



## Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

# Laboratorios de computación salas A y B

*Profesor:* Tista García Edgar Ing.

*Asignatura:* Estructura de Datos y Algoritmos II

*Grupo:* 5

*No de Práctica(s):* 2

*Integrante(s):* Téllez González Jorge Luis

*No. de Equipo de  
cómputo empleado:* 37

*No. de Lista o Brigada:* 37

*Semestre:* 2020-2

*Fecha de entrega:* 16/02/2020

*Observaciones:*

**CALIFICACIÓN:** \_\_\_\_\_



# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Objetivos</b>	<b>3</b>
<b>3. Análisis inicial de Quick-Sort, Bubble-Sort y Heap-Sort</b>	<b>4</b>
3.1. La implementación de Quick-Sort . . . . .	4
3.2. Optimización de Bubble-Sort . . . . .	5
3.3. Funcionamiento de Heapify . . . . .	5
3.4. Pruebas de ejecución . . . . .	6
<b>4. Complejidad computacional</b>	<b>9</b>
4.1. Resultados obtenidos . . . . .	9
4.2. Análisis de resultados . . . . .	11
<b>5. Implementación de Quick-Sort en Java</b>	<b>12</b>
5.1. Análisis del programa . . . . .	13
5.2. Pruebas de ejecución . . . . .	14
<b>6. Conclusiones</b>	<b>14</b>

# 1. Introducción

Los diferentes algoritmos de búsqueda existentes ofrecen diversas bondades así como notables desventajas; como puede ser su complejidad computacional en determinados casos. En diversas aplicaciones del mundo real se busca que un algoritmo sea computacionalmente eficiente y posea un tiempo de ejecución lo más corto posible.

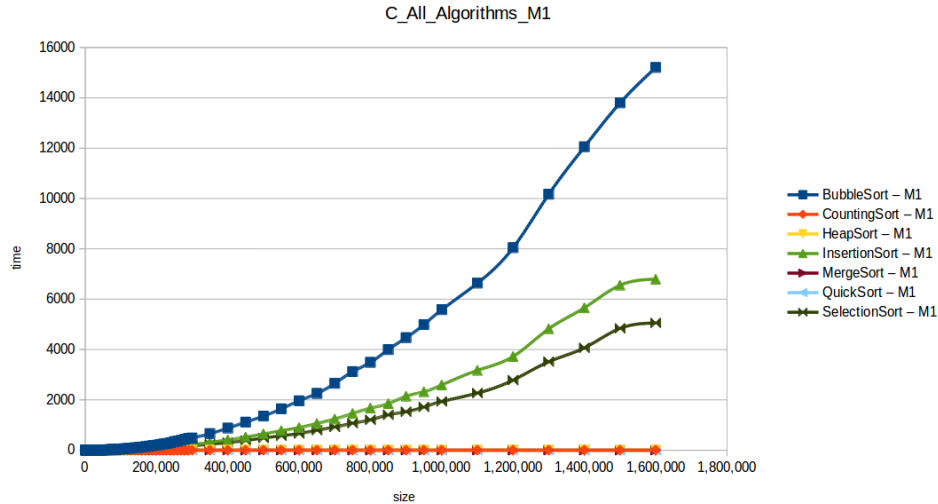


Figura 1: *Tiempo de ejecución de diversos algoritmos de ordenamiento.*

Existen diversas técnicas de diseño de algoritmos con el fin de solucionar problemas de la manera más eficiente posible. Una de estas técnicas se conoce como *Divide y Vencerás*: este paradigma separa un problema en subproblemas que se parecen al problema original, de manera recursiva resuelve los subproblemas y, por último, combina las soluciones de los subproblemas para resolver el problema original. Como *Divide y Vencerás* resuelve subproblemas de manera recursiva, cada subproblema debe ser más pequeño que el problema original. Así mismo, debe de existir un caso base para todos los subproblemas. [1]

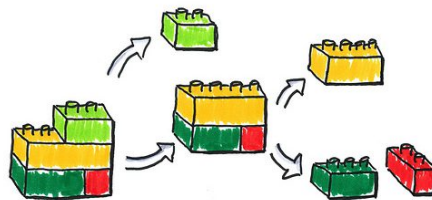


Figura 2: *La suma de las sub-soluciones conduce a la solución final.*

Otros algoritmos pueden optar por soluciones diferentes y únicas, como el caso de Heap-Sort: el cual basa su funcionamiento en la creación de un *Heap* y en la extracción iterativa de los nodos raíz hasta

obtener un arreglo ordenado.

