	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	TISTA GARCÍA EDGAR
Asignatura:	ESTRUCTURA DE DATOS Y ALGORITMOS II
Grupo:	5
No de Práctica(s):	PRACTICA #2 – ALGORITMOS DE ORDENAMIENTO PARTE 2
Integrante(s):	CARRILLO CERVANTES IVETTE ALEJANDRA
No. de Equipo de cómputo empleado:	TRABAJO EN CASA
No. de Lista o Brigada:	NO APLICA
Semestre:	2022 - 1
Fecha de entrega:	26 SEPTIEMBRE 2021
Observaciones:	

CALIFICACIÓN: _____

PRACTICA #2 – ALGORITMOS DE ORDENAMIENTO PARTE 2

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

OBJETIVO DE CLASE: El estudiante observará la importancia del orden de complejidad aplicado en algoritmos de ordenamiento, conocerá diferentes formas de implementar QuickSort y desarrollará habilidades básicas de programación orientada a objetos.

NOTA: Todos los ejercicios de esta práctica se realizaron en un solo archivo llamado: "MenuOrdenamientos.c"

ACTIVIDADES PARA EL DESARROLLO DE LA PRÁCTICA

Ejercicio 1. Agregando Ordenamientos

En la práctica 1 de Estructura de Datos y Algoritmos II, se realizó el analisis e implementación de los algoritmos de ordenamiento "*Insertion Sort*", "*Selection Sort*" y "*Heap Sort*". En esta práctica se realizará el analisis e implementación de los siguientes algoritmos de ordenamiento:

- Quick-Sort

Es un algoritmo de ordenamiento de intercambio, consiste en seleccionar un elemento de la colección el cual se va a considerar como "pivote" y compararlo con el resto de los elementos de la colección; después, se van a realizar los intercambios necesarios para que quede de tal manera que a la izquierda del elemento que se eligió como "pivote" se encuentren los elementos que sean menores o iguales a este, y a su izquierda los elementos mayores. La complejidad de este algoritmo en el mejor caso y caso promedio es de $O(nlog(n))$, mientras que su complejidad en el peor caso es de $O(n^2)$.

La función Quick-Sort tiene como parámetro el arreglo que se va a ordenar, el primer elemento de este, así como su último elemento, dentro de esta función tiene una estructura condicional la cual indica que si el primer elemento de la colección es menor que el último elemento entonces llamará a la función "*partition"* la cual tiene como objetivo dividir el arreglo en varios sub-arreglos (paradigma divide y vencerás).

El algoritmo que se proporcionó para esta práctica es diferente al algoritmo visto en clase, ya que en este último solo depende de una función en la cual se va a dividir el arreglo a ordenar en un principio y después entrará a un ciclo "*do-While"* el cual contendrá las instrucciones necesarias para ordenar el arreglo dado. También, al analizar detalladamente cada algoritmo se pudo observar que en el algoritmo de clase se tomó como "pivote" el elemento que se encuentra a mitad del arreglo, mientras que el algoritmo dado para la práctica se tomó el último elemento del arreglo como "pivote", se observó que esta modificación no impide el funcionamiento del algoritmo.

- Bubble-Sort

Es un algoritmo de ordenamiento de intercambio, consiste en verificar cada elemento de una colección dada en relación con su elemento inmediato siguiente, intercambiándose de posición si están en “orden inverso” al requerido. La complejidad de este algoritmo en el mejor caso, caso promedio y el peor de los casos es de $O(n^2)$.

La función Bubble Sort tiene como parámetro el arreglo a ordenar y el tamaño de este, la complejidad de este algoritmo se debe a que cuenta con dos ciclos “*for*” anidados, ya que se recorrerá la lista (con un ciclo “*for*”, disminuyendo el recorrido por 1 elemento en cada iteración) “n” número de veces (donde “n” es el número de elementos que tiene el arreglo), dentro de estos ciclos se encuentra una estructura condicional la cual indica que, si el elemento en donde se encuentra el indice es mayor que su elemento siguiente, se hará el cambio entre estos dos valores.

