las cosas que se me ocurrio fue representar los niveles que va generando a traves de su recursion e indicar cuando ingresa a la funcion *merge()* asi como cuando sale de cada una de las funciones recursivas, creo que quedo medianamente comprensible, pero en realidad va generando tantas ramas que es un poco complicado demostrar su funcionamiento si no se utiliza un esquema de arbol para hacerlo.

Se incluyen las imagenes de todas formas:

```
1) Elaborar arreglo
2) Imprimir arreglo
3) SelectionSort
4) InsertionSort
5) HeapSort
6) QuickSort
7) BubbleSort
8) MergeSort
9) Salir
8

Arreglo sin ordenar: 64 34 45 49 20 17 84 90 21 83 32 76 20 6 70 22 11 34 85 59

Nivel Inicial, Profundidad: 1   | 64 34 45 49 20 17 84 90 21 83 32 76 20 6 70 22 11 34 85 59
Sort 1, Profundidad: 2  | 64 34 45 49 20 17 84 90 21 83
Sort 1, Profundidad: 3  | 64 34 45 49 20
Sort 1, Profundidad: 4  | 64 34 45
Sort 1, Profundidad: 5  | 64 34
Sort 1, Profundidad: 6  | 64
        Sort 1, Profundidad: 6 OUT
Sort 2, Profundidad: 6  | 64 34
        Sort 2, Profundidad: 6 OUT
        Merge profundidad: 5 |34 64
        Sort 1, Profundidad: 5 OUT
Sort 2, Profundidad: 5  | 34 64 45
        Sort 2, Profundidad: 5 OUT
        Merge profundidad: 4 |34 45 64
        Sort 1, Profundidad: 4 OUT
Sort 2, Profundidad: 4  | 34 45 64 49 20
Sort 1, Profundidad: 5  | 34 45 64 49
        Sort 1, Profundidad: 5 OUT
Sort 2, Profundidad: 5  | 34 45 64 49 20
        Sort 2, Profundidad: 5 OUT
```

Figure 12: Funcionamiento Merge Sort Ejemplo 1

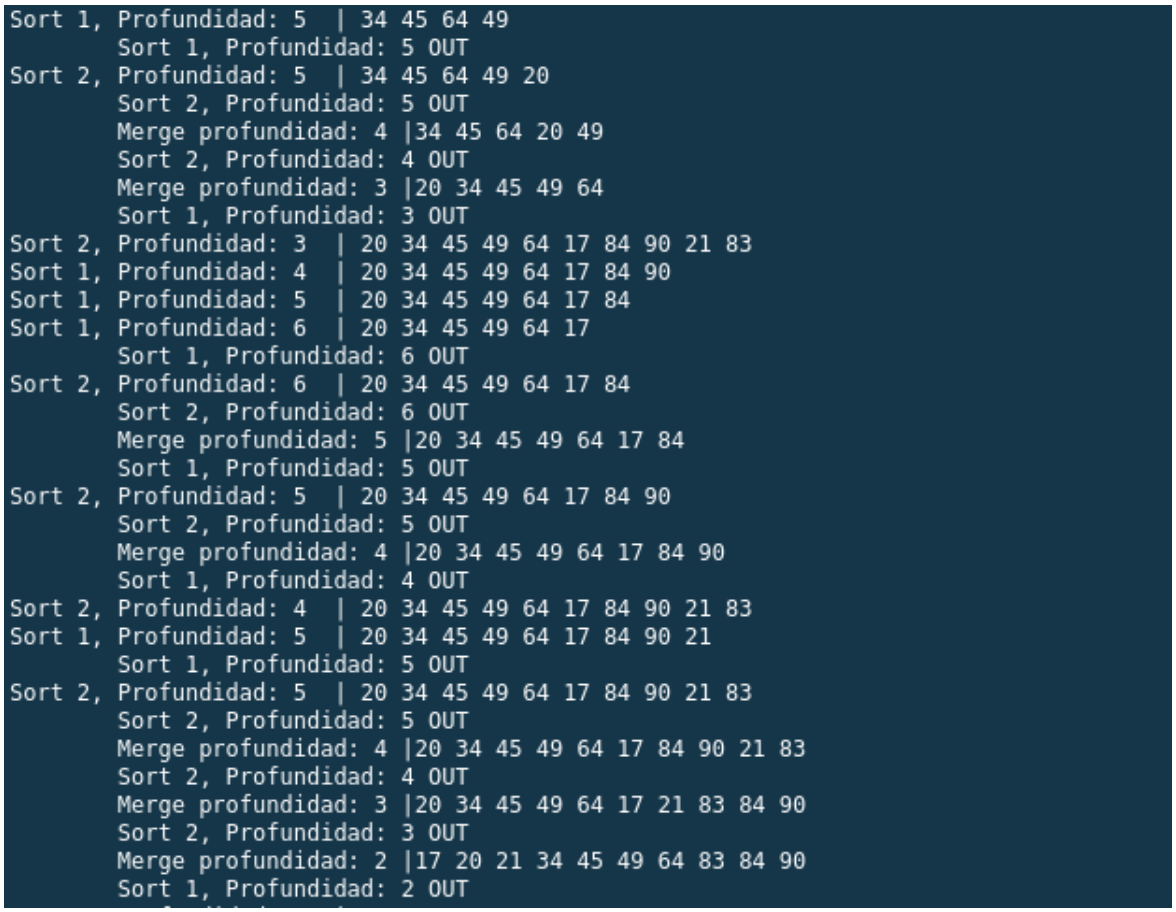

Figure 13: Funcionamiento Merge Sort Ejemplo 2

```
        Sort 1, Profundidad: 2 OUT
Sort 2, Profundidad: 2  | 17 20 21 34 45 49 64 83 84 90 32 76 20 6 70 22 11 34 85 59
Sort 1, Profundidad: 3  | 17 20 21 34 45 49 64 83 84 90 32 76 20 6 70
Sort 1, Profundidad: 4  | 17 20 21 34 45 49 64 83 84 90 32 76 20
Sort 1, Profundidad: 5  | 17 20 21 34 45 49 64 83 84 90 32 76
Sort 1, Profundidad: 6  | 17 20 21 34 45 49 64 83 84 90 32
        Sort 1, Profundidad: 6 OUT
Sort 2, Profundidad: 6  | 17 20 21 34 45 49 64 83 84 90 32 76
        Sort 2, Profundidad: 6 OUT
        Merge profundidad: 5 |17 20 21 34 45 49 64 83 84 90 32 76
        Sort 1, Profundidad: 5 OUT
Sort 2, Profundidad: 5  | 17 20 21 34 45 49 64 83 84 90 32 76 20
        Sort 2, Profundidad: 5 OUT
        Merge profundidad: 4 |17 20 21 34 45 49 64 83 84 90 20 32 76
        Sort 1, Profundidad: 4 OUT
Sort 2, Profundidad: 4  | 17 20 21 34 45 49 64 83 84 90 20 32 76 6 70
Sort 1, Profundidad: 5  | 17 20 21 34 45 49 64 83 84 90 20 32 76 6
        Sort 1, Profundidad: 5 OUT
Sort 2, Profundidad: 5  | 17 20 21 34 45 49 64 83 84 90 20 32 76 6 70
        Sort 2, Profundidad: 5 OUT
        Merge profundidad: 4 |17 20 21 34 45 49 64 83 84 90 20 32 76 6 70
        Sort 2, Profundidad: 4 OUT
        Merge profundidad: 3 |17 20 21 34 45 49 64 83 84 90 6 20 32 70 76
        Sort 1, Profundidad: 3 OUT
Sort 2, Profundidad: 3  | 17 20 21 34 45 49 64 83 84 90 6 20 32 70 76 22 11 34 85 59
Sort 1, Profundidad: 4  | 17 20 21 34 45 49 64 83 84 90 6 20 32 70 76 22 11 34
Sort 1, Profundidad: 5  | 17 20 21 34 45 49 64 83 84 90 6 20 32 70 76 22 11
Sort 1, Profundidad: 6  | 17 20 21 34 45 49 64 83 84 90 6 20 32 70 76 22
        Sort 1, Profundidad: 6 OUT
Sort 2, Profundidad: 6  | 17 20 21 34 45 49 64 83 84 90 6 20 32 70 76 22 11
        Sort 2, Profundidad: 6 OUT
        Merge profundidad: 5 |17 20 21 34 45 49 64 83 84 90 6 20 32 70 76 11 22
        Sort 1, Profundidad: 5 OUT
```