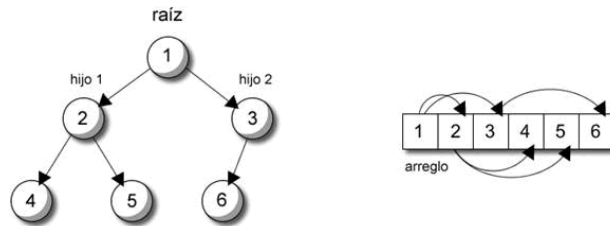


Figura 3: Ordenamiento Heap-Sort o por montículos.

Estos algoritmos de ordenamiento pueden ser mucho más eficientes que otros, a costa de tener un complejo funcionamiento. Otros, en cambio, pueden ser intuitivos y fáciles de comprender como el caso particular de Bubble-Sort. Su sencillez, lamentablemente, viene acompañada de una elevada complejidad computacional que llega a  $O(n^2)$  en el peor caso.

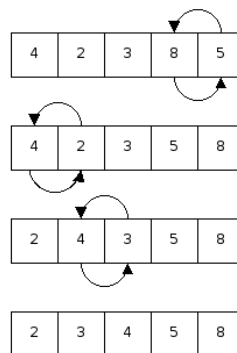


Figura 4: Ordenar arreglos utilizando Bubble-Sort tiene un alto precio.

En este reporte escrito se analizarán tres algoritmos de ordenamiento interno: Quick-Sort, Bubble-Sort y Heap-Sort. Se analizará su funcionamiento y características, se pondrán a prueba y finalmente será verificada su respectiva complejidad computacional. Así mismo, se analizará una implementación de Quick-Sort en el lenguaje Java y se realizará un análisis de su estructura para finalmente realizar su respectiva prueba de ejecución.

## 2. Objetivos

- El estudiante identificará la estructura de los algoritmos de ordenamiento BubbleSort, QuickSort y HeapSort.
- 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.

- Verificar la complejidad computacional de *Bubble-Sort* , *Heap-Sort* y *Quick-Sort* por medio de la cantidad de operaciones realizadas.

### 3. Análisis inicial de Quick-Sort, Bubble-Sort y Heap-Sort

La biblioteca *ordenamientos.h* que había sido utilizada en la anterior práctica se ha visto enriquecida con tres implementaciones de Quick-Sort, Bubble-Sort y Heap-Sort; cada una con sus respectivas funciones auxiliares. A continuación, serán analizados aspectos importantes de cada uno de estos algoritmos de ordenamiento.

#### 3.1. La implementación de Quick-Sort

La implementación brindada de Quick-Sort trabaja haciendo uso de llamadas recursivas y de una función denominada **partition** que se encarga de realizar la partición del arreglo original en sub-arreglos. Una diferencia clave entre el enfoque teórico que fue estudiado en clase y el que se presenta reside en que, a diferencia de la convención de tomar el pivote como el primer elemento a la izquierda del arreglo o de tomar el pivote como el promedio de los índices del primer y último elemento, esta implementación opta por tomar como el pivote al último elemento del arreglo que recibe como entrada.

```
41 | int partition(int arr[], int low, int high){  
42 |     int pivot = arr[high];  
43 |     printf("Pivote: %d\n", pivot);
```

Figura 5: Asignación del pivote en la implementación de C.

Por otra parte, la forma en que se encuentran asignadas las variables de control utilizadas para recorrer el arreglo difiere notablemente al enfoque teórico, pues en vez de asignar a *i* y *j* el índice del primer y último elemento respectivamente, se opta por asignar a ambas variables el valor anterior al índice más bajo de tal forma que ambos índices comparan valores de izquierda a derecha de forma simultánea.

```
int j, i = (low - 1);  
for(j=low; j<=high-1; j++){  
    if(arr[j] <= pivot){  
        i++;  
        swap(&arr[i], &arr[j]);  
    }  
}
```

Figura 6: Operación de comparación e intercambio de Quick-Sort.

### 3.2. Optimización de Bubble-Sort

Bubble-Sort es uno de los algoritmos de ordenamiento más simples e intuitivos que existen, sin embargo, posee una importante falla: el número de comparaciones que realiza es excesivo y no se detiene. Es decir, si Bubble-Sort recibe un arreglo de elementos prácticamente ordenado, como resultado se tendrán múltiples comparaciones para revisar si el arreglo está ordenado, incluso cuando efectivamente lo está.