No obstante, en algunas ocasiones este algoritmo no es tan eficiente ya que puede que solo se tenga que modificar un elemento de la colección para ya estar ordenada y aun así el algoritmo debe de terminar sus recorridos en el arreglo, por lo tanto, sería un desperdicio de tiempo y memoria si es que el arreglo tiene un número muy grande de elementos. Para que no suceda esto, se hicieron las modificaciones necesarias para que al momento de que el arreglo ya este ordenado, se pare la ejecución. Para llevar a cabo esta implementación, se agregó un “else” en la estructura condicional que tiene la función, en este se agrega un contador el cual va aumentando cada vez que no se cumpla la condición de que el elemento sea mayor que su elemento siguiente, en otras palabras, que sea menor o igual a su siguiente elemento. Después, fuera del ciclo en el que se encuentra, se agregó una estructura condicional la que indica que, si el valor del contador es igual a el número de elementos recorridos en esa iteración, significa que la lista ya esta ordenada y entonces se termina la ejecución por medio de un “*break*”.

Se realizó una prueba con 10 elementos y se observó que la lista se ordeno en la iteración 5.

```
Iteración: 5
{ 63, 117, 202, 231, 423, 521, 535, 647, 652, 973}
{ 63, 117, 202, 231, 423, 521, 535, 647, 652, 973}
{ 63, 117, 202, 231, 423, 521, 535, 647, 652, 973}
{ 63, 117, 202, 231, 423, 521, 535, 647, 652, 973}
{ 63, 117, 202, 231, 423, 521, 535, 647, 652, 973}


  -> La lista ya esta ordenada en la iteración: 5 (:

El arreglo ordenado es: { 63, 117, 202, 231, 423, 521, 535, 647, 652, 973}

Numero de operaciones: 55
```

./Ejercicio123/MenuOrdenamientos.c

Ejercicio 2. Merge Sort

Se agregó el algoritmo de ordenamiento Merge-Sort a la biblioteca ordenamientos.h. Merge Sort es un algoritmo de ordenamiento de intercalación y al igual que Quick-Sort se basa en el paradigma divide y vencerás, consiste en dividir todo el arreglo en pequeños sub-arreglos hasta llegar a arreglos de tamaño 1; después, se estos se irán combinando para crear un nuevo arreglo ordenado de tamaño 2 el cual se irá combinando a su vez con otro que ya ha sido ordenado previamente para obtener un arreglo más grande, eso se repetirá sucesivamente hasta obtener el arreglo ordenado dado por el usuario.

Se basó en el pseudocódigo visto en clase para la implementación de este algoritmo, lo único que se le modificó a la función fue al momento de crear el arreglo temporal, ya que en el pseudocódigo se asume que será un arreglo creado con alguna otra función. Se creó este arreglo de tamaño p+r elementos, los cuales son el primer (p) y último (r) índice del arreglo original. Se verificó el funcionamiento de este código:

```
Seleccionaste Merge Sort:

El arreglo ordenado es: { 0, 34, 210, 230, 277, 354, 560, 641, 726, 808}
```

