	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): Edgar Tista García

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 3

No de Práctica(s): 2

Integrante(s): Román Ramos María Fernanda

No. de lista o brigada:

Semestre: 2026-1

Fecha de entrega: 08/09/2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Objetivos

- **Objetivo general:** El estudiante identificará la estructura de los algoritmos de ordenamiento HeapSort y QuickSort
- **Objetivo de la clase:** El estudiante observará la importancia del orden de complejidad aplicado en algoritmos de ordenamiento, conocerá diferentes formas de implementar Quicksort y comenzará a realizar programas sencillos en el lenguaje Java

  **Nota: Para el desarrollo de la práctica se usarán las herramientas:*

  - *Visual Studio Code: para crear e implementar los programas en C y Java.*
  - *Overleaf: para crear este documento en LaTeX del reporte.*

2. Desarrollo

2.1. Ejercicio 1.Agregando Ordenamientos

Agregué las funciones:

- quickSort: en ordenamientos
- printSubArray: en utilerias
- partition: en utilerias
- heapSort: en ordenamientos
- BuildHeap: en utilerias
- Heapify: en utilerias

```
//Funciones ed Quick
void printSubArray(int arr[],int low, int high);
int partition (int arr[], int low, int high);
//funciones de Heap:
void Heapify(int* A, int i, int size);
void BuildHeap(int* A, int size);
```

```
26   //FUNCIONES NECESRIAS PARA QUICKSORT
27 > void printSubArray(int arr[],int low, int high){···
34 > int partition (int arr[], int low, int high){···
60
61   // FUNCIONES PARA HEAP
62 > void BuildHeap(int* A, int size){···
76 > void Heapify(int* A, int i, int size){···
```

Declarar las funciones que necesita cada ordenamiento en utilerias.h y definirlas en utilerias.c

```
//Nuevos:
void quickSort(int arr[], int low, int high);
void HeapSort(int* A, int size);
```

```
> void quickSort(int arr[], int low, int high){···

> void HeapSort(int* A, int size){···
```

Declarar los algoritmos Quisk y Heap en ordenamientos.h y definirlos en ordenamientos.c

1. **Análisis general de los nuevos algoritmos proporcionados** (diferencias con respecto a los algoritmos vistos en la práctica 1)

   - **Quick Sort:** Este algoritmo tiene 2 etapas en cada llamada: Después de elegir el pivote, que comunmente, es el primer o último elemento de la lista, en el caso de esta practica es el último, se realiza lo siguiente:

     a) **Vericifar el pivote** (partition): compararlo con el resto de la lista, para este criterio que es ascendente, todos los menores quedan a su izquierda, y los mayores a la derecha, pero al final, después de comparar con todos, el pivote queda en su posición final.

     b) **Ordenar las sublistas**: Después de comparar al pivote con todos, se trabaja con las subistas que se generan a los lados (se llama recursivamente a *quickSort*).

     Es un algoritmo con complejidad de:
     En el peor caso promedio y mejor caso: $O(nlog(n))$, esto se debe a que en cada iteración trabaja recursivamente con fracciones cada vez más pequeñas de la lista (las sublistas), hasta que llega a su caso base, cuando la lista de tamaño 1, y el pivote está en su posición final.

En el peor caso: O(n^2), ya que se tendrían que hacer TODOS los intercambios posibles.

- **Heap Sort:**
  Este algoritmo tiene 2 etapas:

  a) **Construir el Heap** (BuildHeap): en el caso del código de esta práctica, se usa la estrategia ***Bottom up***

  b) **Eliminar a la raíz** (Heapify): En este caso, se intercambia al padre del Heap (el primer elemento, que es el mas grande) con el último hijo, conservando la estructura del Heap, es decir, tiene qeu mantener la integridad cada vez que se elimina una raíz.

  Bottom up: En esta etsrategia, se construye el Heap sobre la lista original, primero encuentra la posición del último nodo con hijos, y los compara con sus hijos, si no sigue el *criterio* (en este caso, el criterio es que *para cada nodo que se ve, su padre debe ser mayor a él*,), se hace el intercambio de posiciones, esto se hace para cada nodo con hijos hasta llegar a la raíz donde se verifica qeu este sea mayor a sus hijos.
  Su complejidad en todos los casos es de O($nlog(n)$), ya que cada vex que se elimina la raíz y se construye el heap, esto se hace con un heap más pequeño cada vez, ya que el tamaño de la lista se reduce en 1 porque la raíz se movió al final (su posición final) y ya no es tomada en cuenta.