Esta notable falla del algoritmo puede ser arreglada añadiendo una variable pseudo-booleana con valor falso **0** y un valor verdadero **1** junto con una condicional que detenga la ejecución de la función cuando la variable *booleana* no modifique su valor a **1** tras una iteración.

```
for (i=n-1; i>0; i--) {
    int contadorIter=0;
    printf("\n---Revision %i---\n\n", revision);
    revision=revision+1;
    for(j=0; j<i; j++){

        if(a[j]>a[j+1]){
            swap(&a[j], &a[j+1]);
            contadorIter=1;
        }
    }
    printf("\nIteracion %i:\n\n", j);
    printArray(a, n);
}
if(contadorIter==0){
    printf("Se han realizado todos los intercambios necesarios.\n");
    return;
}
```

Figura 7: Condición de parada de Bubble-Sort.

La *optimización* realizada evitará que la función continúe realizando comparaciones cuando se detecte que el arreglo ya ha sido ordenado, verificando que no se han realizado nuevas operaciones de intercambio tras un proceso iterativo en el índice *j*.

### 3.3. Funcionamiento de Heapify

Heap-Sort necesita para su ejecución una función especial denominada **Heapify**. Esta función recibe un arreglo y un entero. Este entero será utilizado para asignar las posiciones iniciales del Heap que siguen un orden de izquierda a derecha.

A continuación se verifica si la posición *l* existe en el arreglo y si el valor en la posición *l* es mayor al valor en la posición *i* recibida como parámetro. Si esta condición es verdadera, se le asigna a la variable *largest* el valor de *l*. Caso contrario, se le asigna a *largest* el valor de *i*.

La función continúa verificando si existe una posición *r* en el arreglo. Si el valor en la posición *r* resulta mayor al valor presente en el índice *largest*, se le asigna a la variable *largest* el valor de *r*.

```
97     int l = 2 * i + 1;
98     int r = 2 * i + 2;
99     int largest;
100
101     if(l <= heapSize && A[l] > A[i])
102         largest = l;
103     else
104         largest = i;
105     if(r <= heapSize && A[r] > A[largest])
106         largest = r;
```

Figura 8: Verificaciones de integridad del Heap.

Tras haber hecho las comparaciones anteriores se verifica si el valor de la variable *largest* es diferente al entero *i* recibido como parámetro. En caso afirmativo, intercambiará los valores de las posiciones del arreglo *largest* e *i*, imprimirá el estado del arreglo y realizará de nuevo el proceso con *largest* como nuevo parámetro.

```
if(largest != i){
    swap(&A[i], &A[largest]);
    printArray(A, size);
    Heapify(A, largest, size);
}
```

Figura 9: Proceso de intercambio, impresion y llamada recursiva.

### 3.4. Pruebas de ejecución

El programa propuesto para poner a prueba los algoritmos sigue la misma línea con respecto al realizado en la práctica anterior: se tiene un menú de selección que permite elegir entre los 3 algoritmos de ordenamiento disponibles. Tras seleccionar uno, se generará un arreglo aleatorio de 10 elementos y se ordenará el arreglo, mostrando cada paso que realiza con el fin de lograr su objetivo.

A continuación se mostrarán los resultados de la ejecución de los tres algoritmos utilizando un arreglo aleatorio con 10 elementos:

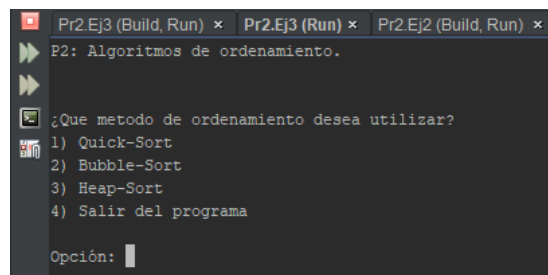


Figura 10: Menú de selección.

```
Usted ha seleccionado Quick-Sort:

El arreglo se está generando...

El arreglo tiene como entradas iniciales:

11 19 44 30 20 31 35 56 98 93

Ahora, ordenando el arreglo...

Pivote: 0
Sub array :
Sub array : 44 30 20 31 35 56 98 93 19
Pivote: 19
Sub array :
Sub array : 30 20 31 35 56 98 93 44
Pivote: 44
```

Figura 11: *Proceso inicial de Quick-Sort.*

```
Pivote: 98
Sub array : 93
Sub array :

¡Felicidades! Su arreglo está ordenado. :)

Sub array : 0 19 20 30 31 35 44 56 93 98
```