./Ejercicio123/MenuOrdenamientos.c

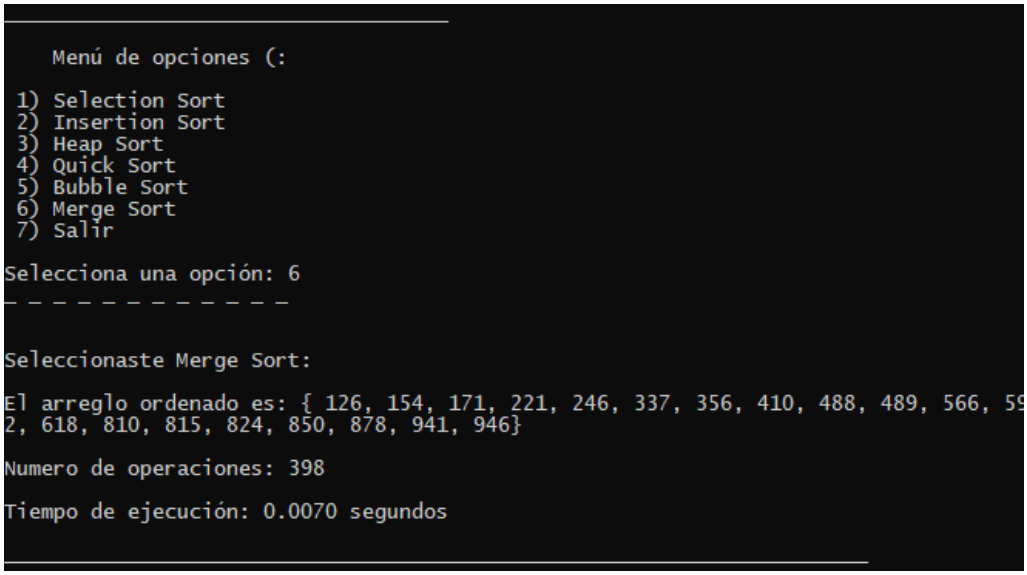
Ejercicio 3. Verificando el funcionamiento

Este programa tiene como función principal probar todos los algoritmos de ordenamiento vistos, para ello se retomó el archivo creado en la práctica 1 donde se realizó un menú de opciones para 3 algoritmos: "*Selection Sort*", "*Insertion Sort*" y "*Heap Sort*". Basado en ese archivo, se hicieron las modificaciones necesarias para agregar al menú de opciones los algoritmos "*Quick Sort*", "*Bubble Sort*" y "*Merge Sort*". Para probar que su implementación fue correcta, se generó un arreglo con 20 elementos y se seleccionó cada algoritmo en el menú de opciones.

```
_____________________________________________________________

       Algoritmos de ordenamiento (:

 ■ ¿Cuántos elementos quieres que tenga tu arreglo? 20

Se generó un arreglo aleatorio con 20 elementos: { 455, 534, 76, 921, 776, 722, 459
, 751, 80, 99, 71, 685, 556, 201, 857, 793, 453, 438, 634, 940}


_____________________________________

    Menú de opciones (:

 1) Selection Sort
 2) Insertion Sort
 3) Heap Sort
 4) Quick Sort
 5) Bubble Sort
 6) Merge Sort
 7) Salir

Selecciona una opción: 4
- - - - - - - - - - - -


Seleccionaste Quick Sort:

Pivote: 940
Sub array :  455 534 76 921 776 722 459 751 80 99 71 685 556 201 857 793 453 438 63
4

Pivote: 634
Sub array :  455 534 76 459 80 99 71 556 201 453 438

Pivote: 438
Sub array :  76 80 99 71 201

                                    ...
Sub array :  793
Sub array :  921
Sub array :

El arreglo ordenado es: { 71, 76, 80, 99, 201, 438, 453, 455, 459, 534, 556, 634, 6
85, 722, 751, 776, 793, 857, 921, 940}

Numero de operaciones: 181

Tiempo de ejecución: 0.0540 segundos
```