Figure 14: Funcionamiento Merge Sort Ejemplo 3

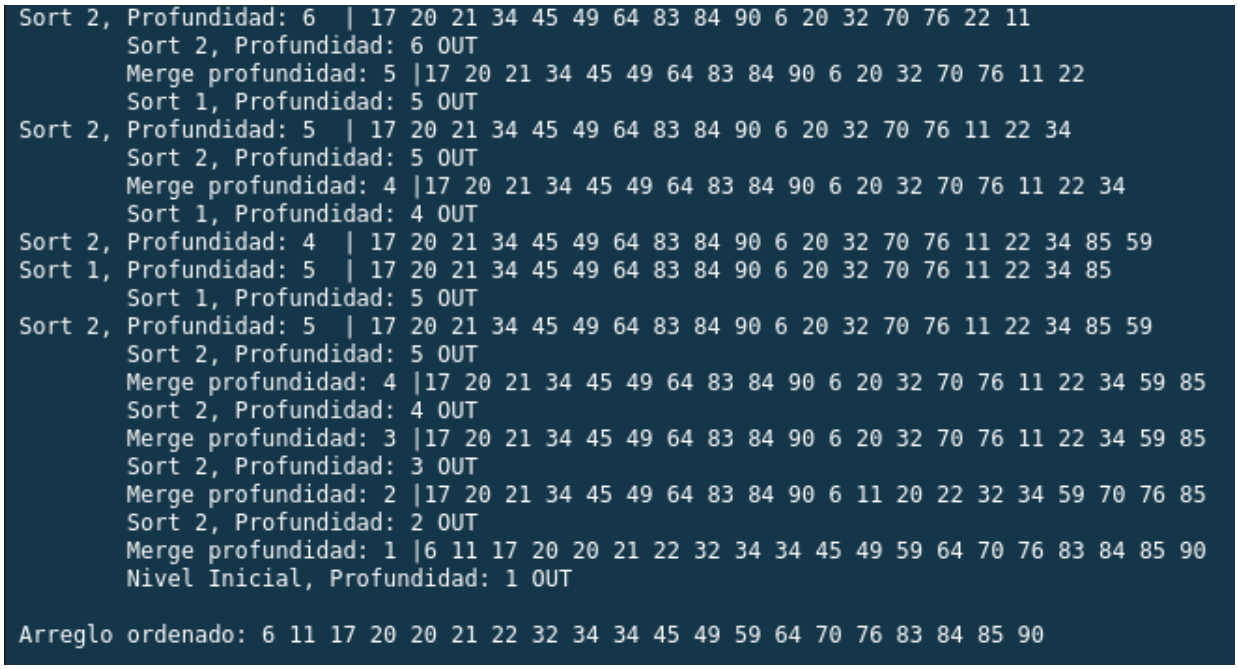

Figure 15: Funcionamiento Merge Sort Ejemplo 4

2.4 Ejercicio 4

Para este ejercicio modifique el codigo que realice en la practica pasada sustituyendo los algoritmos pasados por los nuevos pero ademas agregue algunas modificaciones al codigo de los algoritmos para que se adecuaran a las iteraciones correctas o para que finalizaran sus funciones en caso de que la lista este ordenada.

Una de las cosas que se agregaron a *BubbleSort()* por ejemplo, fue un tipo de *switch* que simplemente se activa en caso de que el programa ejecute algun *swap* (pues eso significaria que el algortimo aun no esta del todo ordenado) y en caso de que ningun *swap* se ejecute, entonces el algoritmo tiene instrucciones para que finalice el proceso.

De igual forma, agregue instrucciones a *QuickSort()* para que hiciera algo similar. Lo que hice fue que en su funcion *partition()* agregue un contador para que sumara el numero de veces que hace un *swap* y si este numero de veces es igual al tamano del arreglo menos una unidad y ademas el contador **NO** es igual a 0, entonces el algortmo de igual manera, termina su ejecucion enviando un numero negativo a la funcion principal.

Para *MergeSort()* no agregue ningun tipo de funcionalidad para terminar el algoritmo, puesto que basado en el modelo RAM, a pesar de que tiene varios *while-loops* dentro de la funcion *merge()* no se considera que afecten la complejidad temporal del algoritmo.

2.4.1 Graficas de Complejidad

En general, a excepcion de *MergeSort()* que tiene una complejidad de $O(nlog(n))$ practicamente para cualquier caso, los otros 2 algoritmos si sucede que su complejidad temporal es mas alta, especialmente para casos en los que las listas estan organizadas de forma descendente.

Como en el ejercicio pasado, se realizaron un total de 45 pruebas que incluyeron arreglos de diferentes tamanos y ordenados tanto de forma ascedente, descendente y aleatoria.

Se escogieron estas cantidades numericas principalmente por que se incuyen 3 potencias de 10 (10, 100 y 1000), 1 cantidad intermedia (500) y el doble de la 3ra potencia (2000) pues apartir de esa cantidad las graficas comienzan a crecer muy rapido.

Esta es la tabla de resultados generales que obtuve en base a las pruebas que realice:

QuickSort	10	100	500	1000	2000
Ascendente	10	100	500	1000	2000
Descendente	64	5149	125749	501499	2002999
Aleatorio	36	761	5650	14208	38062
Bubble Sort	10	100	500	1000	2000
Ascendente	9	99	499	999	1999
Descendente	45	4950	124750	499500	1999000
Aleatorio	45	4950	124750	499500	1999000
Merge Sort	10	100	500	1000	2000
Ascendente	28	298	1498	2998	5998
Descendente	28	298	1498	2998	5998
Aleatorio	28	298	1498	2998	5998

Figure 16: Resultado de suma de ciclos por cantidad de elementos

2.4.2 Quick Sort

La grafica de Quick Sort, tiene diferentes resultados cuando trabaja con un arreglo aleatorio, pero cuando se trata de arreglos ascendentes o descendentes, sus ciclos se mantienen constantes.

Se puede apreciar como los arreglos ordenados de forma descendente aumentan de forma exponencial los ciclos que el algoritmo requiere para finalizar su ejecucion. Esto se da probablemente porque en esta version de *QuickSort()* se toma el ultimo elemento como pivote y debido a que en este tipo de arreglos el ultimo elemento es el menor, entonces probablemente es por esa razon que su tiempo de ejecucion aumenta considerablemente.