Figura 12: *Arreglo ordenado obtenido.*

```
Opción: 2

Usted ha seleccionado Bubble-Sort:

El arreglo se está generando...

El arreglo tiene como entradas iniciales:

41 70 77 88 47 35 87 99 54 72

Ahora, ordenando el arreglo...

---Revision 1---

Iteracion 0:

41 70 77 88 47 35 87 99 54 72

Iteracion 1:

41 70 77 88 47 35 87 99 54 72
```

Figura 13: *Selección de Bubble-Sort.*



```

---Revision 6---

Iteracion 0:

35 41 47 54 70 72 77 87 88 99

Iteracion 1:

35 41 47 54 70 72 77 87 88 99

Iteracion 2:

35 41 47 54 70 72 77 87 88 99

Iteracion 3:

35 41 47 54 70 72 77 87 88 99
Se han realizado todos los intercambios necesarios.

¡Felicidades! Su arreglo está ordenado. :)

35 41 47 54 70 72 77 87 88 99

```

Figura 14: Arreglo ordenado obtenido con Bubble-Sort.

```

Usted ha seleccionado Heap-Sort:

El arreglo se está generando...

El arreglo tiene como entradas iniciales:

53 4 34 29 17 39 48 35 59 86

Ahora, ordenando el arreglo...

53 4 34 29 86 39 48 35 59 17
53 4 34 59 86 39 48 35 29 17
53 4 48 59 86 39 34 35 29 17
53 86 48 59 4 39 34 35 29 17
53 86 48 59 17 39 34 35 29 4
86 53 48 59 17 39 34 35 29 4
86 59 48 53 17 39 34 35 29 4
Terminó de construir el HEAP
Iteracion HS:
4 59 48 53 17 39 34 35 29 86
59 4 48 53 17 39 34 35 29 86
59 53 48 4 17 39 34 35 29 86
59 53 48 35 17 39 34 4 29 86

```

Figura 15: Proceso de construcción del Heap.

```

Iteracion HS:
4 29 17 34 35 39 48 53 59 86
29 4 17 34 35 39 48 53 59 86
Iteracion HS:
17 4 29 34 35 39 48 53 59 86
Iteracion HS:
4 17 29 34 35 39 48 53 59 86

¡Felicidades! Su arreglo está ordenado. :)

4 17 29 34 35 39 48 53 59 86

```

Figura 16: Ordenamiento por extracción de raíces de Heap-Sort.

## 4. Complejidad computacional

Con el fin de verificar la complejidad computacional de los tres algoritmos se ha optado por incluir dos variables con el fin de observar el comportamiento de ambos algoritmos de ordenamiento frente a una cantidad determinada de entradas. La metodología seguida para realizar tal proceso es **idéntica** a la descrita en la práctica anterior; enviando dos variables llamadas *comparaciones* e *intercambios* que almacenan el total de operaciones de cada tipo realizadas durante un proceso de ordenamiento.

```
¡Felicidades! Su arreglo está ordenado. :)
Sub array : 19 42 49 63 63 64 66 75 83 96 98
Número total de comparaciones: 84
Número total de intercambios: 23
```

Figura 17: Cantidad total de operaciones realizadas por Quick-Sort con un arreglo de 10 elementos.

A continuación se han probado las modificaciones hechas al código fuente para verificar el comportamiento de los tres algoritmos con una entrada de 50, 100 y 500 números modificando la constante 'N' definida en el código fuente del programa. El programa arrojará el número y el tipo de operación realizada al final de la ejecución del algoritmo de ordenamiento, como puede verse en [17].

Con el fin de mostrar gráficamente los datos obtenidos se ha recurrido al uso de tablas y gráficas, las cuales fueron realizadas por medio del software Microsoft Excel© 2016.

### 4.1. Resultados obtenidos

Tras haber puesto a prueba los 3 algoritmos con 3 entradas diferentes, se han obtenido los siguientes resultados:

	Quick-Sort		Bubble-Sort		Heap-Sort	
Entrada	Comparaciones I	Intercambios I	Comparaciones J	Intercambios J	Comparaciones K	Intercambios K
50	691	163	2538	616	892	247
100	1624	416	9761	2495	2017	572
500	11875	3664	248193	60462	13603	4034

Figura 18: Cantidad total de operaciones realizadas por los 3 algoritmos de ordenamiento.