NOTA: Para que no fuera una captura muy larga, se hizo el recorte de algunas iteraciones.

```
─────────────────────────────────────────────────────────────────────

        Algoritmos de ordenamiento (:

 ■ ¿Cuántos elementos quieres que tenga tu arreglo? 20

Se generó un arreglo aleatorio con 20 elementos: { 478, 469, 53, 524, 442, 209, 478
, 28, 589, 598, 35, 994, 456, 953, 130, 88, 339, 343, 449, 772}

───────────────────────────────────

    Menú de opciones (:

 1) Selection Sort
 2) Insertion Sort
 3) Heap Sort
 4) Quick Sort
 5) Bubble Sort
 6) Merge Sort
 7) Salir

Selecciona una opción: 5
- - - - - - - - - - - -

Seleccionaste Bubble Sort:

Iteración: 1
{ 469, 478, 53, 524, 442, 209, 478, 28, 589, 598, 35, 994, 456, 953, 130, 88, 339,
343, 449, 772}
{ 469, 53, 478, 524, 442, 209, 478, 28, 589, 598, 35, 994, 456, 953, 130, 88, 339,
343, 449, 772}
{ 469, 53, 478, 524, 442, 209, 478, 28, 589, 598, 35, 994, 456, 953, 130, 88, 339,
343, 449, 772}
{ 469, 53, 478, 442, 524, 209, 478, 28, 589, 598, 35, 994, 456, 953, 130, 88, 339,
343, 449, 772}
{ 469, 53, 478, 442, 209, 524, 478, 28, 589, 598, 35, 994, 456, 953, 130, 88, 339,
343, 449, 772}
{ 469, 53, 478, 442, 209, 478, 524, 28, 589, 598, 35, 994, 456, 953, 130, 88, 339,
343, 449, 772}
{ 469, 53, 478, 442, 209, 478, 28, 524, 589, 598, 35, 994, 456, 953, 130, 88, 339,
343, 449, 772}
{ 469, 53, 478, 442, 209, 478, 28, 524, 589, 598, 35, 994, 456, 953, 130, 88, 339,
343, 449, 772}
{ 469, 53, 478, 442, 209, 478, 28, 524, 589, 598, 35, 994, 456, 953, 130, 88, 339

                                        ...

772, 953, 994}
{ 28, 35, 53, 88, 130, 209, 339, 343, 442, 449, 456, 469, 478, 478, 524, 589, 598,
772, 953, 994}
{ 28, 35, 53, 88, 130, 209, 339, 343, 442, 449, 456, 469, 478, 478, 524, 589, 598,
772, 953, 994}
{ 28, 35, 53, 88, 130, 209, 339, 343, 442, 449, 456, 469, 478, 478, 524, 589, 598,
772, 953, 994}
{ 28, 35, 53, 88, 130, 209, 339, 343, 442, 449, 456, 469, 478, 478, 524, 589, 598,
772, 953, 994}
{ 28, 35, 53, 88, 130, 209, 339, 343, 442, 449, 456, 469, 478, 478, 524, 589, 598,
772, 953, 994}
{ 28, 35, 53, 88, 130, 209, 339, 343, 442, 449, 456, 469, 478, 478, 524, 589, 598,
772, 953, 994}


  -> La lista ya esta ordenada en la iteración: 13 (:

El arreglo ordenado es: { 28, 35, 53, 88, 130, 209, 339, 343, 442, 449, 456, 469, 4
78, 478, 524, 589, 598, 772, 953, 994}

Numero de operaciones: 272

Tiempo de ejecución: 0.4030 segundos

─────────────────────────────────────────────────────────────────────

        Algoritmos de ordenamiento (:

 ■ ¿Cuántos elementos quieres que tenga tu arreglo? 20

Se generó un arreglo aleatorio con 20 elementos: { 488, 946, 878, 410, 618, 221, 81
5, 246, 356, 154, 592, 126, 337, 824, 171, 566, 489, 850, 941, 810}
```



./Ejercicio123/MenuOrdenamientos.c

Ejercicio 4. Complejidad Computacional

En este ejercicio se hicieron las modificaciones necesarias para que en cada una de las funciones se fueran contando cada operación de comparación, intercambio e inserción, con el fin de realizar el análisis de complejidad computacional de cada algoritmo. Se realizó el mismo procedimiento que en la práctica 1 para contar estas operaciones, en cada función se inicializó una variable en cero, la cual irá aumentando cada que se encuentre una operación y al final de la función se retornará el número total de operaciones, si la función es recursiva se sumará el valor de retorno con el valor que lleva el contador.

En este programa se presentó una pequeña dificultad al contar las operaciones de Quick Sort, ya que la función “partition” ya tenía un valor de retorno, así que para solucionar esto se utilizó la función en paso por referencia a través de apuntadores. El resultado de las funciones recursivas, a diferencia de las demás, se imprimieron en el menú de opciones con el fin de que solo se mostrará una vez el valor del total de operaciones.

