	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: M.I. Edgar Tista García

Asignatura: Estructura de datos y algoritmos II -1317

Grupo: 10

No. de práctica(s): 2

Integrante(s): Mendoza Hernández Carlos Emiliano

No. de lista o brigada: 25

Semestre: 2023-1

Fecha de entrega: 11 de septiembre del 2022

Observaciones:

CALIFICACIÓN: _____

Práctica 2.

Algoritmos de ordenamiento. Parte2.

OBJETIVOS

- **Objetivo general:** El estudiante identificará la estructura de los algoritmos de ordenamiento HeapSort, QuickSort y MergeSort.
- **Objetivo de la clase:** El alumno revisará 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.

DESARROLLO

Ejercicio 1. Agregando Ordenamientos

Añade a tu biblioteca “ordenamientos.h” las siguientes funciones correspondientes a los algoritmos de QuickSort, HeapSort.

HeapSort():

```
int heapSize;

void HeapSort(int* A, int size){
    BuildHeap(A,size);
    int i;
    for(i = size - 1; i > 0; i--){
        swap(&A[0],&A[heapSize]);
        heapSize--;
        printf("Iteracion HS: \n");
    printArray(A,size);
    Heapify(A, 0,size);
   }
}

void Heapify(int* A, int i, int size)
{
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    int largest;

    if(l <= heapSize && A[l] > A[i])
        largest = l;
    else
        largest = i;
    if(r <= heapSize && A[r] > A[largest])
        largest = r;
    if(largest != i){
        swap(&A[i],&A[largest]);
        printArray(A,size);
        Heapify(A, largest,size);
     }

}

void BuildHeap(int* A, int size){
    heapSize = size - 1;
    int i;
    for(i = (size - 1) / 2; i >= 0; i--){
        Heapify(A, i,size);
    }
    printf("Termin%c de construir el HEAP \n",162);
}
```

QuickSort():

```
int partition (int arr[], int low, int high){
    int pivot = arr[high];
    printf("Pivote: %d    ",pivot);
    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);
}

void quickSort(int arr[], int low, int high){
    if (low < high){
        int pi = partition(arr, low, high);
        printSubArray(arr,low,pi-1);
        quickSort(arr, low, pi - 1);
        printSubArray(arr,pi+1,high);
        quickSort(arr, pi + 1, high);
    }
}
```

a) Escribe un análisis general de los nuevos algoritmos proporcionados (diferencias con respecto a los vistos en la práctica 1).

HeapSort():

```
void heapSort(int A[], int size) {
    BuildHeap(A, size);
    int i;
    for (i = size - 1; i > 0; i--) {
        swap(&A[0], &A[heapSize]);
        heapSize--;
        printf("Iteracion HS: \n");
        printArray(A, size);
        Heapify(A, 0, size);
    }
}
```

Imagen 1.1 Heap Sort

Diferencias con los algoritmos de la práctica 1: Este algoritmo funciona con ayuda de dos funciones adicionales *BuildHeap()* y *Heapify()*, las cuales llaman recursivamente a *Heapify()*. Hacen esto para ordenar el *heap* recursivamente; en cambio, los algoritmos de la práctica 1 no utilizan recursividad ni funciones adicionales.

Parámetro *"A"*: Arreglo de datos tipo *int*.

Parámetro *"size"*: Cantidad de elementos del arreglo.

QuickSort():

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        printSubArray(arr, low, pi - 1);
        quickSort(arr, low, pi - 1);
        printSubArray(arr, pi + 1, high);
        quickSort(arr, pi + 1, high);
    }
}
```

Imagen 1.2 Quick Sort

Diferencias con los algoritmos de la práctica 1: Este algoritmo también utiliza la recursividad y se auxilia de la función *partition()* para tomar un pivote y colocarlo en su posición final. A partir de esa posición, el arreglo se divide en dos subarreglos y así continua recursivamente.

Parámetro "*arr*": Arreglo de datos tipo *int*.

Parámetro "*low*": Índice del primer elemento del arreglo.

Parámetro "*high*": Índice del último elemento del arreglo.

b) Revisa cuidadosamente el algoritmo de QuickSort e indica si se trata de alguna de las implementaciones vistas en clase o es una diferente, deberás incluir evidencia para sustentar tu afirmación.

En el código de quickSort() se observa:

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        printSubArray(arr, low, pi - 1);
        quickSort(arr, low, pi - 1);
        printSubArray(arr, pi + 1, high);
        quickSort(arr, pi + 1, high);
    }
}
```

partition() devuelve un entero a *pi*

Los llamados recursivos de Quick Sort se hacen particionando al arreglo en *pi*

En el Código de partition() se observa:

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

El pivote es el **último** elemento del arreglo

Regresa la posición final del pivote

El algoritmo de esta práctica es una forma diferente a los vistos en clase, porque en este algoritmo pivoteamos desde el último elemento. En cambio, los algoritmos vistos en clase pivotean desde el primero y desde el elemento de en medio.

c) En el caso de HeapSort, describe las funciones asociando lo visto en los videos de teoría.

BuildHeap():

```
void BuildHeap(int A[], int size) {
    heapSize = size - 1;
    int i;
    for (i = (size - 1) / 2; i >= 0; i--) {
        Heapify(A, i, size);
    }
    printf("Termin%c de construir el HEAP \n", 162);
}
```

Imagen 1.3 Build Heap

Descripción breve de la función: Convierte el arreglo a ordenar en un heap

Parámetro *"A"*: Arreglo de datos tipo *int*. *A es la dirección en memoria del arreglo.

Parámetro "i": Posición en el *heap* del elemento más grande.

Parámetro "*size*": Cantidad de elementos del arreglo.

Heapify():

```
void Heapify(int A[], int i, int size) {
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    int largest;

    if (l <= heapSize && A[l] > A[i])
        largest = l;
    else
        largest = i;
    if (r <= heapSize && A[r] > A[largest])
        largest = r;
    if (largest != i) {
        swap(&A[i], &A[largest]);
        printArray(A, size);
        Heapify(A, largest, size);
    }
}
```

Imagen 1.4 Heapify

Descripción breve de la función: Conserva la integridad de la estructura de heap con el resto de elementos.

Parámetro *"A"*: Arreglo de datos tipo *int*. **A* es la dirección en memoria del arreglo.

Parámetro "i": Posición en el *heap* del elemento más grande.

Parámetro *"size"*: Cantidad de elementos del arreglo.

Ejercicio 2. MergeSort

Agrega a la biblioteca ordenamientos el algoritmo de MergeSort basado en el pseudocódigo mostrado en el video. Indica en el reporte las dificultades de convertir el pseudocódigo de este algoritmo en lenguaje C.

Observaciones: Con el arreglo ordenado en su estado inicial, el algoritmo no realizó ninguna iteración, por lo tanto, la implementación es correcta.

MergeSort():

```
void mergeSort(int A[], int low, int high) {
    int mid;
    if (low < high) {
        mid = (low + high) / 2;
        mergeSort(A, low, mid);
        mergeSort(A, mid + 1, high);
        merge(A, low, mid, high);
    }
}
```

Imagen 1.5 Merge Sort

Diferencias con los algoritmos de la práctica 1: Este algoritmo también utiliza la recursividad para dividir el arreglo en subarreglos más pequeños y se auxilia de la función *merge()* para "unir" dos subarreglos contiguos en orden. Utiliza memoria adicional.

Parámetro *"A"*: Arreglo de datos tipo *int*.

Parámetro *"low"*: Índice del primer elemento del arreglo.

Parámetro *"high"*: Índice del último elemento del arreglo.

merge():

```c
void merge(int A[], int left, int mid, int right) {
    int A2[right];
    int i, j, k;
    k = 0;
    i = left;
    j = mid + 1;
    while (i <= mid && j <= right) {
        if (A[i] < A[j])
            A2[k++] = A[i++];
        else
            A2[k++] = A[j++];
    }
    while (i <= mid)
        A2[k++] = A[i++];
    while (j <= right)
        A2[k++] = A[j++];
    for (i = right; i >= left; i--)
        A[i] = A2[--k];
}
```

Imagen 1.6 merge

Parámetro "*A*": Arreglo de datos tipo *int*.

Parámetro "*left*": Índice del primer elemento del primer sub-arreglo.

Parámetro "*mid*": Índice del último elemento del primer sub-arreglo.

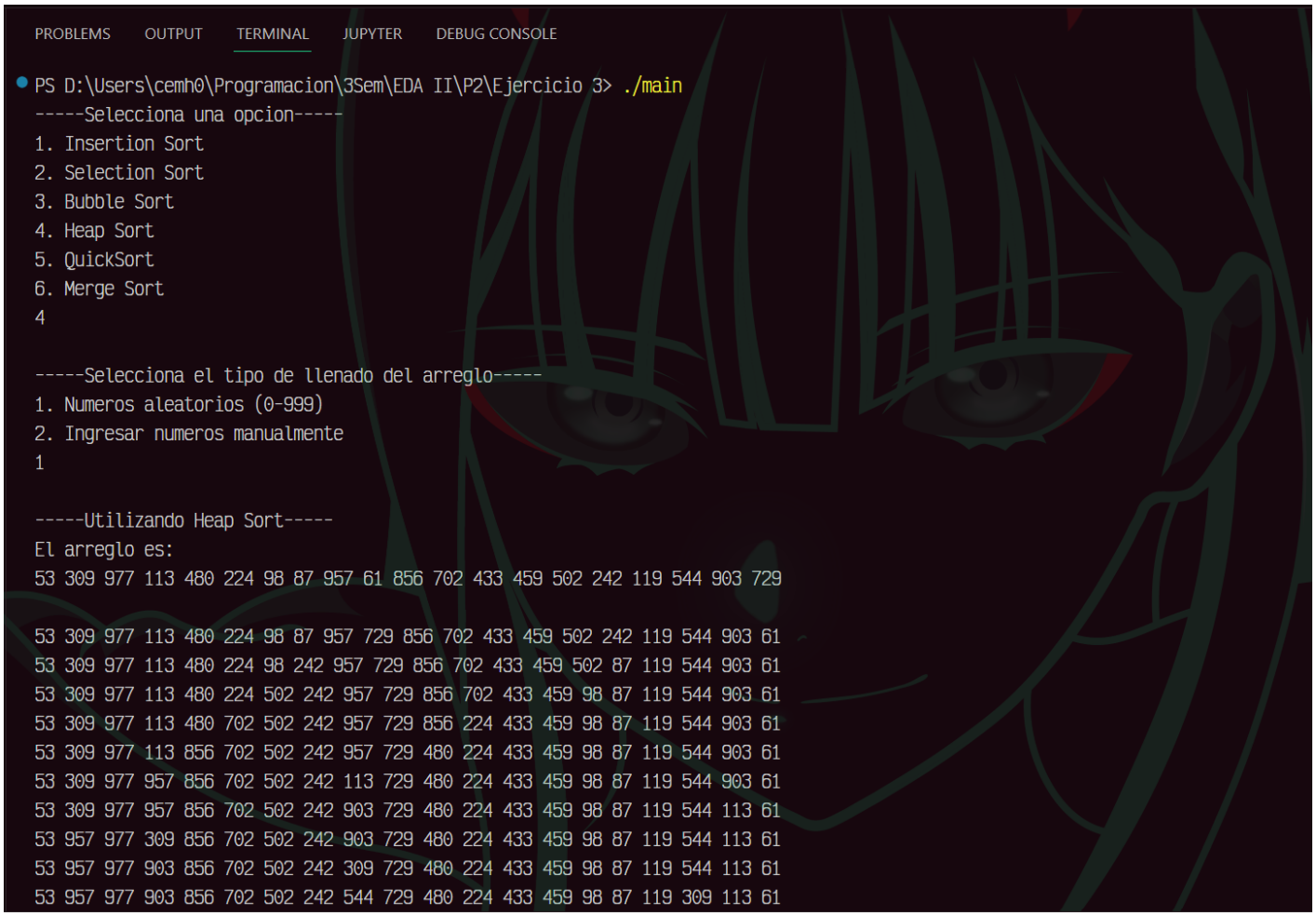
Parámetro "*right*": Índice del último elemento del segundo sub-arreglo.

Observaciones: La única dificultad que se presentó al codificar el pseudocódigo en C fue reemplazar el uso de una lista por un arreglo de datos tipo *int*.

Ejercicio 3. Verificando el funcionamiento

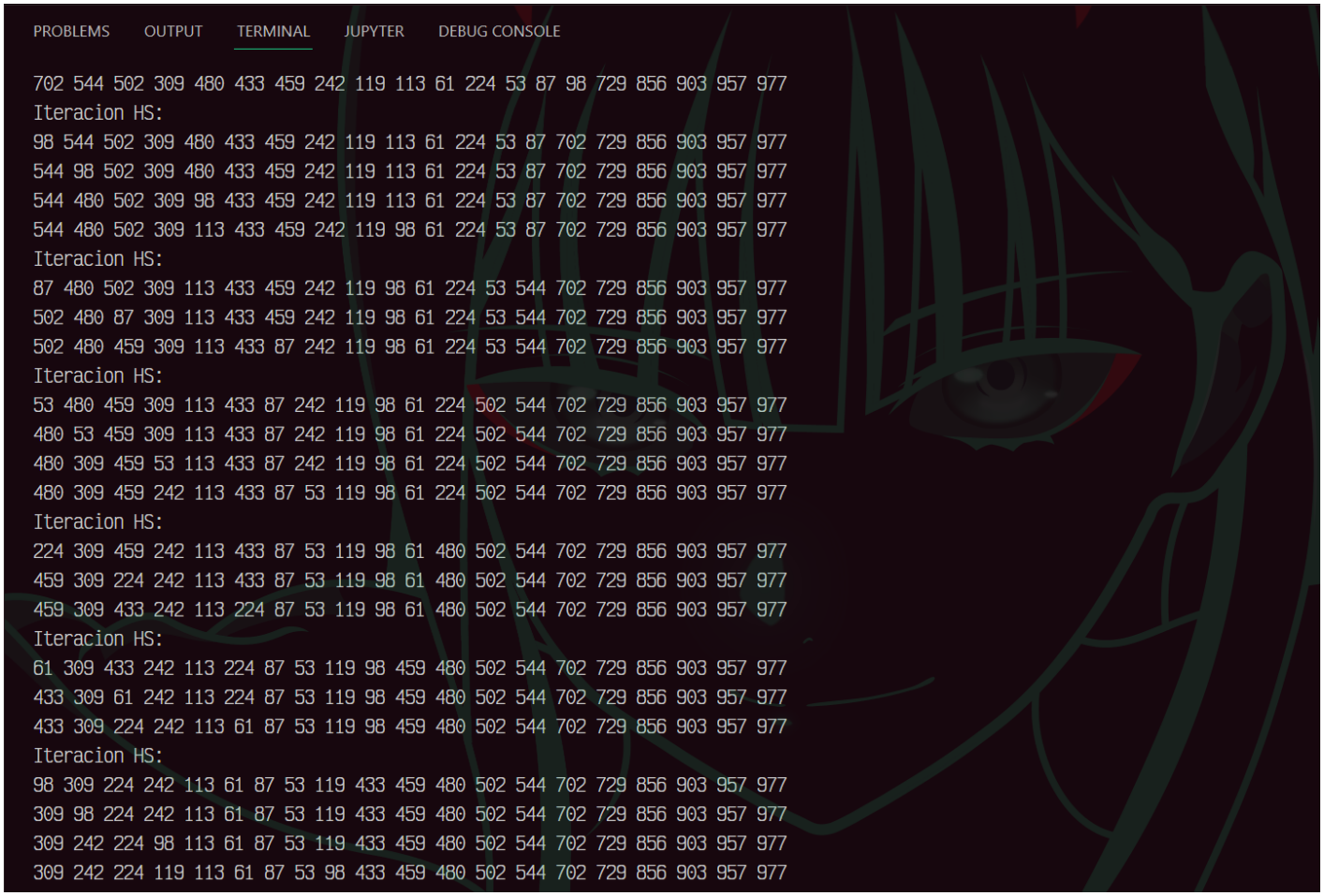
- En el menú de usuario del ejercicio 2 de la práctica 1, agrega los nuevos ordenamientos para que el usuario pueda verificar el funcionamiento de los nuevos algoritmos integrados en el arreglo de prueba de 20 elementos.
- Deberás incluir en el reporte capturas de pantalla de la ejecución del programa, así como el análisis de los puntos importantes a considerar en su elaboración.

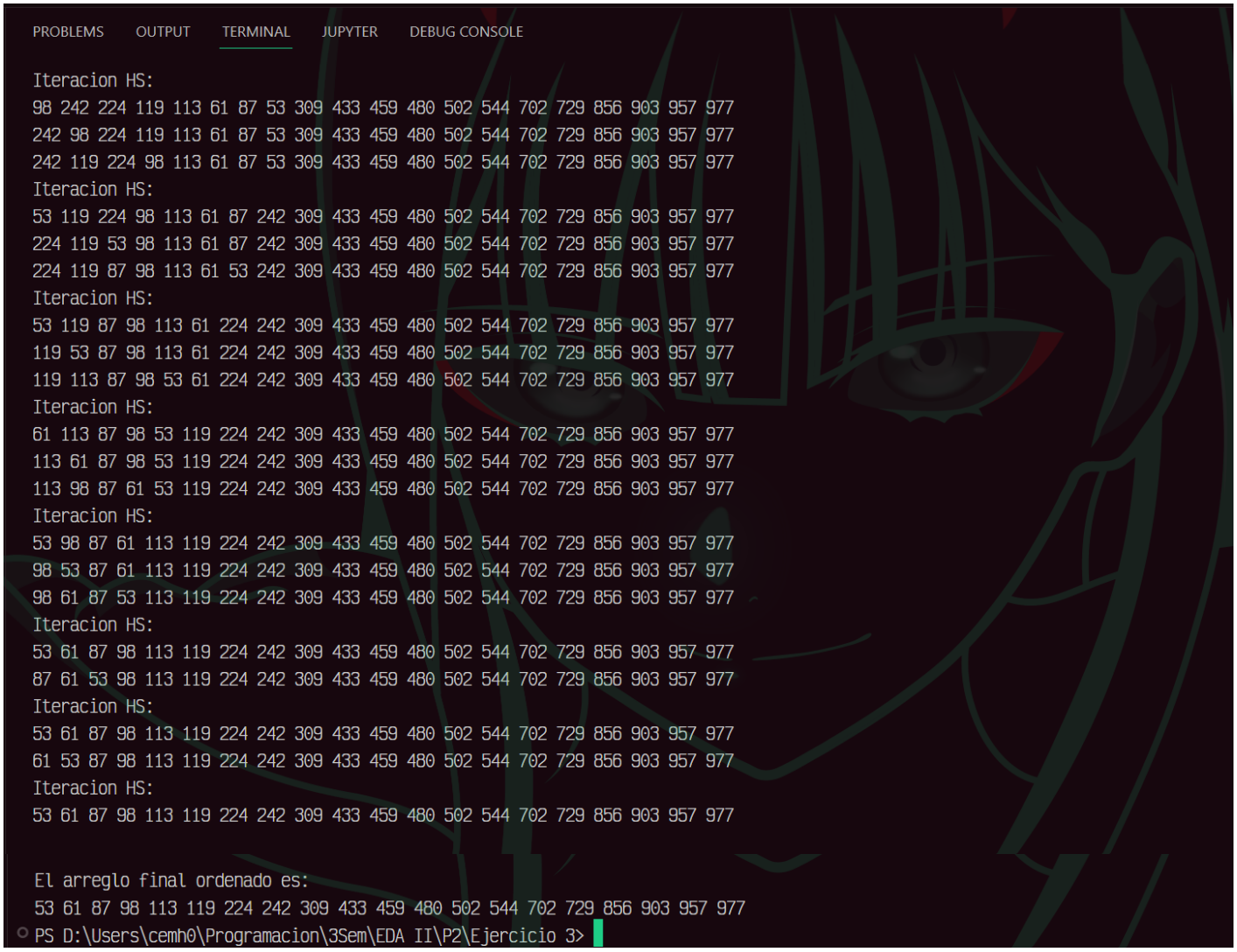
Ejecución de Heap Sort con 20 números aleatorios



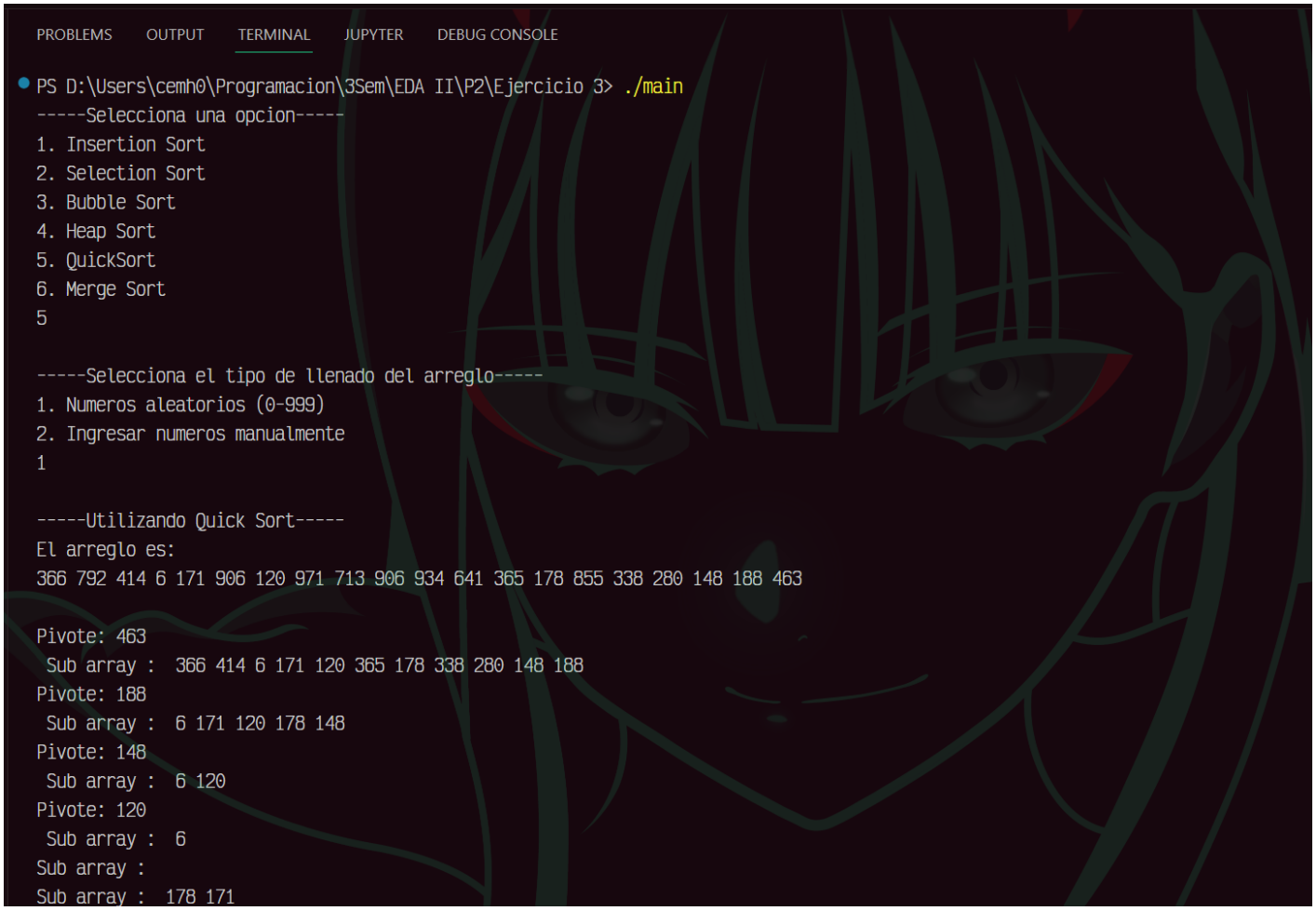
PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE

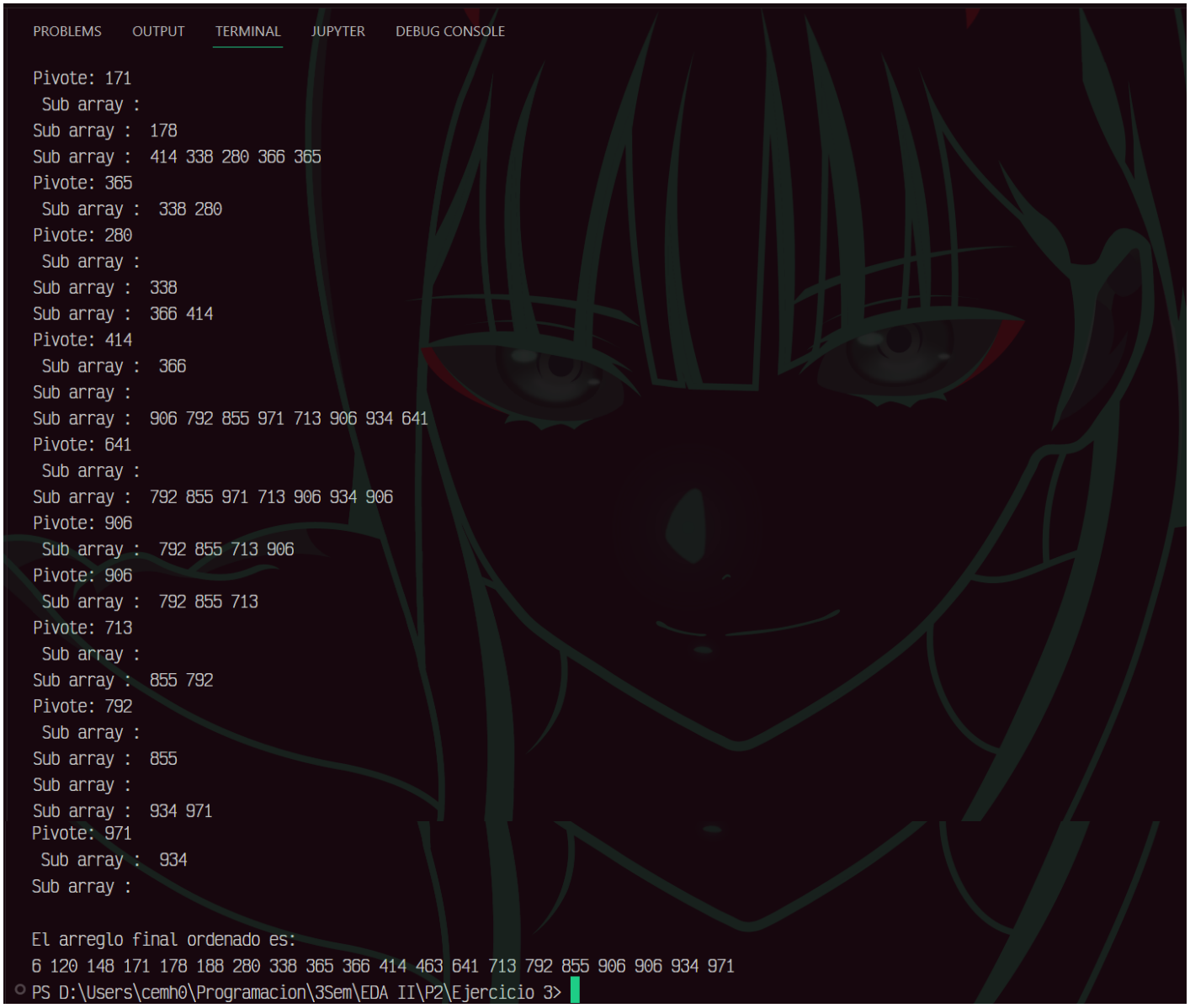
```
977 957 53 903 856 702 502 242 544 729 480 224 433 459 98 87 119 309 113 61
977 957 702 903 856 53 502 242 544 729 480 224 433 459 98 87 119 309 113 61
977 957 702 903 856 433 502 242 544 729 480 224 53 459 98 87 119 309 113 61
Terminó de construir el HEAP
Iteracion HS:
61 957 702 903 856 433 502 242 544 729 480 224 53 459 98 87 119 309 113 977
957 61 702 903 856 433 502 242 544 729 480 224 53 459 98 87 119 309 113 977
957 903 702 61 856 433 502 242 544 729 480 224 53 459 98 87 119 309 113 977
957 903 702 544 856 433 502 242 61 729 480 224 53 459 98 87 119 309 113 977
957 903 702 544 856 433 502 242 309 729 480 224 53 459 98 87 119 61 113 977
Iteracion HS:
113 903 702 544 856 433 502 242 309 729 480 224 53 459 98 87 119 61 957 977
903 113 702 544 856 433 502 242 309 729 480 224 53 459 98 87 119 61 957 977
903 856 702 544 113 433 502 242 309 729 480 224 53 459 98 87 119 61 957 977
903 856 702 544 729 433 502 242 309 113 480 224 53 459 98 87 119 61 957 977
Iteracion HS:
61 856 702 544 729 433 502 242 309 113 480 224 53 459 98 87 119 903 957 977
856 61 702 544 729 433 502 242 309 113 480 224 53 459 98 87 119 903 957 977
856 729 702 544 61 433 502 242 309 113 480 224 53 459 98 87 119 903 957 977
856 729 702 544 480 433 502 242 309 113 61 224 53 459 98 87 119 903 957 977
Iteracion HS:
119 729 702 544 480 433 502 242 309 113 61 224 53 459 98 87 856 903 957 977
729 119 702 544 480 433 502 242 309 113 61 224 53 459 98 87 856 903 957 977
729 544 702 119 480 433 502 242 309 113 61 224 53 459 98 87 856 903 957 977
729 544 702 309 480 433 502 242 119 113 61 224 53 459 98 87 856 903 957 977
Iteracion HS:
87 544 702 309 480 433 502 242 119 113 61 224 53 459 98 729 856 903 957 977
702 544 87 309 480 433 502 242 119 113 61 224 53 459 98 729 856 903 957 977
702 544 502 309 480 433 87 242 119 113 61 224 53 459 98 729 856 903 957 977
```


PROBLEMS
OUTPUT
TERMINAL
JUPYTER
DEBUG CONSOLE
702 544 502 309 480 433 459 242 119 113 61 224 53 87 98 729 856 903 957 977
Iteracion HS:
98 544 502 309 480 433 459 242 119 113 61 224 53 87 702 729 856 903 957 977
544 98 502 309 480 433 459 242 119 113 61 224 53 87 702 729 856 903 957 977
544 480 502 309 98 433 459 242 119 113 61 224 53 87 702 729 856 903 957 977
544 480 502 309 113 433 459 242 119 98 61 224 53 87 702 729 856 903 957 977
Iteracion HS:
87 480 502 309 113 433 459 242 119 98 61 224 53 544 702 729 856 903 957 977
502 480 87 309 113 433 459 242 119 98 61 224 53 544 702 729 856 903 957 977
502 480 459 309 113 433 87 242 119 98 61 224 53 544 702 729 856 903 957 977
Iteracion HS:
53 480 459 309 113 433 87 242 119 98 61 224 502 544 702 729 856 903 957 977
480 53 459 309 113 433 87 242 119 98 61 224 502 544 702 729 856 903 957 977
480 309 459 53 113 433 87 242 119 98 61 224 502 544 702 729 856 903 957 977
480 309 459 242 113 433 87 53 119 98 61 224 502 544 702 729 856 903 957 977
Iteracion HS:
224 309 459 242 113 433 87 53 119 98 61 480 502 544 702 729 856 903 957 977
459 309 224 242 113 433 87 53 119 98 61 480 502 544 702 729 856 903 957 977
459 309 433 242 113 224 87 53 119 98 61 480 502 544 702 729 856 903 957 977
Iteracion HS:
61 309 433 242 113 224 87 53 119 98 459 480 502 544 702 729 856 903 957 977
433 309 61 242 113 224 87 53 119 98 459 480 502 544 702 729 856 903 957 977
433 309 224 242 113 61 87 53 119 98 459 480 502 544 702 729 856 903 957 977
Iteracion HS:
98 309 224 242 113 61 87 53 119 433 459 480 502 544 702 729 856 903 957 977
309 98 224 242 113 61 87 53 119 433 459 480 502 544 702 729 856 903 957 977
309 242 224 98 113 61 87 53 119 433 459 480 502 544 702 729 856 903 957 977
309 242 224 119 113 61 87 53 98 433 459 480 502 544 702 729 856 903 957 977

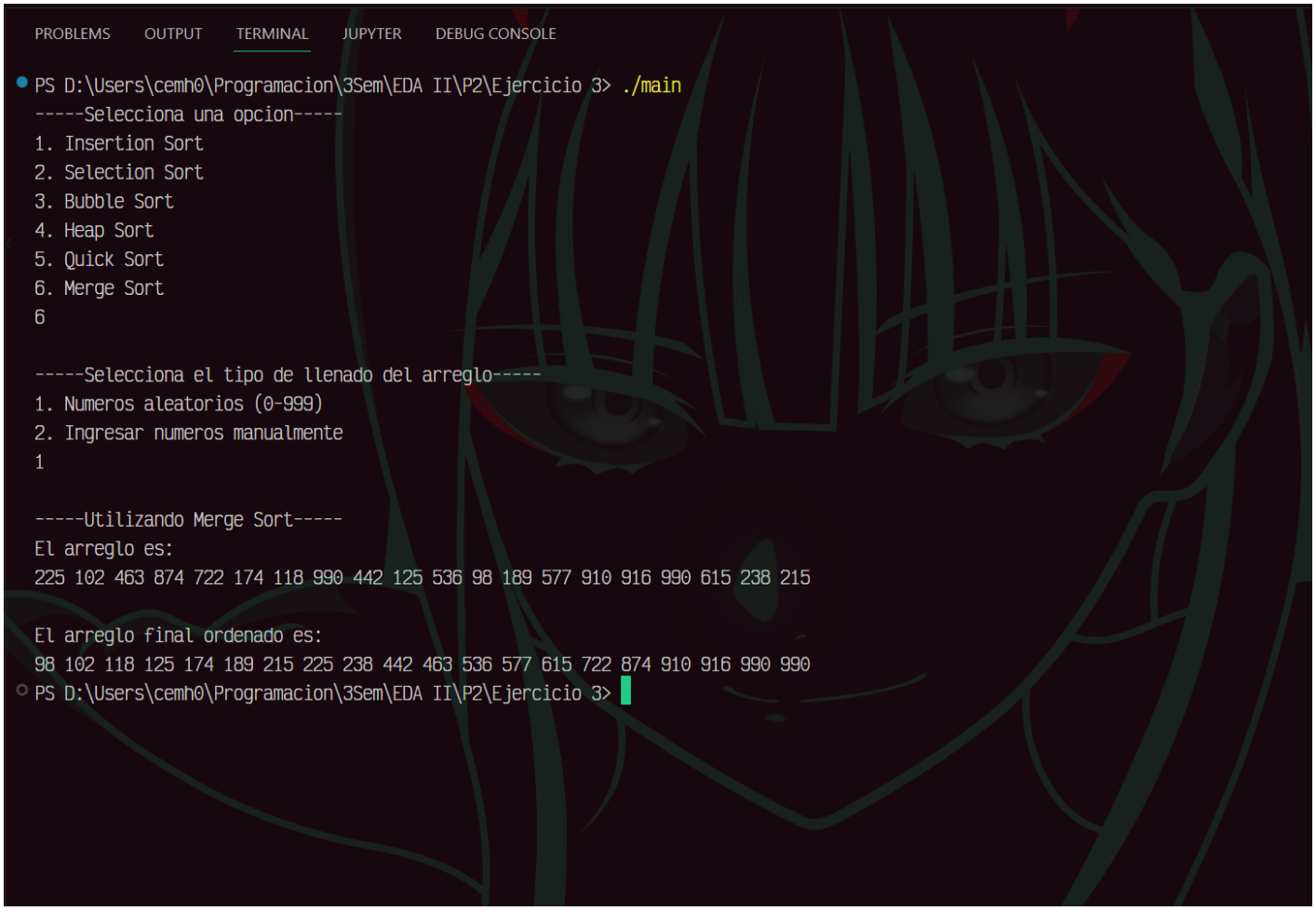

PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE
Iteracion HS:
98 242 224 119 113 61 87 53 309 433 459 480 502 544 702 729 856 903 957 977
242 98 224 119 113 61 87 53 309 433 459 480 502 544 702 729 856 903 957 977
242 119 224 98 113 61 87 53 309 433 459 480 502 544 702 729 856 903 957 977
Iteracion HS:
53 119 224 98 113 61 87 242 309 433 459 480 502 544 702 729 856 903 957 977
224 119 53 98 113 61 87 242 309 433 459 480 502 544 702 729 856 903 957 977
224 119 87 98 113 61 53 242 309 433 459 480 502 544 702 729 856 903 957 977
Iteracion HS:
53 119 87 98 113 61 224 242 309 433 459 480 502 544 702 729 856 903 957 977
119 53 87 98 113 61 224 242 309 433 459 480 502 544 702 729 856 903 957 977
119 113 87 98 53 61 224 242 309 433 459 480 502 544 702 729 856 903 957 977
Iteracion HS:
61 113 87 98 53 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
113 61 87 98 53 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
113 98 87 61 53 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
Iteracion HS:
53 98 87 61 113 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
98 53 87 61 113 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
98 61 87 53 113 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
Iteracion HS:
53 61 87 98 113 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
87 61 53 98 113 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
Iteracion HS:
53 61 87 98 113 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
61 53 87 98 113 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
Iteracion HS:
53 61 87 98 113 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
El arreglo final ordenado es:
53 61 87 98 113 119 224 242 309 433 459 480 502 544 702 729 856 903 957 977
PS D:\Users\cemh0\Programacion\3Sem\EDA II\P2\Ejercicio 3>

Ejecución de Quick Sort con 20 números aleatorios




PROBLEMS
OUTPUT
TERMINAL
JUPYTER
DEBUG CONSOLE
Pivote: 171
Sub array :
Sub array : 178
Sub array : 414 338 280 366 365
Pivote: 365
Sub array : 338 280
Pivote: 280
Sub array :
Sub array : 338
Sub array : 366 414
Pivote: 414
Sub array : 366
Sub array :
Sub array : 906 792 855 971 713 906 934 641
Pivote: 641
Sub array :
Sub array : 792 855 971 713 906 934 906
Pivote: 906
Sub array : 792 855 713 906
Pivote: 906
Sub array : 792 855 713
Pivote: 713
Sub array :
Sub array : 855 792
Pivote: 792
Sub array :
Sub array : 855
Sub array :
Sub array : 934 971
Pivote: 971
Sub array : 934
Sub array :
El arreglo final ordenado es:
6 120 148 171 178 188 280 338 365 366 414 463 641 713 792 855 906 906 934 971
PS D:\Users\cemh0\Programacion\3Sem\EDA II\P2\Ejercicio 3>

Ejecución de Merge Sort con 20 números aleatorios

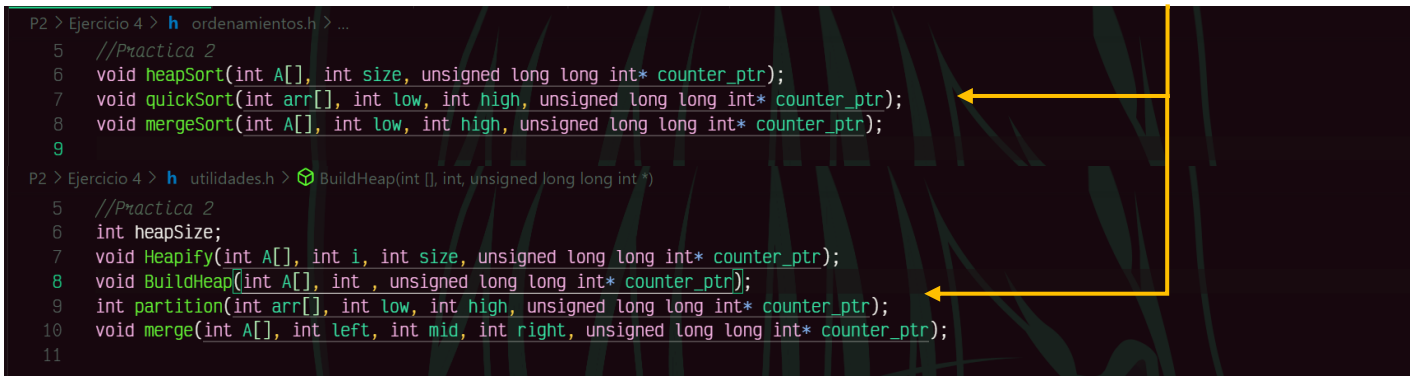


Observaciones: Se siguió una estructura similar para cada *case* agregado al menú del programa de la práctica 1. Consiste en Imprimir el arreglo antes de ordenarse, llamar a la función de ordenamiento e imprimir el arreglo después de ordenarse.

Ejercicio 4. Complejidad computacional

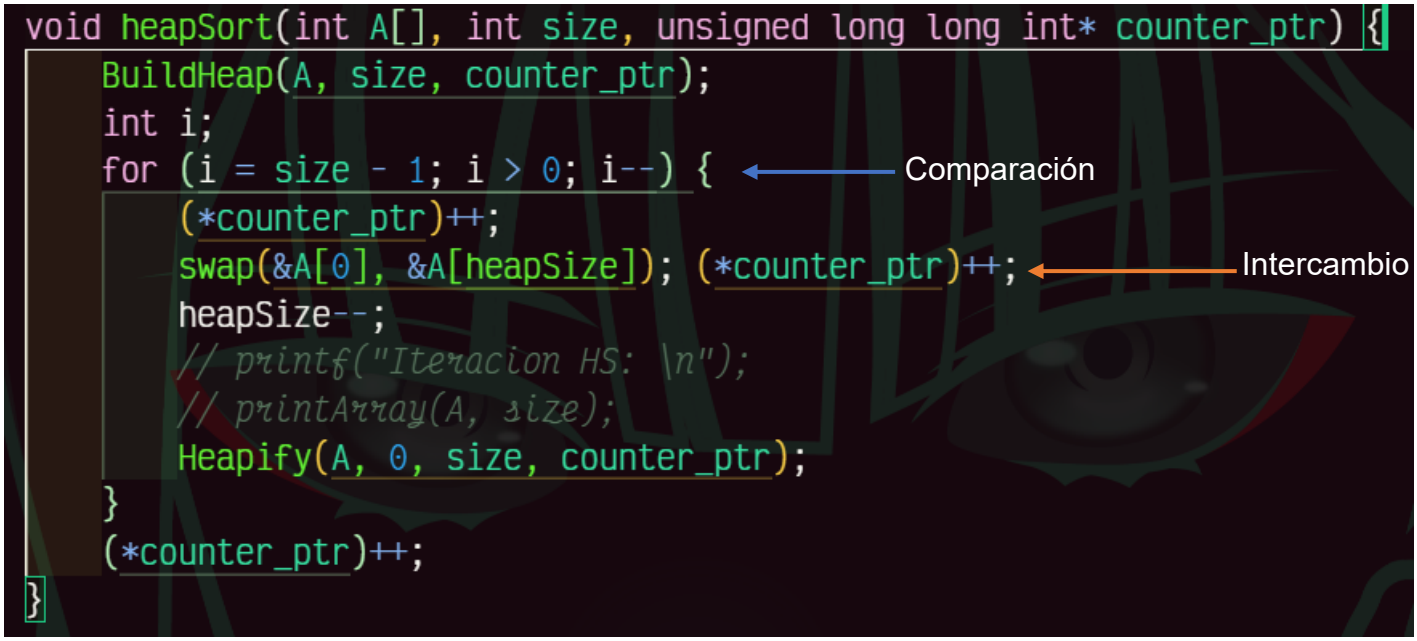
- Agrega en los algoritmos de ordenamiento, las instrucciones necesarias para contabilizar el número de operaciones (recuerda que se deben considerar aquellas que afectan el arreglo a ordenar).

Observaciones: Para contar las operaciones, se agregó a la firma de las funciones un apuntador a la variable global que contará el total de operaciones en el *main*.



Nota: Se comentaron las sentencias que imprimen el arreglo para agilizar el conteo.

Operaciones contabilizadas:



```c
void quickSort(int arr[], int low, int high, unsigned long long int* counter_ptr) {
    (*counter_ptr)++;
    if (low < high) {
        int pi = partition(arr, low, high, counter_ptr);
        // printSubArray(arr, low, pi - 1);
        quickSort(arr, low, pi - 1, counter_ptr);
        // printSubArray(arr, pi + 1, high);
        quickSort(arr, pi + 1, high, counter_ptr);
    }
}
```

Comparación

```c
void mergeSort(int A[], int low, int high, unsigned long long int* counter_ptr) {
    int mid;
    (*counter_ptr)++;
    if (low < high) {
        mid = (low + high) / 2;
        mergeSort(A, low, mid, counter_ptr);
        mergeSort(A, mid + 1, high, counter_ptr);
        merge(A, low, mid, high, counter_ptr);
    }
}
```

Comparación

```c
void BuildHeap(int A[], int size, unsigned long long int* counter_ptr) {
    heapSize = size - 1;
    int i;
    for (i = (size - 1) / 2; i ⩾ 0; i--) {
        (*counter_ptr)++;
        Heapify(A, i, size, counter_ptr);
    }
    (*counter_ptr)++;
    // printf("Termin%c de construir el HEAP \n", 162);
}
```

Comparación

```c
void Heapify(int A[], int i, int size, unsigned long long int* counter_ptr) {
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    int largest;
    (*counter_ptr)++;
    if (l ≤ heapSize && A[l] > A[i])
        largest = l;
    else
        largest = i;
    (*counter_ptr)++;
    if (r ≤ heapSize && A[r] > A[largest])
        largest = r;
    (*counter_ptr)++;
    if (largest ≠ i) {
        swap(&A[i], &A[largest]); (*counter_ptr)++;
        // printArray(A, size);
        Heapify(A, largest, size, counter_ptr);
    }
}
```

Comparación

Comparación

Comparación

Intercambio

```c
int partition(int arr[], int low, int high, unsigned long long int* counter_ptr) {
    int pivot = arr[high];
    // printf("Pivote: %d   \n ", pivot);
    int j, i = (low - 1);
    for (j = low; j ≤ high - 1; j++) {
        (*counter_ptr)+=2;
        if (arr[j] ≤ pivot) {
            i++;
            swap(&arr[i], &arr[j]); (*counter_ptr)++;
        }
    }
    (*counter_ptr)++;
    swap(&arr[i + 1], &arr[high]); (*counter_ptr)++;
    return (i + 1);
}
```

Comparación

Comparación

Intercambio

Intercambio

```
void merge(int A[], int left, int mid, int right, unsigned long long int* counter_ptr) {
    int A2[right];
    int i, j, k;
    k = 0;
    i = left;
    j = mid + 1;
    while (i ⩽ mid && j ⩽ right) {
        (*counter_ptr)+=2;
        if (A[i] < A[j])
            A2[k++] = A[i++];
        else
            A2[k++] = A[j++];
    }
    (*counter_ptr)++;
    while (i ⩽ mid) {
        A2[k++] = A[i++]; (*counter_ptr)++;
    }
    (*counter_ptr)++;
    while (j ⩽ right) {
        A2[k++] = A[j++]; (*counter_ptr)++;
    }
    (*counter_ptr)++;
    for (i = right; i ⩾ left; i--)
        A[i] = A2[--k]; (*counter_ptr)+=2;
    (*counter_ptr)++;
}
```

Comparación

Comparación

Comparación

Comparación

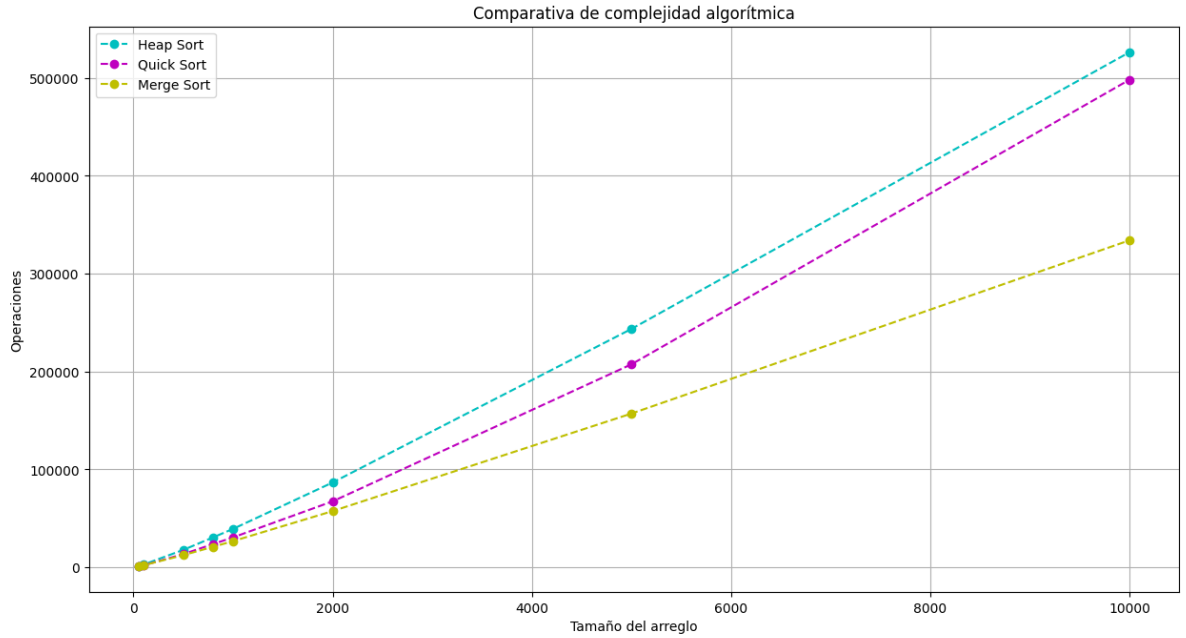
Comparación

Inserción

- Al igual que en la práctica 1, realiza pruebas con arreglos de diferente tamaño:
  - 50, 100, 500, 800, 1000, 2000, 5000, 10,000
- Realiza al menos 5 ejecuciones en los tamaños mencionados y muestra una tabla general con promedio del número de operaciones en cada uno.

TAM	Heap Sort	Quick Sort	Merge Sort
50	1109	784	898
100	2645	1902	2005
500	17 697	13 404	12 329
800	30 517	23 679	20 887
1000	39 171	30 481	26 671
2000	86 595	67 322	57 360
5000	243 513	207 357	157 030
10 000	526 219	498 129	334 021

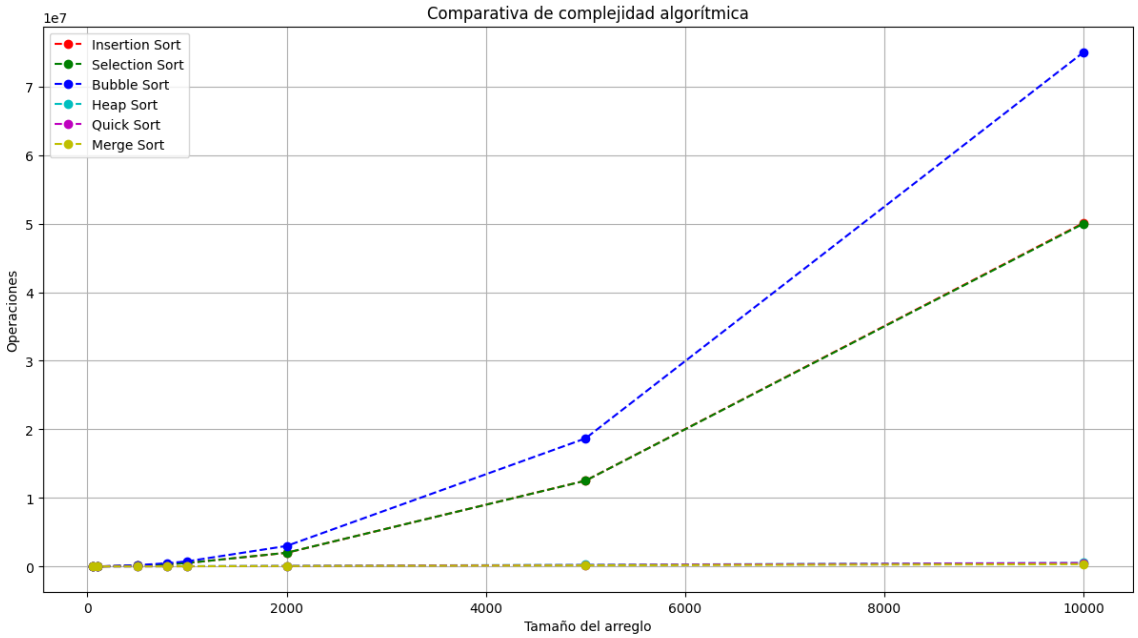
- Realiza gráficas de la cantidad de operaciones que se realizan de acuerdo con el tamaño de la entrada y explica tus resultados.



Gráfica 1. Comparativa entre Heap Sort, Quick Sort y Merge Sort

Observaciones: Merge Sort fue el que arrojó mejores resultados con base en la cantidad de operaciones; sin embargo, si tomamos en cuenta el uso de memoria adicional, Merge Sort tiene una desventaja frente a Quick Sort y Heap Sort, los cuales no necesitan memoria adicional.

- En estas gráficas observarás la complejidad de cada algoritmo, de ser posible compáralas con las gráficas obtenidas en la práctica 1.



Observaciones: Insertion Sort, Selection Sort y Bubble Sort tienen comportamientos "similares" dado que los tres tienen una complejidad $O(n^2)$. En cambio, Heap Sort, Quick Sort y Merge Sort están muy por debajo de las operaciones de los tres anteriores porque su complejidad es $O(n \cdot \log n)$.

Ejercicio 5. Ordenamiento en Java y probando programas sencillos

Con ayuda de NetBeans crea un nuevo proyecto “Practica2ApellidoPaternoyNombre”, agrega en este proyecto los archivos “QuickSort.java”, “MergeSort.java” y “utilerias.java”. En ellos coloca el siguiente código:

```
class Quicksort {
    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.swap(arr, i,j);

            }
        }

        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;
        return i+1;
    }

    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);
        }
    }
}

class Utilerias {

    static void printArray(int arr[]){
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }

    static void swap(int arr[], int i, int j){
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

}
```

```
class MergeSort {

    void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    void merge(int arr[], int l, int m, int r)
    {
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[] = new int [n1];
        int R[] = new int [n2];

        for (int i=0; i<n1; ++i)
            L[i] = arr[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = arr[m + 1+ j];

        int i = 0, j = 0;

        int k = l;
        while (i < n1 && j < n2)
        {
            if (L[i] <= R[j])
            {
                arr[k] = L[i];
                i++;
            }
            else
            {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    void sort(int arr[], int l, int r) {
        if (l < r) {
            int m = (l+r)/2;

            sort(arr, l, m);
            sort(arr , m+1, r);

            merge(arr, l, m, r);
        }
    }
}
```

- Crea un nuevo archivo “Principal.java” para la ejecución del programa (una clase con la función main) y agrega las instrucciones necesarias para crear un arreglo de 20 elementos y utilizar el método correspondiente para realizar el ordenamiento de QuickSort.

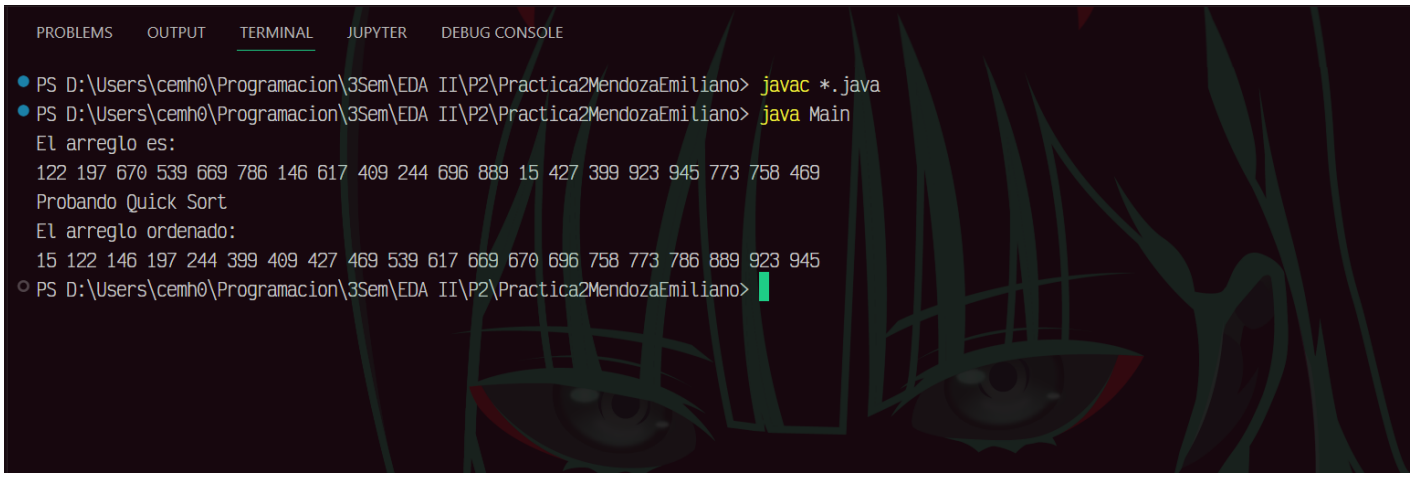
```
public class Principal {
     public static void main(String args[]){
        //Declara e inicializa un arreglo de 20 elementos
        //agrega las instruccuones para utilizar el ordenamiento Quick sort
        //imprime el arreglo ordenado

     }
}
```

- Reemplaza las instrucciones del uso de QuickSort para probar ahora el algoritmo de MergeSort, agrega las capturas de pantalla en el reporte de la práctica.
- En el reporte indica las dificultades que tuviste al realizar el programa, y también agrega comentarios respecto a tu acercamiento al lenguaje hasta este punto.

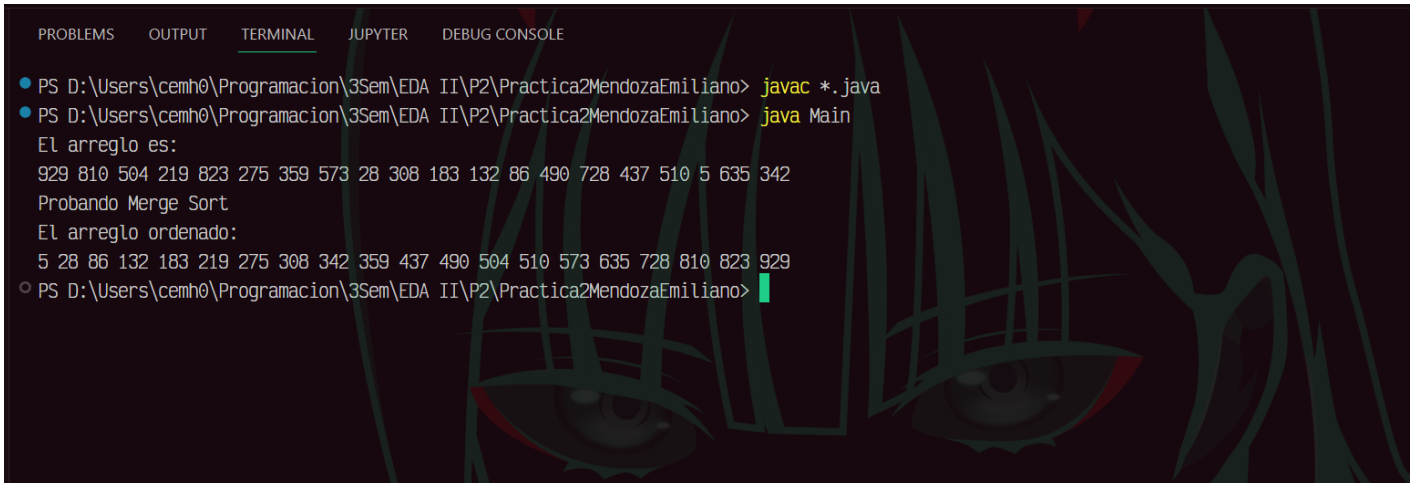
Ejecución del proyecto

NOTA: Se utilizó el editor de texto Visual Studio Code para abrir el proyecto, y la ejecución usando las extensiones *Language Support for Java by Red Hat* y *Debugger for Java.*

Quick Sort



Merge Sort



Observaciones: Los métodos de Quick Sort son estáticos, por lo tanto, se hizo el llamado a ellos de manera estática. Por otra parte, los métodos de Merge Sort no lo son, así que se instanció la clase para llamar a los métodos.

Comentario adicional: Hasta la fecha mi experiencia con Java ha sido el uso de clases de uso general, arreglos, manejo de cadenas, estructuras de repetición, control de flujo y herencia.

CONCLUSIONES

Algunos puntos que se concluyen de la primera actividad son los siguientes:

- Los algoritmos de Heap Sort, Quick Sort y Merge Sort utilizan la recursividad y utilizan funciones adicionales.
- Quick Sort tiene al menos 3 maneras de resolver el problema, dependiendo del elemento que escojamos para pivotear. Se revisó en el primer elemento, en el último y en el de en medio.

Para el segundo ejercicio se puede concluir que:

- El pseudocódigo es muy similar a la codificación en C.
- La modificación que se realizó fue para trabajar con arreglos de enteros.

Al término del cuarto ejercicio se llegó a las siguientes conclusiones:

- Merge Sort fue el que menos operaciones realizó para el caso promedio, pero su desventaja es que usa memoria adicional.
- Quick Sort y Heap Sort tienen comportamientos similares para el caso promedio.

En comparación con los algoritmos de la práctica 1 (Insertion Sort, Selection Sort y Bubble Sort), los algoritmos de esta práctica son bastante más efectivos ($O(n \cdot \log n)$ vs $O(n^2)$).
Por último, los aspectos relevantes a los que se llegó con el quinto ejercicio son:

- Se encontraron dos maneras diferentes de hacer el llamado a los algoritmos: esto se debe a la diferencia de ser un método estático en el caso de Quick Sort, el cual no necesita crear una instancia de la clase; y Merge Sort, que no es un método estático y por lo tanto se debe crear de antemano una instancia de la clase Merge.

Dicho todo lo anterior, se ha cumplido con cada uno de los objetivos planteados para el desarrollo de esta práctica dado que se identificaron las diferencias de los algoritmos de esta práctica con los de la práctica anterior, así como se comprendió su funcionamiento. También, se identificaron varias formas de implementar Quick Sort. Por último, se realizó un programa sencillo en Java aplicando los algoritmos aprendidos en esta práctica.

Como comentario final, considero que la mayor ventaja de estos algoritmos es su eficiencia; sin embargo, la comprensión de su funcionamiento es más compleja que los algoritmos anteriores.