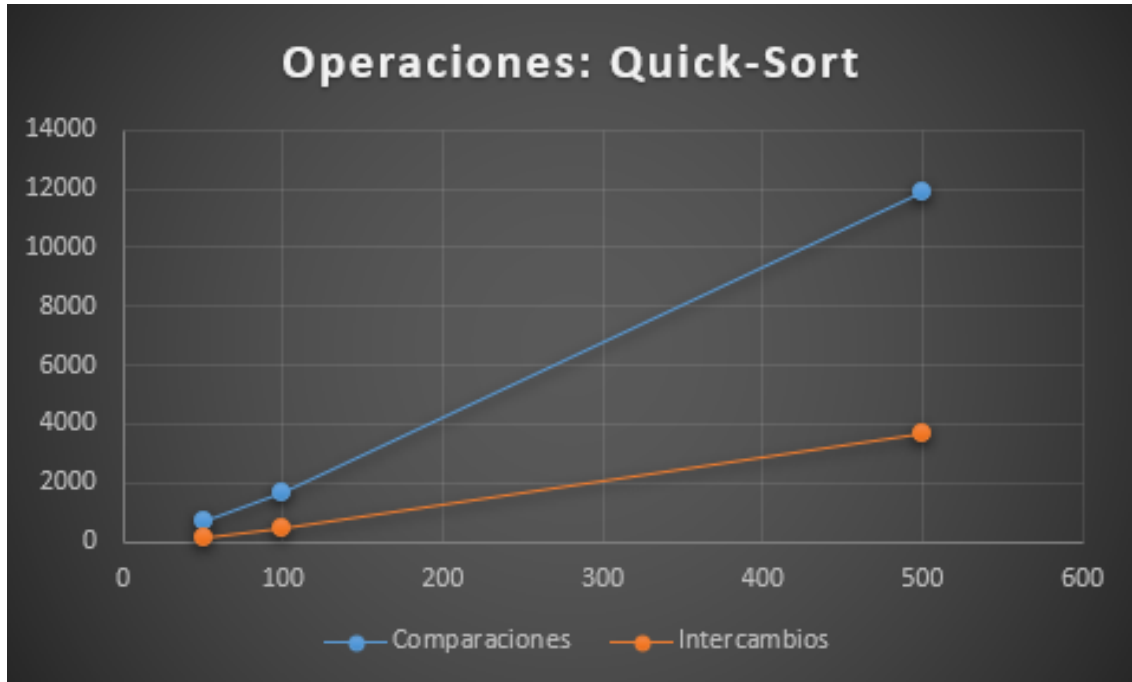


Figura 19: Cantidad total de operaciones realizadas por Quick-Sort.

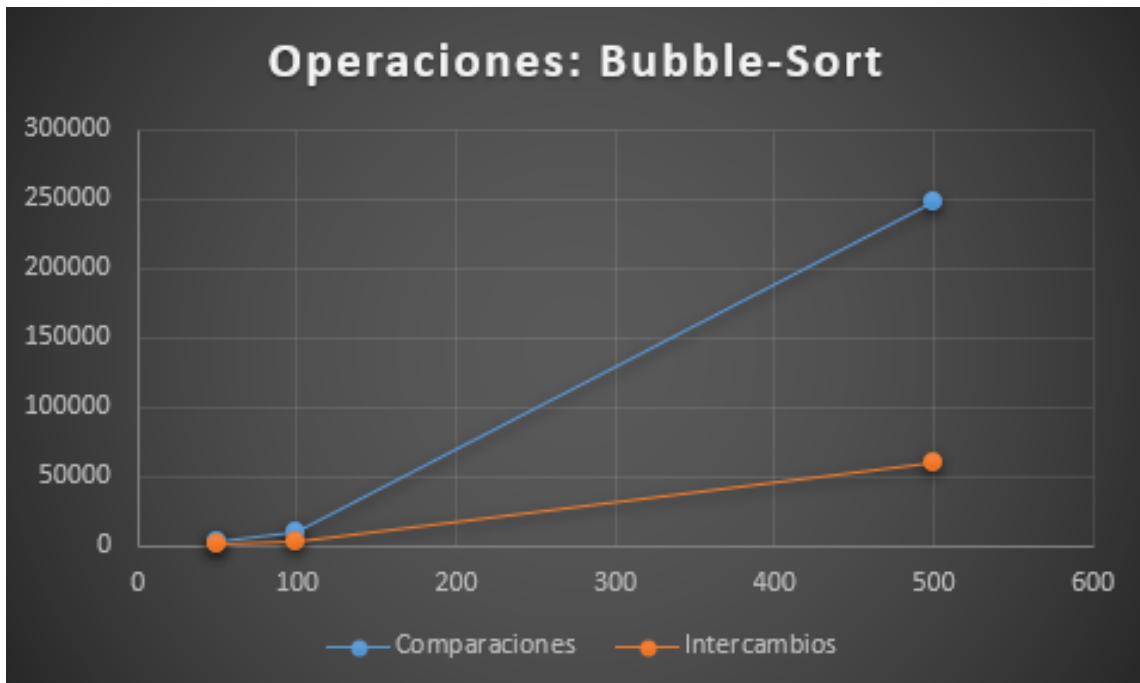


Figura 20: Cantidad total de operaciones realizadas por Bubble-Sort.

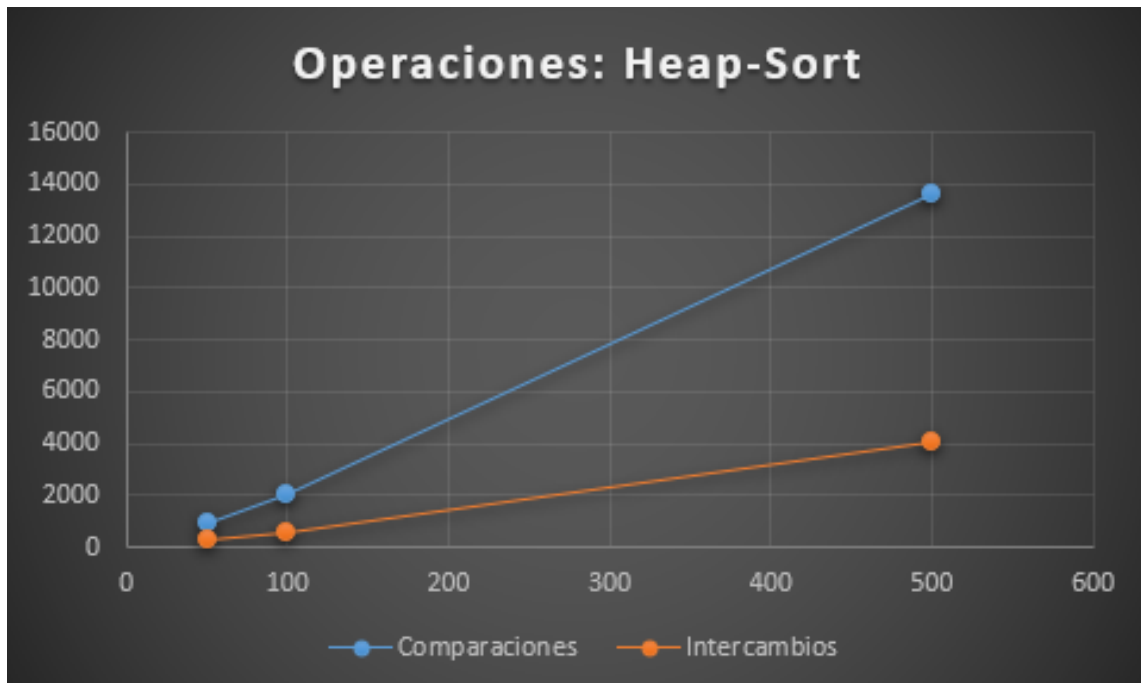


Figura 21: Cantidad total de operaciones realizadas por Heap-Sort.

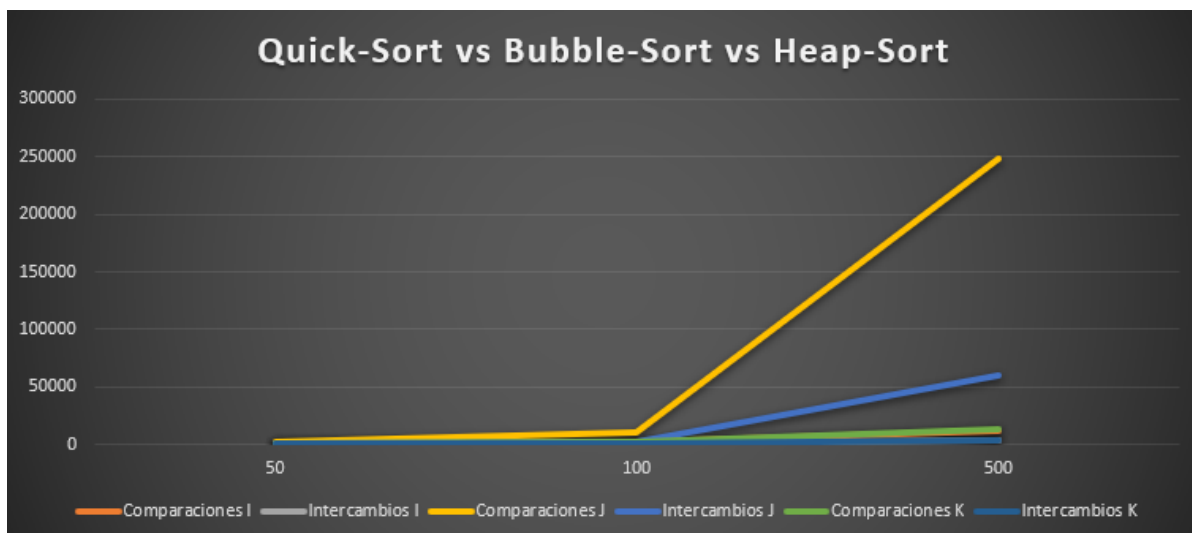


Figura 22: Cantidad total de operaciones realizadas por los 3 algoritmos de ordenamiento.

## 4.2. Análisis de resultados

Es importante destacar los siguientes puntos con respecto a los resultados obtenidos por los 3 algoritmos:

- Quick-Sort y Heap-Sort obtuvieron un número similar de operaciones de comparación e intercambio

entre cada prueba, sin embargo, Quick-Sort fue mucho más rápido en su ejecución y requirió un número de operaciones ligeramente menor para cumplir su objetivo.

- Bubble-Sort demostró ser tremendamente ineficiente conforme se aumentaba la entrada de datos. A diferencia de los otros algoritmos, Bubble-Sort requirió un tiempo mucho más elevado para lograr su objetivo así como un número total de operaciones mucho mayor.

```
¡Felicidades! Su arreglo está ordenado. :)
0 1 2 2 3 3 5 5 6 8 11 11 12 13 13 14 15 15 17 17
3 63 63 65 65 66 67 68 68 68 70 71 72 75 77 77 79
Número total de comparaciones: 10198
Número total de intercambios: 2466
```

Figura 23: La complejidad computacional  $O(n^2)$  de Bubble-Sort se hace presente.

- Quick-Sort fue el más veloz de los 3 algoritmos de ordenamiento. Heap-Sort se quedó un poco atrás, sin embargo, también logró su objetivo en un tiempo de espera razonable. En cambio, Bubble-Sort requirió una cantidad demasiado elevada de tiempo para lograr su objetivo con respecto a los 2 algoritmos de complejidad  $O(n * \log(n))$ .

## 5. Implementación de Quick-Sort en Java

La siguiente actividad consistió en crear un nuevo proyecto en **NetBeans** denominado *Practica2Téllez* y crear una nueva clase denominada *Quicksort* con la implementación del algoritmo en Java.

```
package practica2téllez;

/**
 *
 * @author jorje
 */
public class Quicksort {

    int partition(int arr[], int low, int high){
        int pivot=arr[high];
        int i=(low-1);
        for(int j=low; j<high; j++){
```

Figura 24: Clase *Quicksort* implementada en Java.

Así mismo, se creó otra clase denominada *Utilidades* con un método llamado **printArray**. Además, se incluyó su respectivo procedimiento que tiene como objetivo imprimir el estado actual de un arreglo recibido como parámetro.

```
5 public class Utilidades {  
6  
7     static void printArray(int arr[]){  
8         int n = arr.length;  
9         for(int i=0;i<n;i++){  
10             System.out.print(arr[i]+" ");  
11             System.out.println();  
12     }  
13 }
```

Figura 25: Clase *Utilidades* en Java.

## 5.1. Análisis del programa

- Al momento de crear un proyecto de Java en *NetBeans* es generado un paquete con el nombre del proyecto escrito completamente en minúsculas: este paquete es el encargado de almacenar las clases que son parte de un programa escrito en este lenguaje.

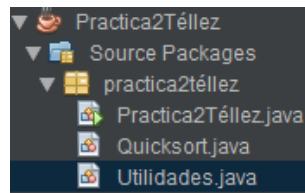


Figura 26: Clases pertenecientes al proyecto.

- La clase *Quicksort* no tiene registrado ningún atributo, y en cambio, entre sus métodos se encuentra **partition** y **sort**. Una diferencia existente con respecto a la implementación de C es que, en vez de utilizar una función auxiliar de intercambio, el método **partition** realiza el proceso de intercambio de índices en sí mismo haciendo uso de un variable temporal.

```
for(int j=low; j<high; j++){  
    if(arr[j] <=pivot){  
        i++;  
        int temp=arr[i];  
        arr[i]=arr[j];  
        arr[j]=temp;  
    }  
}
```

Figura 27: Proceso de intercambio interno.

La clase posee un modificador de acceso *public*, lo cual implica que todo lo que se encuentra dentro de esa clase (atributos y métodos) puede ser utilizados por clases externas pertenecientes al mismo paquete, como puede observarse en [24].

- El método **printArray** presenta ciertas diferencias con respecto a su homólogo en C: no es necesario que se reciba como parámetro el tamaño de un arreglo puesto que el arreglo ya posee como atributo interno su tamaño. Para extraerlo, se utiliza el operador unario '.' y el valor se guarda en una variable entera *n*; como puede observarse en [25]. Este método no devuelve nada y es de tipo estático.

- La forma de imprimir cadenas de caracteres es diferente: para imprimir cadenas en Java se utiliza la clase para acceder a su método *print*; enviando como parámetro las variables o las cadenas solicitadas. Un detalle importante es que Java admite la *concatenación* de cadenas de caracteres y variables por medio del operador `'+'`.

## 5.2. Pruebas de ejecución

Finalmente, se escribió en el método `main` de la clase principal un arreglo estático y se puso a prueba la implementación del algoritmo mediante la creación de un objeto de tipo *QuickSort*.

```
9 public class Practica2Téllez {
10
11     public static void main(String[] args) {
12
13         int arr[]={87,4,32,15,8,12,10,30,22};
14         int n=arr.length;
15         Quicksort ordenamiento= new Quicksort();
16         ordenamiento.sort(arr, 0, n-1);
17         Utilidades.printArray(arr);
18     }
```

Figura 28: Creación del objeto *Quicksort*.

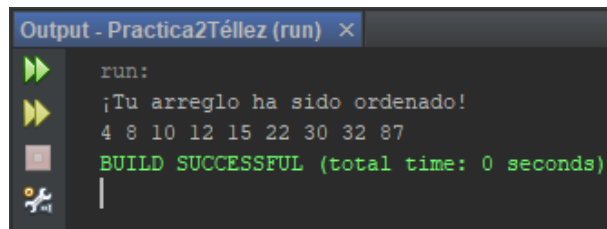


Figura 29: Resultado obtenido tras ejecutar el programa.

## 6. Conclusiones

El análisis y trabajo realizado con los 3 algoritmos de ordenamiento propuestos ha resultado en una experiencia enriquecedora y que complementa de forma notable al desarrollo de la primera práctica.

A raíz del trabajo realizado, deseo abordar los siguientes puntos:

1. Considero que el flujo de trabajo seguido en esta práctica ha mejorado con respecto a la práctica anterior, sin embargo, todavía hacen falta muchos detalles por afinar en cuanto a la velocidad de trabajo y análisis.



2. Se verificó experimentalmente la gran eficiencia que tiene tanto Quick-Sort como Heap-Sort en situaciones reales. Así mismo, se pudo observar el porqué Bubble-Sort es un algoritmo muy ineficiente y que suele estar relegado a un uso meramente educacional.
3. Los primeros pasos en el uso aplicado de Java en el análisis de algoritmos, como en la práctica anterior, es una gran experiencia que complementa las lecciones impartidas en la asignatura de *Programación Orientada a Objetos* y que permite observar la estrecha relación que ambas asignaturas guardan entre sí.

Los enfoques de las implementaciones de C fueron interesantes y retadores de comprender; como el caso de Quick-Sort que marcó una diferencia notable con respecto al enfoque teórico de la clase. Así mismo, el análisis de una de las funciones de Heap-Sort, **Heapify**, fue muy complicado en un inicio. Por otra parte, la solución al problema de Bubble-Sort (que fue planteado durante la clase) fue implementada en el código de la implementación proporcionada de forma exitosa.

Considerando que todas las actividades propuestas han sido cumplidas de forma exitosa, incluyendo los análisis propuestos, las pruebas de ejecución y las pruebas de complejidad, es posible concluir que los objetivos propuestos al inicio de este reporte han sido cumplidos con éxito.

## Referencias

- [1] Algoritmos de divide y vencerás. Recuperado de: <https://es.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms>. Fecha de consulta: 15/02/2020.
- [2] Vista general del ordenamiento rápido. Recuperado de: <https://es.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort>. Fecha de consulta: 15/02/2020.

Los créditos de las fotografías pertenecen a sus respectivos autores. ©