Finalmente, al igual que en la práctica 1, se hicieron pruebas con cada algoritmo de ordenamiento con arreglos de diferente tamaño: 50, 100, 200, 500, 1000, 2000. Se hicieron 5 ejecuciones por cada tamaño y se realizó una tabla general con el promedio del número de operaciones de cada uno. (Los datos obtenidos se encuentran en el archivo “*Tablas.xlsx*” en la carpeta “*Ejercicio123*”).

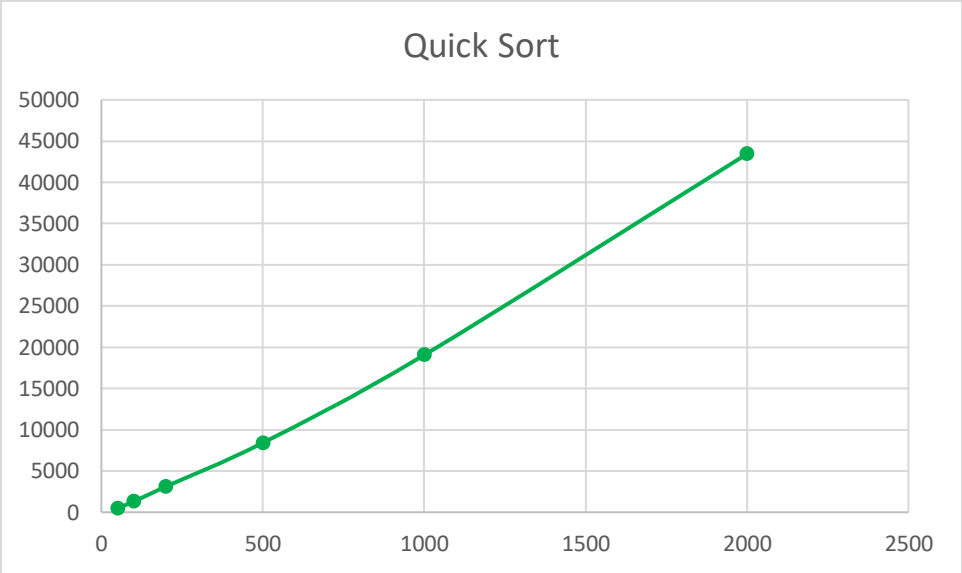
A continuación, se muestran las tablas en general, al igual que las gráficas de cada algoritmo.

Quick Sort

Complejidad de este algoritmo en el mejor casos y caso promedio es de $O(n\log(n))$, mientras que para el peor de los casos es de $O(n^2)$.

QUICK SORT	
Valor	Operaciones
50	506.2
100	1293.4
200	3120.6
500	8418.8
1000	19062.8
2000	43438.4