Diferencias con respecto a los algoritmos vistos en la práctica 1:
Ambos (Quick y Heap) son algortiemos con complejidad de O($nlog(n)$), mucho más eficiente que los O(n^2) de Insertion, Selection y Bubble Sort.
Por lo tanto, es una diferencia considerable en cuanto al numero de comparaciones y operaciones que se realizan en Insertion, Selection y Bubble Sort, ya que estos siempre comparan todos los elementos entre sí en el mejor, peor, y caso promedio (a excepción de la mejora realizada en Bubble sort en la Prsctica 1 donde para el *mejor caso* se logró mejorar la complejidad a O(n), pero aún así sigue siendo O(n^2) para los demás casos).

2. **Implementación de Quick Sort:** En esta ocasión, se usó la versión que revisamos en la *Tarea 1 - Quick Sort*, donde se elige como pivote al último elemento, se compara al pivote con los elementos, si el elemento es menor o igual al pivote, se intercambia con el de a su lado, y con el índice i indica donde está el último elemento que fué menor al pivote, para que al final, se inetercambien el pivote con el índice i+1

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

Imagen 2.1 Partition en la implementación

```
partition(list, inicio, fin)
    pivot = list[fin]
    i = inicio - 1
    for(j=inicio; j<= fin - 1)
        if list[j] <= pivot {
            i = i + 1
            swap(list[i], list[j])
        }
    swap(list[i + 1], list[fin])
    return i + 1
```

Imagen 2.2 Partition en la el Pseudocódigo de la tarea

3. Modificación el código de heapsort
   Evitar utilizar la variable global heapsize
   Nota: Si fue necesario hacerlo para ejecutar correctamente el código, ya que si dejaba Heapsize como variable global, salía un error de variable declarada multiples veces
   Cambios realizados:

   - Eliminé la variable global heapSize declarada en utilerias.h
   - Declaré heapSize como variable local dentro de HeapSort
   - Mpdifiqué las funciones BuildHeap y Heapify para aceptar heapSize como parámetro
   - En BuildHeap, se pasa heapSize por referencia para poder modificarlo sin modificar la original.
     (Nota: había tenido problemas con esta parte, pero puede solucionarla así)

```c
void HeapSort(int* A, int size){
    int heapSize = size - 1; // local en lugar de global

    BuildHeap(A, size, &heapSize); //pasar heapSize

    printf("=== ELIMINAR LA RAÍZ ===\n");
    for(int i = size - 1; i > 0; i--){
        printf("intercambio de raiz (%d) con ultimo elemento (%d)\n", A[0], A[i]);
        swap(&A[0], &A[i]);
        heapSize--;
        printf("Heap después del intercambio: ");
        PrintArray(A, size);
        Heapify(A, 0, size, heapSize); // pasar heapSize
        printf("Heap después de reordenar: ");
        PrintArray(A, size);
        printf("\n");
    }
}
```

Imagen 3.1 Declarar heapSize como variable local dentro de HeapSort

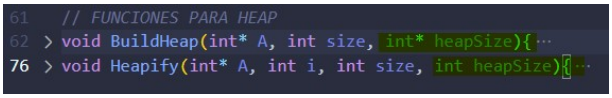


Imagen 2.2 Modificar los parámetros de BuildHeap y Heapify

2.1.1. Resultados

En esta ocasión, se mostrarán los resultados en la *sección* ***2. Verificando el funcionamiento***

2.2. Ejercicio 2. Verificando el funcionamiento

Compilación:

```
PS C:\Users\roman\EDA 2\PRACTICA 2 - TODO\Reporte\Menu Ordenamientos y Utilerias> gcc -o main main.c utilerias.c ordenamientos.c
PS C:\Users\roman\EDA 2\PRACTICA 2 - TODO\Reporte\Menu Ordenamientos y Utilerias> ./main.exe
```

Ejecución:

1. **Quick Sort**

```
=== MENU DE ORDENAMIENTO ===
1. Insertion Sort
2. Selection Sort
3. Bubble Sort
4. Quick Sort
5. Heap Sort
6. Salir
Seleccione una opcion: 4

=== QUICK SORT ===

Arreglo original: 923 819 204 973 934 119 983 275 647 832 776 467 733 359 222 931 484 741 507 905
Llama a QuickSort: Sub array :  923 819 204 973 934 119 983 275 647 832 776 467 733 359 222 931 484 741 507 905

-- partition --
sublista: Sub array :  923 819 204 973 934 119 983 275 647 832 776 467 733 359 222 931 484 741 507 905
pivote: 905
Despues de dividir: Sub array :  819 204 119 275 647 832 776 467 733 359 222 484 741 507 905 931 973 934 923 983
Pivote en posicion final: 14
----

Dividiendo:
-> Sublista izquierda: Sub array :  819 204 119 275 647 832 776 467 733 359 222 484 741 507
-> Sublista derecha: Sub array :  931 973 934 923 983
```

Iteraciones *

```
Llama a QuickSort: Sub array :  934 973

-- partition --
sublista: Sub array :  934 973
pivote: 973
Despues de dividir: Sub array :  934 973
Pivote en posicion final: 18
----
Dividiendo:
-> Sublista izquierda: Sub array :  934
-> Sublista derecha: Sub array :


Arreglo ordenado: 119 204 222 275 359 467 484 507 647 733 741 776 819 832 905 923 931 934 973 983
=== MENU DE ORDENAMIENTO ===
1. Insertion Sort
2. Selection Sort
3. Bubble Sort
4. Quick Sort
5. Heap Sort
6. Salir
Seleccione una opcion: 6
Saliendo..
PS C:\Users\roman\EDA 2\PRACTICA 2 - TODO\Reporte\Menu Ordenamientos y Utilerias> ./main.exe
```

2. **Heap Sort**

```
=== MENU DE ORDENAMIENTO ===
1. Insertion Sort
2. Selection Sort
3. Bubble Sort
4. Quick Sort
5. Heap Sort
6. Salir
Seleccione una opcion: 5

=== HEAP SORT ===

Arreglo original: 645 921 258 549 695 958 706 708 460 598 857 433 992 81 490 793 21 484 880 513
---Construyendo heap---
Heapify en la posic. 9 (valor: 598)
-->Heap actual: 645 921 258 549 695 958 706 708 460 598 857 433 992 81 490 793 21 484 880 513
Heapify en la posic. 8 (valor: 460)
Intercambiando 460 y 880
645 921 258 549 695 958 706 708 880 598 857 433 992 81 490 793 21 484 460 513 -->Heap actual: 645 921 258 549 695 958 706 708 880 598 857 433 992 81 490 793 21 484 460 513
Heapify en la posic. 7 (valor: 708)
Intercambiando 708 y 793
645 921 258 549 695 958 706 793 880 598 857 433 992 81 490 708 21 484 460 513 -->Heap actual: 645 921 258 549 695 958 706 793 880 598 857 433 992 81 490 708 21 484 460 513
Heapify en la posic. 6 (valor: 706)
-->Heap actual: 645 921 258 549 695 958 706 793 880 598 857 433 992 81 490 708 21 484 460 513
Heapify en la posic. 5 (valor: 958)
Intercambiando 958 y 992
```

Iteraciones *

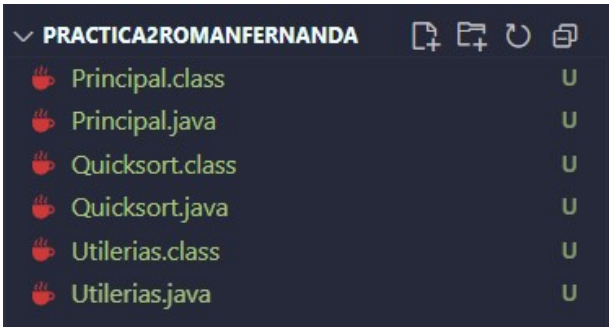
```
Heap despu|*s del intercambio: 21 258 433 81 460 484 490 513 549 598 645 695 706 708 793 857 880 921 958 992 Intercambiando 21 y 433
433 258 21 81 460 484 490 513 549 598 645 695 706 708 793 857 880 921 958 992 Heap despu|*s de reordenar: 433 258 21 81 460 484 490 513 549
598 645 695 706 708 793 857 880 921 958 992
intercambio de raiz (433) con ultimo elemento (81)
Heap despu|*s del intercambio: 81 258 21 433 460 484 490 513 549 598 645 695 706 708 793 857 880 921 958 992 Intercambiando 81 y 258
258 81 21 433 460 484 490 513 549 598 645 695 706 708 793 857 880 921 958 992 Heap despu|*s de reordenar: 258 81 21 433 460 484 490 513 549
598 645 695 706 708 793 857 880 921 958 992
intercambio de raiz (258) con ultimo elemento (21)
Heap despu|*s del intercambio: 21 81 258 433 460 484 490 513 549 598 645 695 706 708 793 857 880 921 958 992 Intercambiando 21 y 81
81 21 258 433 460 484 490 513 549 598 645 695 706 708 793 857 880 921 958 992 Heap despu|*s de reordenar: 81 21 258 433 460 484 490 513 549
598 645 695 706 708 793 857 880 921 958 992
intercambio de raiz (81) con ultimo elemento (21)
Heap despu|*s del intercambio: 21 81 258 433 460 484 490 513 549 598 645 695 706 708 793 857 880 921 958 992 Heap despu|*s de reordenar: 21
81 258 433 460 484 490 513 549 598 645 695 706 708 793 857 880 921 958 992

Arreglo ordenado: 21 81 258 433 460 484 490 513 549 598 645 695 706 708 793 857 880 921 958 992
```

2.3. Ejercicio 3. Ordenamiento en Java y practicando programas sencillos

Nota: en mi caso, usaré Visual Studio Code

1. Creación del proyecto:



2. QuickSort.java:

```java
public class Quicksort {
    public static int partition(int arr[], int low, int high){
        int pivot = arr[high];
        int i = (low-1);
        for (int j=low; j<high; j++){
            if (arr[j] <= pivot) {
                i++;
                Utilerias.intercambiar(arr, i,j);
            }
        }
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;
    return i+1;
    }
    public static void QuickSort(int arr[], int low, int high){
        if (low < high){
            int pi = partition(arr, low, high);
            QuickSort(arr, low, pi-1);
            QuickSort(arr, pi+1, high);
        }
    }
}
```

3. Principal.java:

```java
Principal.java > ...
public class Principal {
    Run | Debug
    public static void main(String args[]){
        int[] arr1 = {9,14,3,2,43,11,58,22,40,1,7,96,45,110,23,74,5,47,44,50};

        System.out.println(x:"Arreglos Originales");
        Utilerias.imprimirArreglo(arr1);

        int size=arr1.length; //obtener el tamaño del arreglo
        //pasarle a Quicksort el arreglo, su inicio, y su ultimo indice, que es su numero de elementos menos 1
        Quicksort.QuickSort(arr1, low:0, size-1);

        //Imprimir desp. de ordenar
        System.out.println(x:"\nArreglo ordenado");
        Utilerias.imprimirArreglo(arr1);

    }
}
```

Aquí usé la función lenght para arreglos que devuelve el tamaño del arreglo.

2.3.1. Resultados

Compilación y ejecución:

```
roman@RomL MINGW64 ~/EDA 2/PRACTICA 2 - TODO/Reporte/Practica2RomanFernanda
$ javac Principal.java

roman@RomL MINGW64 ~/EDA 2/PRACTICA 2 - TODO/Reporte/Practica2RomanFernanda
$ java Principal
Arreglos Originales
9 14 3 2 43 11 58 22 40 1 7 96 45 110 23 74 5 47 44 50

Arreglo ordenado
1 2 3 5 7 9 11 14 22 23 40 43 44 45 47 50 58 74 96 110

roman@RomL MINGW64 ~/EDA 2/PRACTICA 2 - TODO/Reporte/Practica2RomanFernanda
$
```

2.3.2. *Extras*

En Principal.java agregué las funciones:

- Con ayuda de la librería Random (*import java.util.Random;*) que el arreglo se llene con numeros aleatorios, cada vez que se llama al programa, genera un arreglo aleatorio diferente y lo ordena.

```
import java.util.Random;
int[] arr1 = new int[size];
Random random = new Random();

for (int i = 0; i < size ; i++) {
    arr1[i] = random.nextInt(bound:100);
}
```

- Con ayuda de la librería Scanner (*import java.util.Scanner;*) que el arreglo sea de tamaño dinámico, pedido al usuario en tiempo de ejecución, el scanner que creé se llama *sc* y recibe en entero el tamañp deseado.

```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
System.out.println(x:"¿De qué tamaño desea el arreglo? ");
int size = sc.nextInt();
int[] arr1 = new int[size];
```