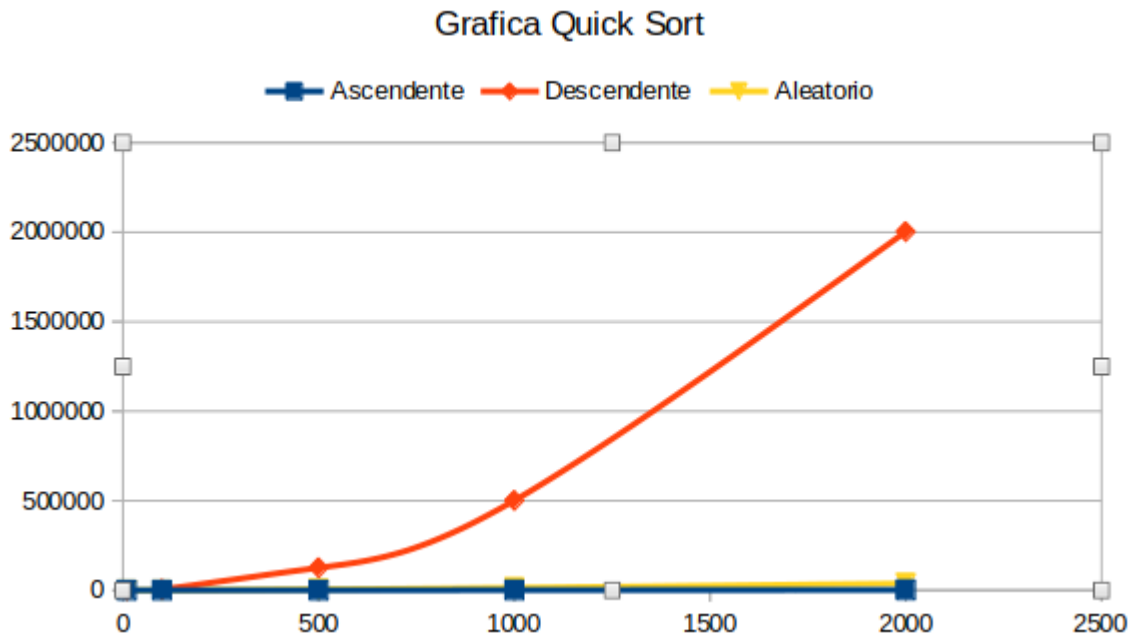

Figure 17: Grafica QuickSort, en el eje x se muestran los datos y en el y los ciclos

2.4.3 Bubble Sort

En cuanto al algoritmo de *BubbleSort()*, por la forma en la que este se comporta, como su nombre bien lo dice, en forma de "burbuja", unicamente puede llegar al caso de $O(n)$ cuando todos sus elementos estan ordenados de forma ascendente.

En cualquier otro caso, el algoritmo tiene un tiempo similar al de $O(n^2)$ siendo en las listas descendentes cuando principalmente se comporta de esta manera, pues al momento de tener al numero menor en el otro extremo del arreglo y al mayor en el extremo inicial, este algoritmo recorre practicamente, $n * n$ veces el arreglo para poder ordenarlo de forma ascendente.

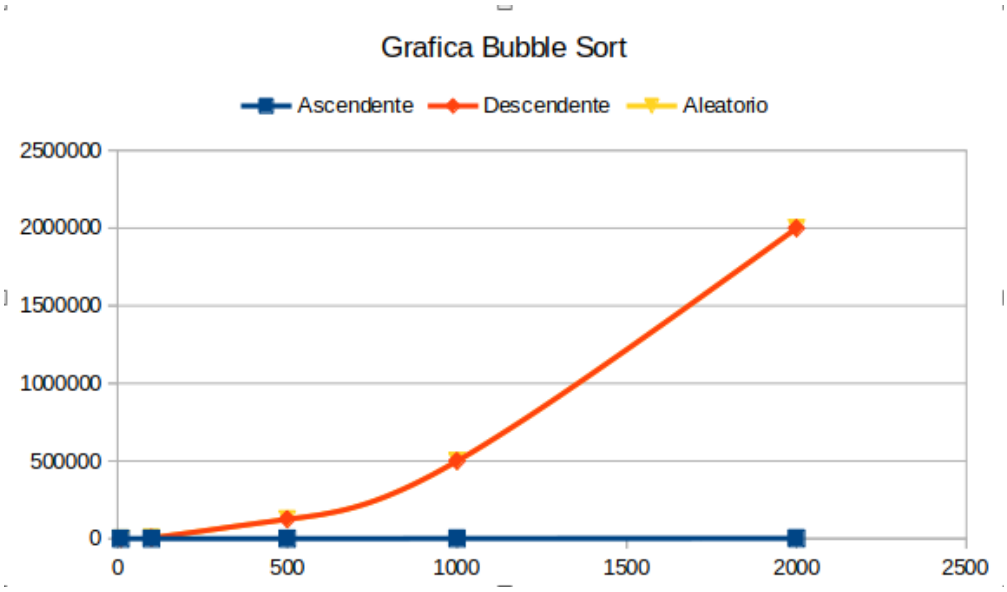

Figure 18: Grafica BubbleSort, en el eje x se muestran los datos y en el y los ciclos

2.4.4 Merge Sort

MergeSort() que se basa en la tecnica *"divide and conquer"* tiene un tiempo estable porque en realidad es un algoritmo que no toma en cuenta los valores del arreglo hasta que se encuentra todo subdividido dentro de las funciones recursivas que ejecuta y es ahi cuando comienza el proceso de arreglo, por lo que *MergeSort()* no es posible limitarlo cuando se le transfiere un arreglo ya ordenado, pues tal vez la unica manera de que *MergeSort()* tuviera una complejidad de $O(n)$ para un arreglo ascendente, seria evaluar el arreglo con anterioridad de forma lineal y asi determinar que no es necesario usar el algoritmo.

Esta es basicamente la razon por la que su grafica se ve casi lineal, pues para todos los casos *MergeSort()* hace practicamente lo mismo y el proceso de ordenamiento no se entiende como modificador de la complejidad temporal bajo el modelo *RAM*.

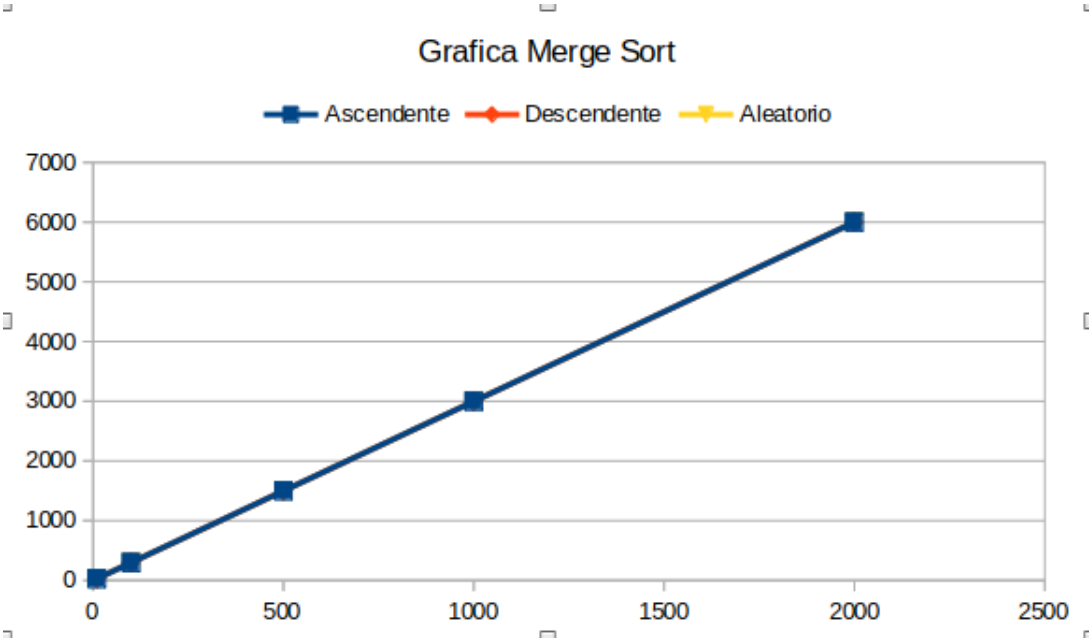

Figure 19: Grafica MergeSort, en el eje x se muestran los datos y en el y los ciclos

2.4.5 Output

Como en la practica pasada, se adjuntan las capturas de pantalla de como se comporto el programa automatizado al momento de ejecutarse.

```
~/eda2/practica2/ejercicio4/ $ ./ejercicio4
Tipo: Ascendente, Elementos: 10
-- Quick Sort --
Big O: Worst case: n^2, Best case: nlog(n)
Corriendo...
Total: 10

Tipo: Descendente, Elementos: 10
-- Quick Sort --
Big O: Worst case: n^2, Best case: nlog(n)
Corriendo...
Total: 64

Tipo: Aleatorio, Elementos: 10
-- Quick Sort --
Big O: Worst case: n^2, Best case: nlog(n)
Corriendo...
Total: 36

Tipo: Ascendente, Elementos: 100
-- Quick Sort --
Big O: Worst case: n^2, Best case: nlog(n)
Corriendo...
Total: 100

Tipo: Descendente, Elementos: 100
-- Quick Sort --
Big O: Worst case: n^2, Best case: nlog(n)
Corriendo...
Total: 5149
```

Figure 20: Output programa automarizado

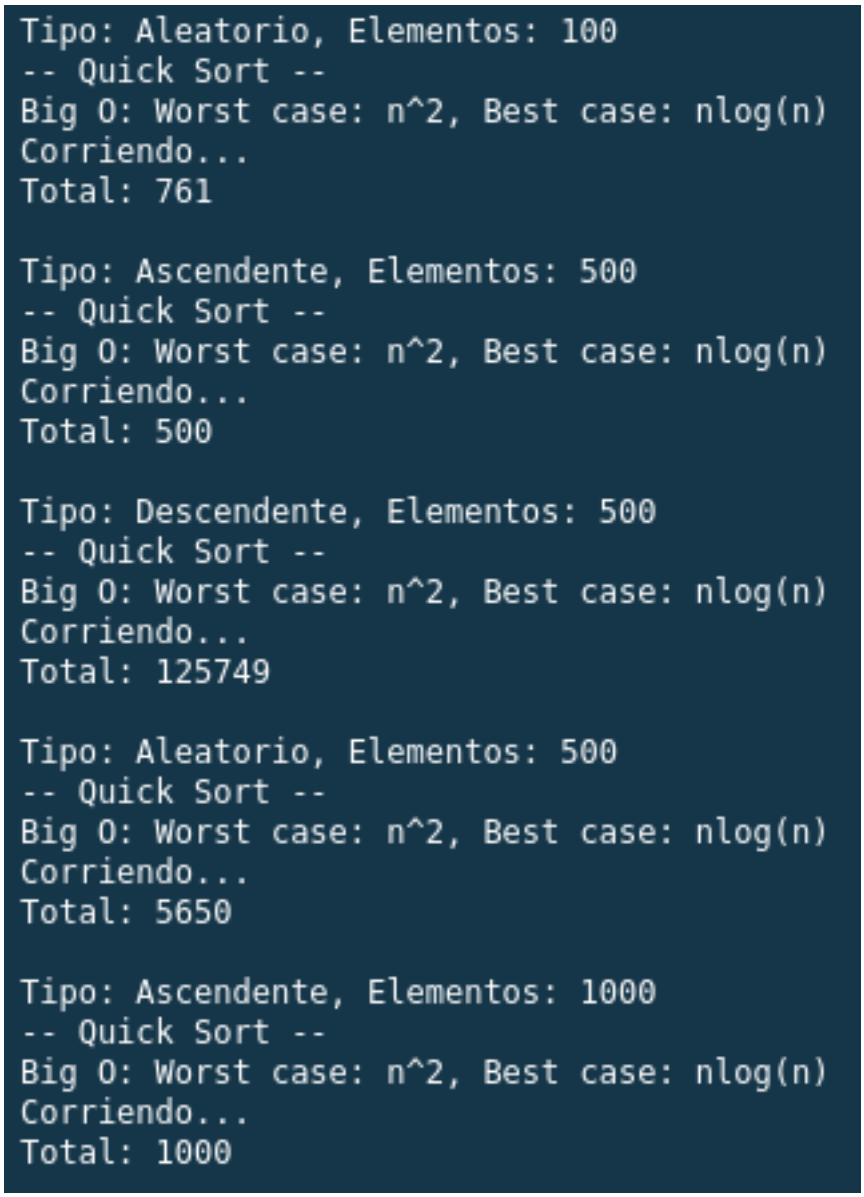

Figure 21: Output programa automarizado

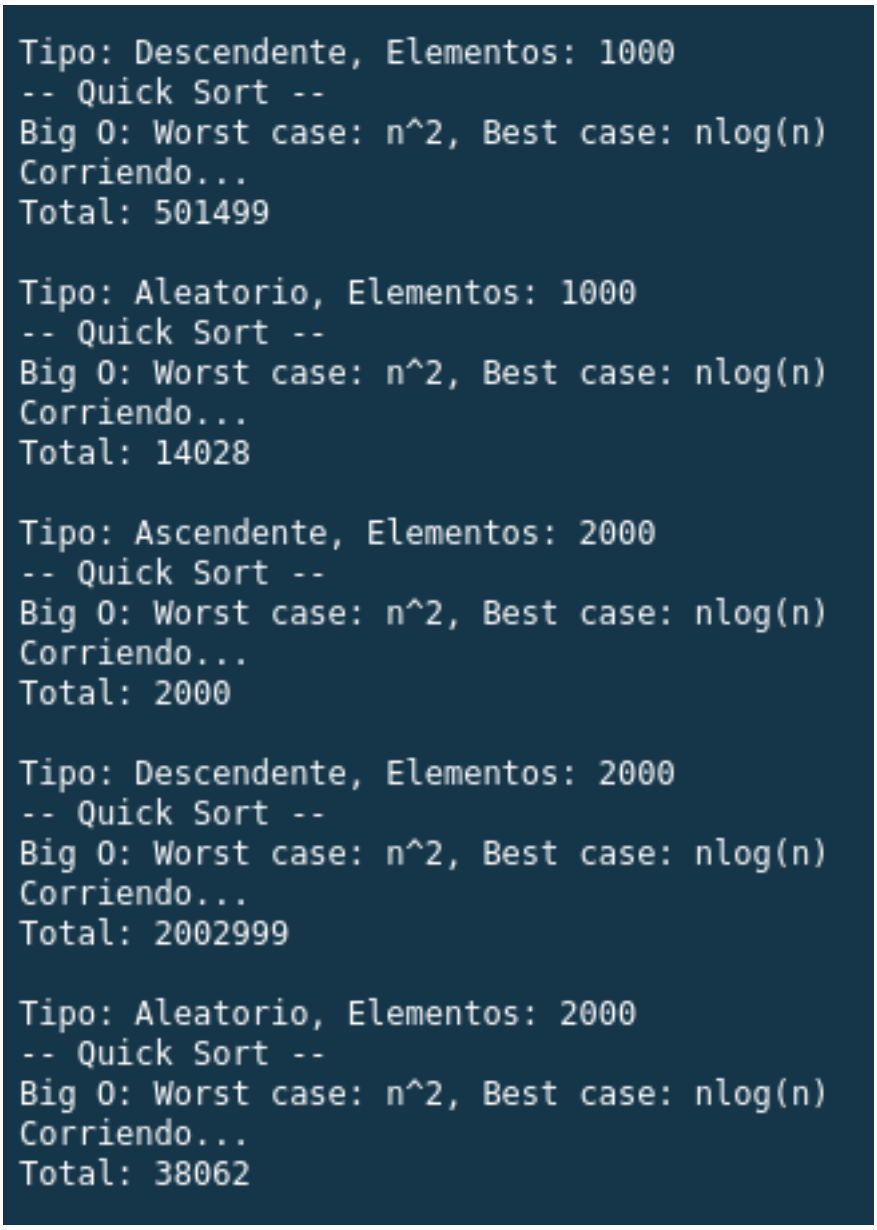

Figure 22: Output programa automarizado

```
Tipo: Ascendente, Elementos: 10
-- Bubble Sort --
Big O: Worst case: n^2, Best case: n
Corriendo...
Total: 9

Tipo: Descendente, Elementos: 10
-- Bubble Sort --
Big O: Worst case: n^2, Best case: n
Corriendo...
Total: 45

Tipo: Aleatorio, Elementos: 10
-- Bubble Sort --
Big O: Worst case: n^2, Best case: n
Corriendo...
Total: 45

Tipo: Ascendente, Elementos: 100
-- Bubble Sort --
Big O: Worst case: n^2, Best case: n
Corriendo...
Total: 99

Tipo: Descendente, Elementos: 100
-- Bubble Sort --
Big O: Worst case: n^2, Best case: n
Corriendo...
Total: 4950
```

Figure 23: Output programa automarizado

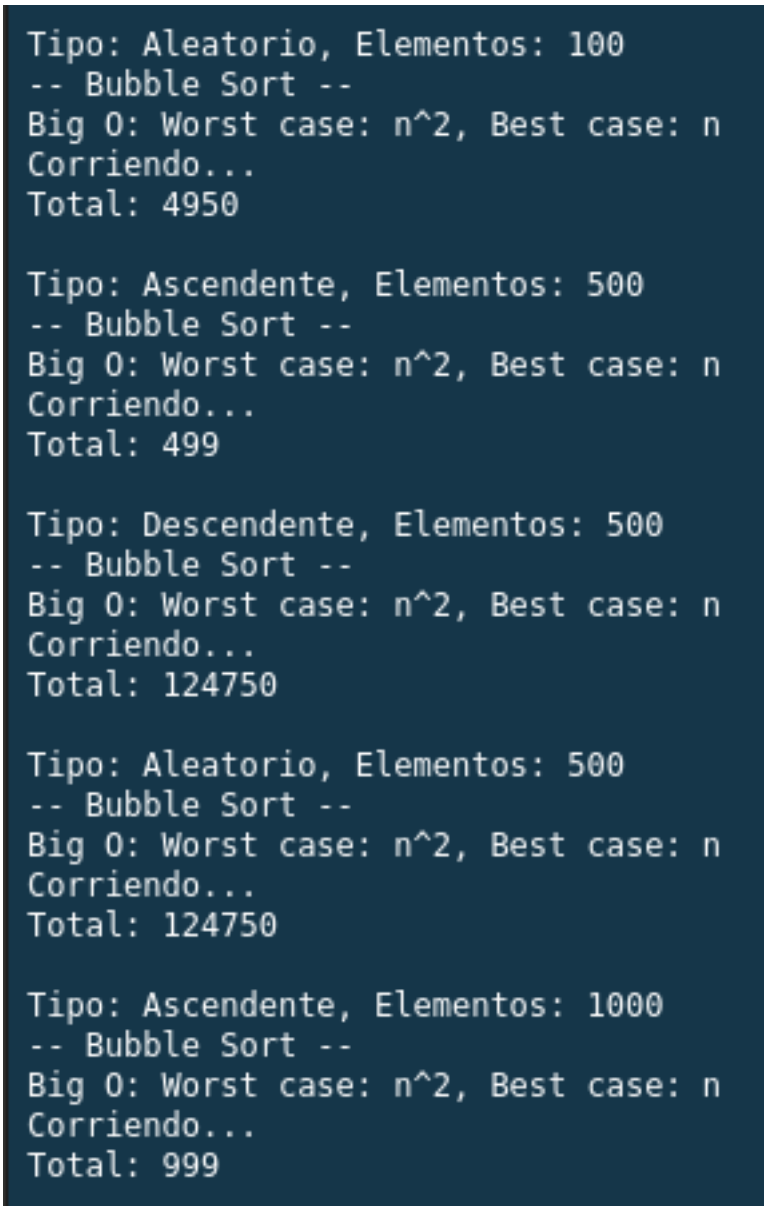

Figure 24: Output programa automarizado

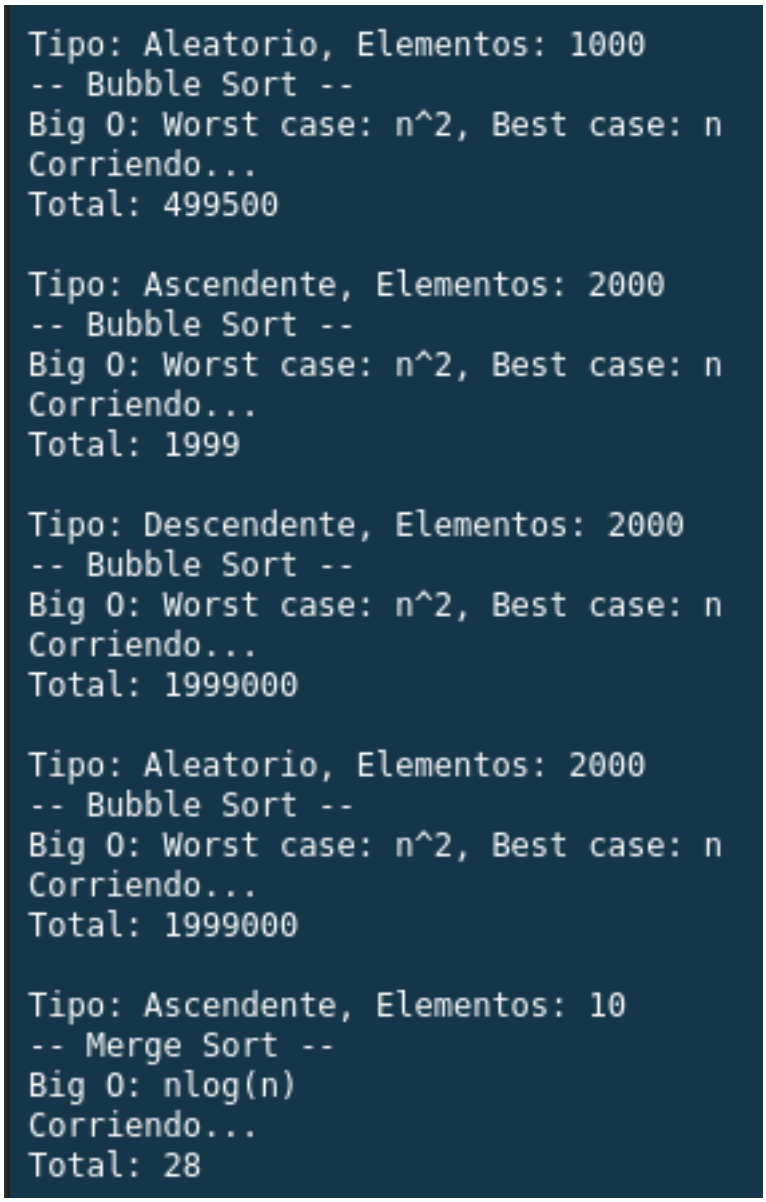

Figure 25: Output programa automarizado

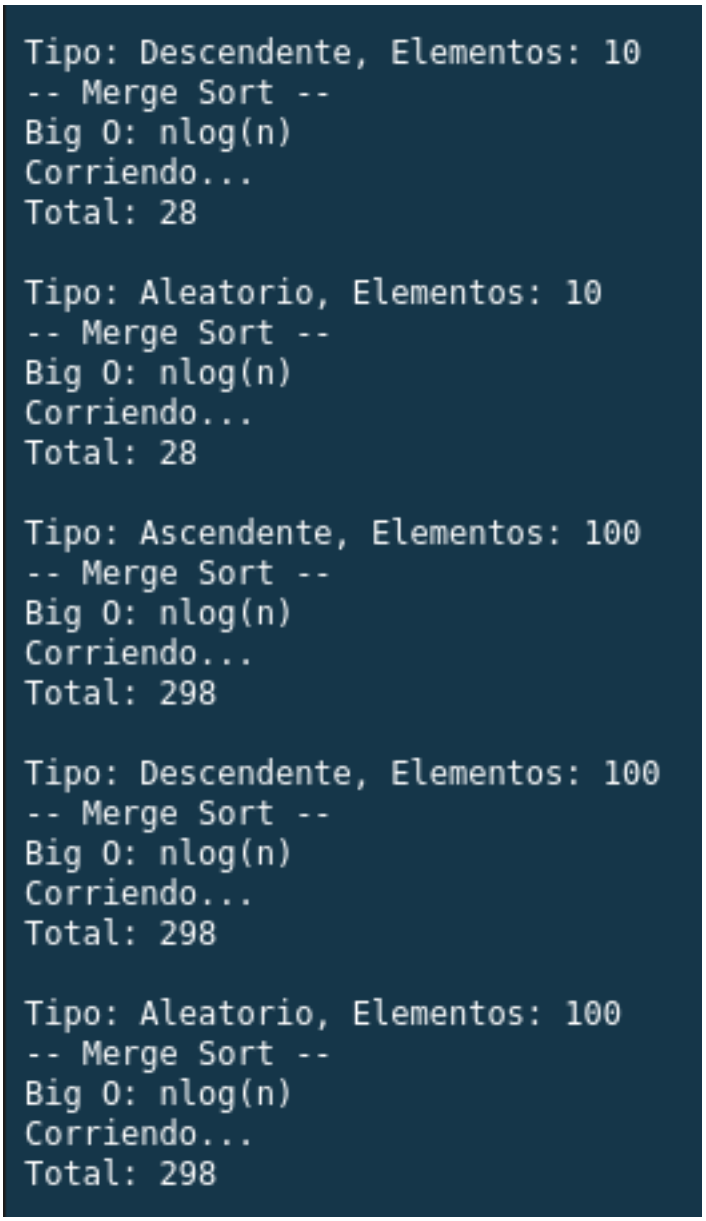

Figure 26: Output programa automarizado

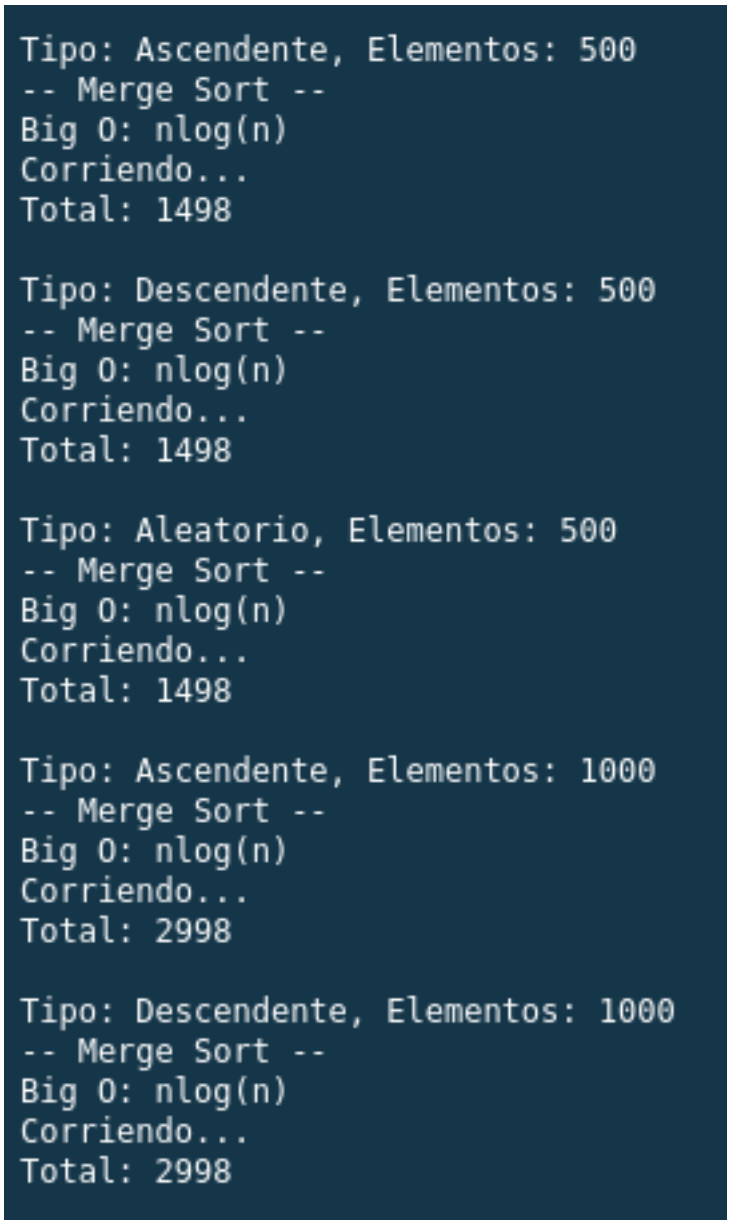

Figure 27: Output programa automarizado

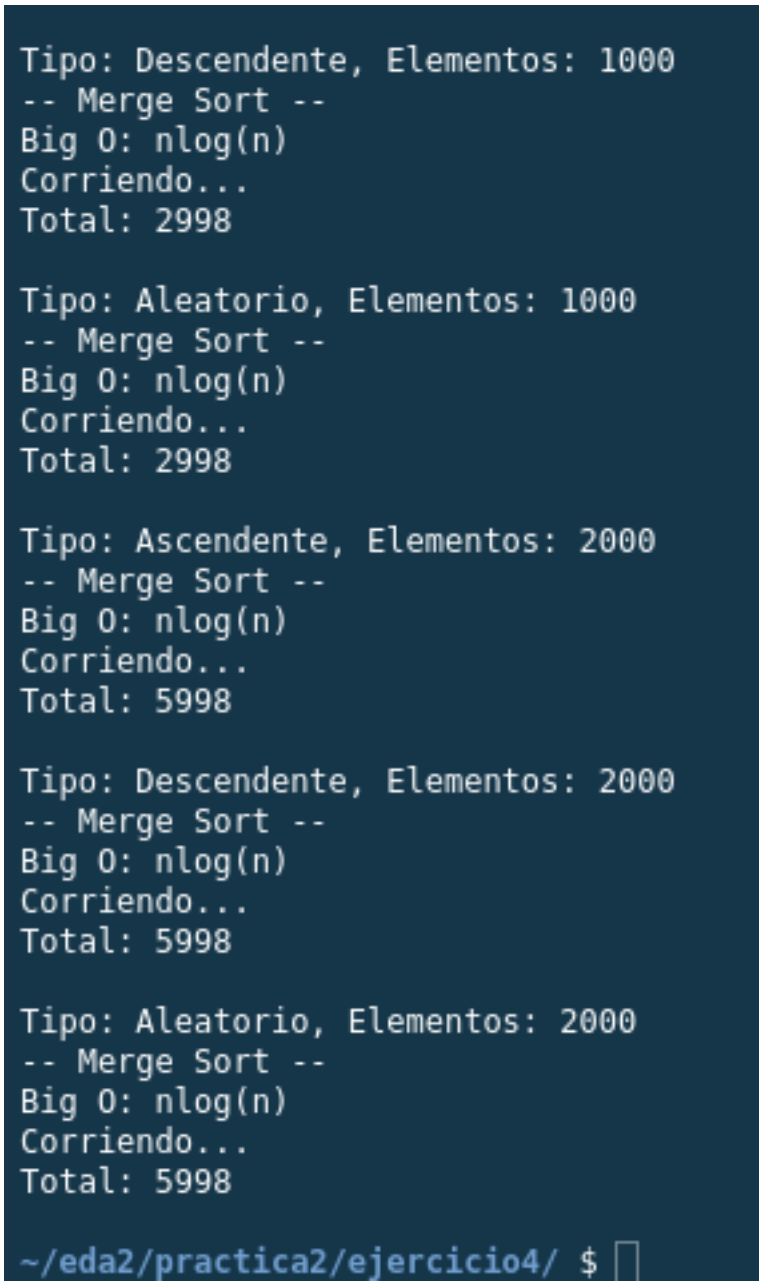

Figure 28: Output programa automarizado

2.5 Ejercicio 5

El ejercicio en realidad no se me dificulto, pero si tuve que hacer algunas busquedas principalmente sobre como declarar un arreglo, pues no sabia como se declaraban los arreglos en *Java* pero al estilo de *Java*, que principalmente es una forma diferente a como se hace en *C* y que se escribe de la siguiente manera:

int[ ] foo = new int[num];

Por otro lado me parece que es la primera ocasion que creo un metodo por mi parte y lo llamo desde otra clase.

Una de las cosas que son interesantes en *Java* es que yo suponia que habia que agregar las clases de la misma forma que se hace en *C* con los archivos con extensiones `.c` o `.h` que se deben de incluir al principio del documento.

Me parecde que con *Java* precisamente por la forma en la que trabaja el *JDK* que basicamente genera un sistema donde todas las clases navegan en el mismo ambiente, pues les es posible interactuar unas con otras ya que dentro del *JDK* de cierta forma se encuentran dentro del mismo *"scope"*.

Otra cosa que no me queda aun bien clara, es que no hubo problema en quitar la descripcion `public` de la clase y funciones dentro del archivo `Utilidades.java`, pues yo tenia la idea de que si no se indicaba que la clase es publica, entonces no se puede tener acceso a ella. No se si esto sea considerado una buena o mala practica, pues apenas me estoy familiriazando con todo el lenguaje.

Por otro lado quisiera aprender mas sobre los *getters* y *setters* que aun no he podido aprender adecuadamente.

En general, fuera de investigar como declarar un arreglo, el ejercicio lo hice yo solo y no me parecio especialmente complicada la implementacion.

El programa que hice, simplemente declara las clases que se mencionan en la practica y utilizo la clase *Scanner* para obtener 20 numeros enteros de un usuario a traves de un *for-loop* que despues de ser almacenados en un arreglo con capacidad para 20 elementos previamente creado (tampoco sabia que los arreglos en *Java* son dinamicos), utilizo el metodo *.quickSort()* para ordenar esos elementos, invocandolo desde la clase *QuickSort*.

Adicionalmente como lo recomienda la practica, tambien agregue la funcionalidad necesaria dentro del metodo *.printSubArray()* perteneciente a la clase *Utilidades* para poder imprimir las iteraciones que genera el metodo *.quickSort()* al momento de ordenar el arreglo creado por el propio usuario.

Finalmente, cuando el algoritmo termina de ejecutarse, se imprime como

salida a la pantalla y el programa finaliza.

A continuacion se adjuntan imagenes de como se ejecuta mi programa:

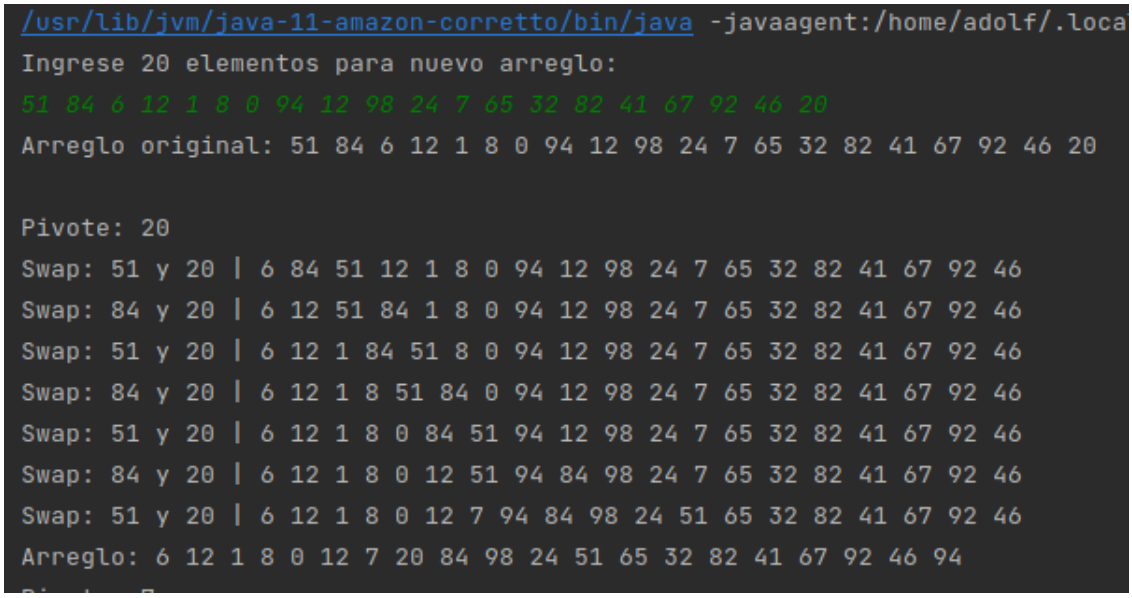

Figure 29: Output QuickSort en IntelliJ IDEA

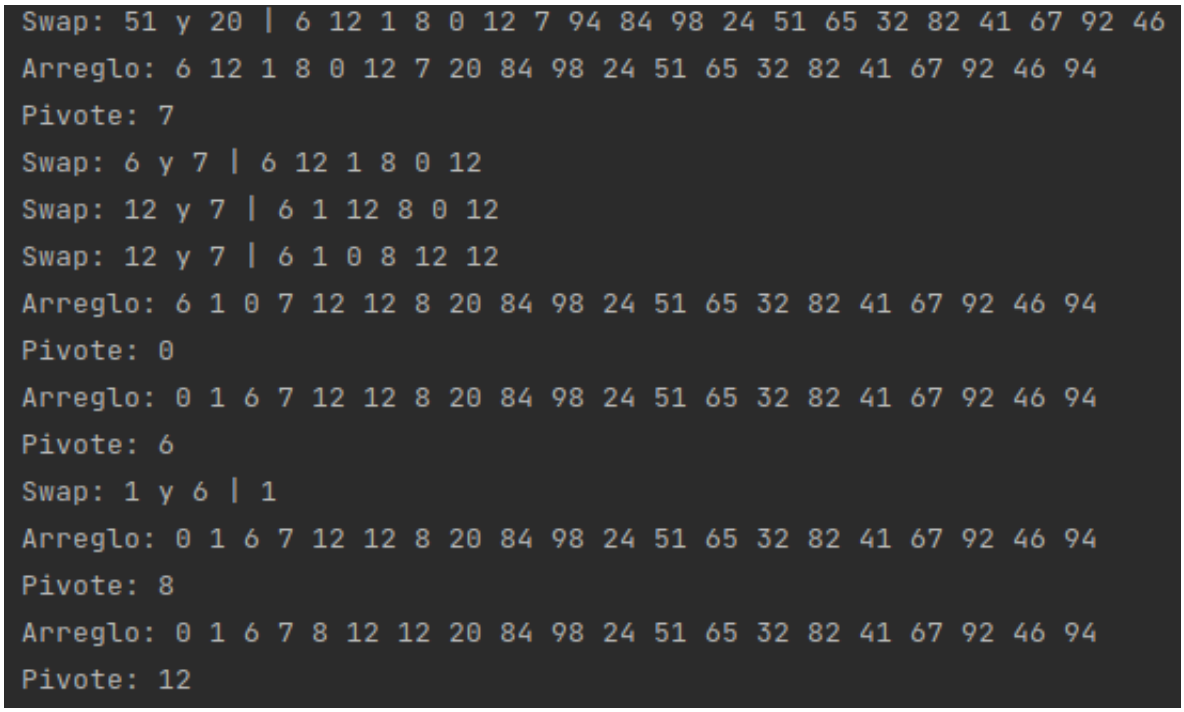

Figure 30: Output QuickSort en IntelliJ IDEA

```
Pivote: 12
Swap: 12 y 12 | 12
Arreglo: 0 1 6 7 8 12 12 20 84 98 24 51 65 32 82 41 67 92 46 94
Pivote: 94
Swap: 84 y 94 | 84 98 24 51 65 32 82 41 67 92 46
Swap: 98 y 94 | 84 24 98 51 65 32 82 41 67 92 46
Swap: 98 y 94 | 84 24 51 98 65 32 82 41 67 92 46
Swap: 98 y 94 | 84 24 51 65 98 32 82 41 67 92 46
Swap: 98 y 94 | 84 24 51 65 32 98 82 41 67 92 46
Swap: 98 y 94 | 84 24 51 65 32 82 98 41 67 92 46
Swap: 98 y 94 | 84 24 51 65 32 82 41 98 67 92 46
Swap: 98 y 94 | 84 24 51 65 32 82 41 67 98 92 46
Swap: 98 y 94 | 84 24 51 65 32 82 41 67 92 98 46
Swap: 98 y 94 | 84 24 51 65 32 82 41 67 92 46 98
Arreglo: 0 1 6 7 8 12 12 20 84 24 51 65 32 82 41 67 92 46 94 98
Pivote: 46
Swap: 84 y 46 | 24 84 51 65 32 82 41 67 92
Swap: 84 y 46 | 24 32 51 65 84 82 41 67 92
Swap: 51 y 46 | 24 32 41 65 84 82 51 67 92
Arreglo: 0 1 6 7 8 12 12 20 24 32 41 46 84 82 51 67 92 65 94 98
Pivote: 41
Swap: 24 y 41 | 24 32
```

Figure 31: Output QuickSort en IntelliJ IDEA

```
Arreglo: 0 1 6 7 8 12 12 20 24 32 41 46 84 82 51 67 92 65 94 98
Pivote: 41
Swap: 24 y 41 | 24 32
Swap: 32 y 41 | 24 32
Arreglo: 0 1 6 7 8 12 12 20 24 32 41 46 84 82 51 67 92 65 94 98
Pivote: 32
Swap: 24 y 32 | 24
Arreglo: 0 1 6 7 8 12 12 20 24 32 41 46 84 82 51 67 92 65 94 98
Pivote: 65
Swap: 84 y 65 | 51 82 84 67 92
Arreglo: 0 1 6 7 8 12 12 20 24 32 41 46 51 65 84 67 92 82 94 98
Pivote: 82
Swap: 84 y 82 | 67 84 92
Arreglo: 0 1 6 7 8 12 12 20 24 32 41 46 51 65 67 82 92 84 94 98
Pivote: 84
Arreglo: 0 1 6 7 8 12 12 20 24 32 41 46 51 65 67 82 84 92 94 98

Arreglo despues de QuickSort: 0 1 6 7 8 12 12 20 24 32 41 46 51 65 67 82 84 92 94 98

Process finished with exit code 0
```

Figure 32: Output QuickSort en IntelliJ IDEA

3 Conclusiones

Durante esta practica que fue similar a la pasada, uno de los algoritmos que se me complico un poco fue *Quick Sort* pues su implementacion es un poco confusa y no entendia apropiadamente en que partes realizaba el intercambio de elementos o en que partes no.

Una de las partes donde no sabia concretamente que hacer fue al momento que en la funcion *partition()*, el algoritmo lo que evalua es si el elemento en **j** es menor o igual al pivote y pues al momento evaluar arreglos previamente ordenados, el algoritmo continuaba evaluando valores a pesar de que estos ya estuvieran ordenados, por lo que tuve que idear una forma diferente de poder detener la funcion y ayudarle a determinar el momento en el que un arreglo ya se encuentra ordenado.