Tabla Quick-Sort



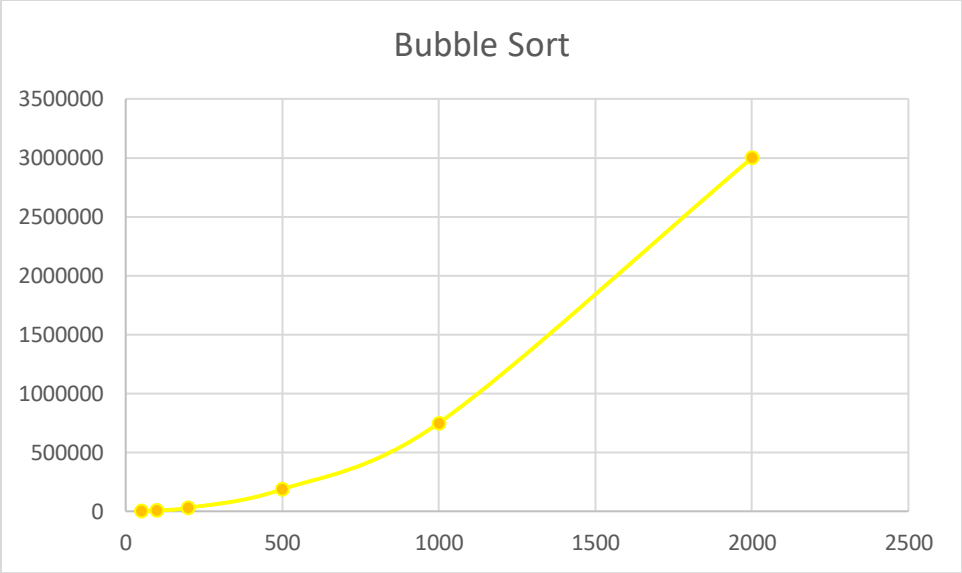
Gráfica Quick-Sort

Bubble Sort

Complejidad de este algoritmo en el mejor caso, caso promedio y el peor caso es de $O(n^2)$.

BUBBLE SORT	
Valor	Operaciones
50	1855.4
100	7367
200	29590
500	187050.6
1000	748665.4
2000	2998699.4

Tabla Bubble-Sort



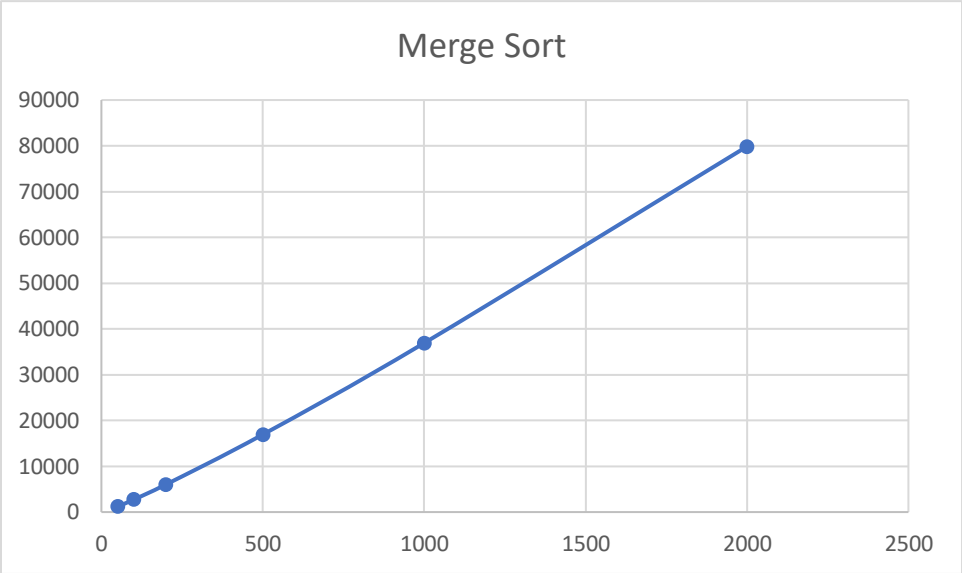
Gráfica Bubble-Sort

Merge Sort

Complejidad de este algoritmo para el mejor caso, caso promedio y el peor de los casos una complejidad de $O(n\log(n))$.

MERGE SORT	
Valor	Operaciones
50	1202
100	2710
200	6026
500	16938
1000	36922
2000	79850

Tabla Merge-Sort



Gráfica Merge-Sort

Ejercicio 5. Quicksort en Java y practicando la orientación a objetos

Se creó un nuevo proyecto en NetBeans el cual tiene como función principal implementar el algoritmo de Quick-Sort, así como probar su funcionamiento. Para la implementación de este programa se creó una clase llamada "*QuickSort*" la cual tiene los siguientes 2 métodos:

- Partition

Tiene como función dividir un arreglo en varios sub-arreglos.

- QuickSort

Es el método es donde se hace todo el procedimiento de ordenamiento para un determinado arreglo, dentro de este método se llama a su vez al método "partition".

También se creó una clase llamada "utilerías" en la cual se tendrán los siguientes 3 métodos:

- printArray

Tiene como función imprimir el arreglo completo. Recibe como parámetro el arreglo, a diferencia de su implementación en C, este método obtiene el número de elementos que tiene el arreglo con el método "*length*".