Código completo:

```java
import java.util.Random;
import java.util.Scanner;

public class Principal {
    Run | Debug
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        System.out.println(x:"¿De qué tamaño desea el arreglo? ");
        int size = sc.nextInt();
        int[] arr1 = new int[size];
        Random random = new Random();

        for (int i = 0; i < size ; i++) {
            arr1[i] = random.nextInt(bound:100);
        }

        System.out.println(x:"Arreglo Original");
        Utilerias.imprimirArreglo(arr1);


        Quicksort.QuickSort(arr1, low:0, size-1);

        //Imprimir desp. de ordenar
        System.out.println(x:"\nArreglo ordenado");
        Utilerias.imprimirArreglo(arr1);

        sc.close();
    }
}
```

Resultados

Probé crear arreglos de 20, 30 y 50 elementos:

```
roman@RomL MINGW64 ~/EDA 2/PRACTICA 2 - TODO/Reporte/Practica2RomanFernanda
$ java Principal
¿De qué tamaño desea el arreglo?
20
Arreglo Original
63 56 89 76 83 31 53 32 95 26 58 83 58 54 22 65 81 24 10 94

Arreglo ordenado
10 22 24 26 31 32 53 54 56 58 58 63 65 76 81 83 83 89 94 95

roman@RomL MINGW64 ~/EDA 2/PRACTICA 2 - TODO/Reporte/Practica2RomanFernanda
$ java Principal
¿De qué tamaño desea el arreglo?
30
Arreglo Original
1 78 66 86 14 44 58 12 72 67 38 59 35 2 41 15 29 74 79 71 33 5 31 77 75 75 53 34 96 59

Arreglo ordenado
1 2 5 12 14 15 29 31 33 34 35 38 41 44 53 58 59 59 66 67 71 72 74 75 75 77 78 79 86 96

roman@RomL MINGW64 ~/EDA 2/PRACTICA 2 - TODO/Reporte/Practica2RomanFernanda
$ java Principal
¿De qué tamaño desea el arreglo?
50
Arreglo Original
74 38 8 32 95 33 16 22 12 32 93 25 66 51 74 8 83 70 93 42 60 46 74 52 82 3 61 28 5 37 52 47 58 10 0 62 38 32 87 17 14 24 34 35 6 10 93 54 2 52

Arreglo ordenado
0 2 3 5 6 8 8 10 10 12 14 16 17 22 24 25 28 32 32 33 34 35 37 38 38 42 46 47 51 52 52 52 54 58 60 61 62 66 70 74 74 74 82 83 87 93 93 93 95

roman@RomL MINGW64 ~/EDA 2/PRACTICA 2 - TODO/Reporte/Practica2RomanFernanda
$
```

2.3.3. Acercamiento al lenguaje Java hasta el momento

En mi caso, considero que no tengo mayor problema en implementar códigos o algoritmos en Java ya que personalmente antes del inicio del semestre tomé un curso de lo básico del lenguaje, así que me siento bastante familiarizada con él, considero que sólo es cuestión de seguir practicando para conocer todas sus funciones, y esto me interesa mucho ya que hasta el momento me ha gustado mucho y me parece muy interesante.

3. Conclusiones

Esta práctica me fué muy útil para seguir analizando los algoritmos de Quick y Heap sort, sobre todo la parte de sus complejidades, logré con esta práctica, en la parte del análisis general de los algotitmos (Ejercicio 1.a), ver cómo estos se diferencían de los algoritmos Insertion, Selection y Bubble vistos en la práctica 1, y comprendiendo realmente el por qué son de complejidad O($nlog(n)$), ya que esta parte se me llegó a dificultar en clase, pero considero que ahora realmente puedo identificarlos y ocmprender el por qué su complejidad, también logré conocer diferentes formas de implementar Quick sort sobre todo en cuestión a la elección del pivote y el momento en el que este se pone en su posición final, así que estos objetivos fueron cumplidos.
Por otro lado, en la parte de la implementación en Java, no tuve mayores problemas ya que como lo mencioné en el Ejercicio 3, antes de iniciar el semestre me familiaricé un poco con Java (a nivel básico), así que por esta parte, también fue cumplido el objetivo.

4. Referencias