Para *Bubble Sort* esa parte no me fue tan complicada en realidad, simplemente idee una manera de sencilla de detener las iteraciones del algoritmo siempre y cuando no haya realizado ningun cambio al arreglo sobre el que se ejecutaba lo que significaria que el arreglo entonces ya esta ordenado.

Sin embargo para *Merge Sort* lo que me parecio mas complejo fue poder representar alguna forma no tan dificil de imprimir la logica del algoritmo, pues *Merge Sort* genera muchisimas ramificaciones que es dificil representar en una forma lineal.

Por otro lado me di cuenta como el modelo *RAM* funciona para casos como el de *Merge Sort*, pues contabilizando todos los ciclos que el algoritmo lleva a cabo para ordenar un arreglo, se puede dar uno cuenta que se transforman en miles para unos cuantos elementos, pero estos ciclos al no estar identados dentro de otros *loops* pues no afectan la complejidad del programa.

Con respecto de *Java* aun no me siento del todo familiarizado con el lenguaje y creo que me falta aun mas practica, aunque en realidad la logica de los programas es muy similar. Un profesor de computacion explicaba que para una persona que se dedica de manera profesional al area de la computacion, aprender un nuevo lenguaje es similar a manejar un carro, pues cuando sabes manejar solo es cuestion de un poco de practica manejar el carro que desees pues todos los carros utilizan la misma logica de manejo. Para los lenguajes de programacion la situacion es similar y con *Java*, a pesar de no conocer mucho sobre el lenguaje, tampoco lo siento imposible ni mucho menos, pues una vez que se conoce mas o menos como funciona la logica de los lenguajes de programacion, lo demas es solo cuestion de tiempo y practica.

Me gusto mucho la practica y creo que se cumplieron muy bien los objetivos de la clase pues entiendi bastante sobre la implementacion de estos algoritmos, como calcular su complejidad, una de las multiples formas de implementar *Quick*

Sort y un poco de *Java*. Espero que pueda pasar a aprender a utilizar estos algoritmos que estuvimos viendo, no solo para ordenar listas de enteros, sino tambien para poder comenzar a usarlos para ordenar archivos o cosas mas complejas que me imagino es en realidad para lo que existen.

Para finalizar me da gusto que ya me siento mas fluido en LaTeXy se me ha facilitado mucho hacer las practicas en este procesador. Ya he aprendido mas sobre como insertar imagenes, insertar codigo, etc y pienso volverme mejor en esto ya que al parecer es una herramienta bastante util.

Muchas gracias por leer mi practica!