- Swap

Tiene como función intercambiar dos elementos en un arreglo. Tiene como parámetros los elementos que se quieren cambiar y el arreglo.

- printSubArray

Tiene como función imprimir los sub-arreglos que se forman al dividir un arreglo completo.

En el método principal "*main*" ubicado en la clase "*Practica2CarrilloIvette*" se inicializó un arreglo de tamaño "n" (número que el usuario indique, llamando a la biblioteca Scanner para que el usuario pueda introducir el número), se llamó a la biblioteca "*Random*" para generar números aleatorios del 0 al 500 y mediante un ciclo for ir guardando estos números en el arreglo. Después, se llamo al método constructor QuickSort pasando como parámetro el arreglo, así como el primer y último elemento de este. Cabe destacar que, este método se modifico para que aparecieran los sub-arreglos, así como el pivote de cada iteración; esto se logró agregando el método "*printSubArray*" en la clase utilerías.

Comprobamos su funcionamiento, ingresando 20 elementos en el arreglo.

```
_____________________________________________________________
                        Quick Sort
Ingresa el número de elementos que quieres que tenga tu arreglo: 20

Se generó el siguiente arreglo aleatorio:
{ 299, 74, 168, 157, 164, 301, 19, 16, 229, 377, 56, 410, 44, 480, 260, 444, 482, 307, 316, 434}
- - - - - - - - - - - - - - - - - - - - - -

Pivote: 434
Sub array :  299 74 168 157 164 301 19 16 229 377 56 410 44 260 307 316

Pivote: 316
Sub array :  299 74 168 157 164 301 19 16 229 56 44 260 307

Pivote: 307
Sub array :  299 74 168 157 164 301 19 16 229 56 44 260

Pivote: 260
Sub array :  74 168 157 164 19 16 229 56 44

Pivote: 44
Sub array :  19 16

Pivote: 16
Sub array :
Sub array :  19
Sub array :  164 74 168 229 56 157

Pivote: 157
Sub array :  74 56

Pivote: 56
Sub array :
Sub array :  74
Sub array :  229 164 168

Pivote: 168
Sub array :  164
Sub array :  229
Sub array :  299 301

Pivote: 301
Sub array :  299
Sub array :
Sub array :
Sub array :  377 410

Pivote: 410
Sub array :  377
Sub array :
Sub array :  480 444 482

Pivote: 482
Sub array :  480 444
```

```
Pivote: 444
Sub array :
Sub array :  480
Sub array :
- - - - - - - - - - - - - - - - - - - - - - - -

El arreglo ordenado con QuickSort es:
{ 16, 19, 44, 56, 74, 157, 164, 168, 229, 260, 299, 301, 307, 316, 377, 410, 434, 444, 480, 482}

BUILD SUCCESSFUL (total time: 4 seconds)
```

./Practica2[CarrilloIvette]/Practica2CarrilloIvette.java

Conclusiones

Se cumplieron con los objetivos de esta práctica, ya que identificamos la estructura de 3 algoritmos de ordenamientos nuevos: “Bubble Sort”, “Quick Sort” y “Merge Sort”; así como se realizaros varias pruebas con ellos con la finalidad de observar la complejidad computacional de cada algoritmo, se observó esta complejidad a través de gráficas, (el número de operaciones con respecto al número al número de elementos que contiene el arreglo). Se pudo observar en estas dos prácticas los diferentes tipos de algoritmos de ordenamiento y se llegó a la conclusión que los más eficientes son “*Heap Sort*” y “*Merge Sort*”

También se observó que hay diferentes formas de implementar el algoritmo QuickSort, y que no importa donde se encuentre el “*pivote*” seguirá funcionando el algoritmo de igual manera.

Se cumplieron con todas las actividades de esta práctica, considero que estas primeras dos prácticas tuvieron los ejercicios correspondientes a el tema de Algoritmos de Ordenamiento. Además, se realizó un ejercicio en el lenguaje Java y se observó un poco más de la Programación Orientada a Objetos. (: