



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 2

Grupo: 8

No de Práctica(s): 1] ALGORITMOS DE ORDENAMIENTO PARTE 1

Integrante(s): ADOLFO ROMÁN JIMENEZ

No. de Equipo de cómputo empleado: TRABAJO EN CASA

No. de Lista o Brigada:

Semestre: 2022 - 1

Fecha de entrega: 16 DE SEPTIEMBRE DE 2021

CALIFICACIÓN: _____

Práctica 1

Algoritmos de Ordenamiento Parte 1

September 16, 2021

1 Objetivo

El estudiante identificará la estructura de los algoritmos de ordenamiento InsertionSort, SelectionSort y HeapSort.

2 Desarrollo

2.1 Ejercicio 1

2.1.1 utilidades.h

El archivo de `utilidades.h` contiene prototipos de funciones auxiliares para la ejecución de la biblioteca `ordenamientos.c`. Los prototipos de las funciones, son los siguientes:

- `swap`
- `printArray`
- `printSubArray`

2.1.2 utilidades.c

El archivo de `utilidades.c` contiene la declaración de las 3 funciones auxiliares que servirán a `ordenamientos.c` principalmente para el *output* grafico del resultado de la ejecución de los algoritmos de ordenamiento.

Las funciones incluidas, son las siguientes:

- `swap()`

La función *swap* es de tipo *void* (lo que significa que no devuelve ningún tipo de dato) y utiliza dos apuntadores como argumentos que son *a* y *b*. Dentro de la función se utiliza una variable *t* para depositar el valor en *a*, después, se asigna el valor en *b* al valor en *a* y por ultimo se transmite el valor depositado en la variable temporal *t* a donde apunta *b*.

- `printArray()`

La función *printArray*, toma como argumento un arreglo de enteros *arr* y la longitud de ese arreglo con una variable entera *size*. Después declara una variable entera *i* para efectos del *for-loop* que ira imprimiendo cada uno de los valores dentro del arreglo.

Al finalizar la función imprime un salto de linea.

- `printSubArray()`

La función *printSubArray* toma 3 argumentos que son un arreglo de enteros *arr* y dos variables enteras, *low* y *high*. Después, ya dentro de la función, se declara una variable entera *i* para que funcione como contador dentro del *for-loop* y se imprime la leyenda "*Sub array:*" antes de comenzar a imprimir el arreglo.

Al momento de iniciar el *for-loop* el valor en *low* se asigna a la variable *i* y comenzara a iterar sumando 1 en cada ciclo hasta alcanzar un valor menor o igual al existente en la variable *high*, en cada ciclo se imprime el valor del arreglo a partir del indice contenido en *low* y termina con el valor en el indice *high*.

Al finalizar la función, imprime un salto de linea.

2.1.3 ordenamientos.h

El archivo `ordenamientos.h` contiene el prototipo de las funciones que se describen en el archivo `ordenamientos.c`

Contiene 5 prototipos que son:

- `selectionSort`
- `insertionSort`
- `HeapSort`
- `BuildHeap`
- `Heapify`

2.1.4 ordenamientos.c

Este archivo contiene el código de cada una de las funciones declaradas en `ordenamientos.h`, las funciones se describen de la siguiente manera:

- `selectionSort()`

La función de *selectionSort* usa dos argumentos que son un arreglo de números enteros **arreglo** y un entero **n** como segundo argumento que es la longitud del arreglo.

Al principio de la función se declaran 3 variables enteras que son **indiceMenor**, **i** y **j**.

Se crea un *for-loop* donde **i** se iguala a 0 y el cual correrá hasta el valor de la longitud del arreglo menos una unidad.

Después de esto **indiceMenor** se iguala a el valor en **i** de ese momento y después se inicia otro *for-loop* donde **j** se iguala al valor de **i + 1** y corre hasta que **j** sea menor a **n**.

Después, el programa evalúa si el valor del arreglo en el índice correspondiente a **j** es menor que el valor del arreglo correspondiente al índice del mismo arreglo en el valor dado a **indiceMenor**, en caso afirmativo entonces el valor en **indiceMenor** cambia al valor en **j** y este *for-loop* evalúa cada uno de los elementos del arreglo.

Después de que el *loop* termina, el programa evalúa si el valor en **i** no es igual al valor en **indiceMenor**, osea, si en efecto hubo un cambio en el *loop* anterior; si así fue, entonces el programa usa la función *swap()* que invierte la posición de los valores en el arreglo al valor de la posición en **i** por el arreglo en la posición **indiceMenor**.

Al finalizar el ciclo, la función imprime la leyenda *Iteración numero (i + 1)* donde **i + 1** es el numero de la iteración correspondiente y acto seguido se invoca a la función *printArray()* para que haga el *output* del arreglo modificado durante ese ciclo.

Este proceso se repite hasta que el ciclo termina, que es cuando los valores del arreglo ya deben de estar todos ordenados.

- **insertionSort()**

La función *insertionSort()* utiliza un par de argumentos que son el arreglo a ordenar **a** y el tamaño del arreglo que es una variable entera **n**.

Al momento de comenzar la función se declaran 4 variables enteras que son **i**, **j**, **k** y **aux**. La variable **i** funcionara como contador del *for-loop* que se usara para el algoritmo, la variable **aux** como variable auxiliar para depositar valores del arreglo y la variable **j** para llevar un control de las iteraciones correspondiente al *while-loop* interior.

Primero se inicia el *for-loop* igualando la variable **i** a 1 y con una condicional de que iterara hasta que **i** sea menor a **n** que corresponde al tamaño del arreglo. En cada ciclo se suma 1 a **i**.

Dentro del *loop* se deposita el valor de **i** en **j** y después se deposita el valor de el índice numero 1 de **a** en **aux**. Acto seguido se inicia un *while-loop* cuya condición para continuar el ciclo se da cuando la variable **j** sea mayor a 0 y la variable **aux** sea menor al valor en el índice **j - 1** de el arreglo **a**.

Si la condicional pasada es verdadera, entonces el valor en el índice **j - 1** de **a**, se deposita en el índice **j** de **a** y se resta 1 al valor en **j**.

Si la condicional sea falsa o una vez que la condicional sea falsa, el *while-loop* termina y el valor contenido en la variable **aux** se deposita en el índice **j** de **a**.

Básicamente lo que todo esto significa en palabras mas simples, es que el algoritmo lo que hace, es que toma un elemento del arreglo y lo compara con el anterior, si el valor del elemento anterior es mas grande entonces los cambia de lugar y esto lo hace recursivamente hacia la izquierda. De esta forma, va acomodando los elementos menores a la izquierda y los mayores hacia la derecha.

Al finalizar el ciclo, la función imprime la leyenda *Iteración numero (i)* donde **i** es el numero de la iteración correspondiente y acto seguido se invoca a la función *printArray()* para que haga el *output* del arreglo modificado durante ese ciclo.

Este proceso se repite hasta que el *for-loop* termina, que es cuando los valores del arreglo ya deben de estar todos ordenados.

- HeapSort()

Primeramente hay que notar que se declara una variable global entera **heapSize** antes de comenzar la función *heapSort()* es de tipo *void* y usa dos argumentos para trabajar que son un arreglo de enteros **A** y una variable entera **size** que contiene el tamaño del arreglo.

El algoritmo *HeapSort()* contiene ademas dos sub-funciones que complementan su algoritmo y los trataremos dentro de este punto primeramente antes de explicar como funciona *HeapSort*.

- Heapify()

La función *Heapify()* lo que hace es reordenar el arreglo ingresado en forma de árbol binario.

Usa 3 argumentos que son un apuntador **A**, una variable entera **i** y una variable entera **size**.

El apuntador **A**, apunta al arreglo que se pretende reordenar, la variable **i** contiene el índice de la mitad del arreglo y la variable **size** contiene la longitud total del arreglo.

Entrando a la función se declaran las variables enteras **l**, **r** y **largest**. La variable **l** adquiere el valor del tamaño del arreglo si este es impar y la variable **r** adquiere el valor del tamaño del arreglo en caso de que este sea par.

Después se evalúa si el valor de **l** es menor o igual al valor de **heapSize** (que anteriormente se definió como el valor en **size - 1**) y si el valor en el índice **l** en **A** es mayor que el valor en el índice **i** de **A** también. Como **l** es mayor que **i**, entonces lo que esto nos dice es que evalúa si un elemento anterior del arreglo tiene un valor menor que otro elemento mas adelante en el arreglo.

Si esto es verdadero, entonces el valor en **l** se le asigna a la variable **largest**, si no lo es, entonces el valor en **i** se le asigna a **largest**.

Después se hace lo mismo pero para la variable **r**, recordemos que la variable **r** se usa en el caso de que el arreglo sea par.

Dado esto, evalúa si la variable **largest** no es igual a la variable **i**. Si no son iguales, entonces significa que el valor en el índice de **l** es mas grande que el valor en el índice **i** por lo que se procede a usar la función *swap()* para hacer el cambio de posiciones.

Para efectos de visualización del arreglo modificado, se invoca a la función *printArray()* para que genere el *output* del arreglo modificado y se invoca recursivamente a la misma función *Heapify* para verificar que el valor que se acaba de modificar, se encuentre en la posición correcta.

– BuildHeap()

La función *BuildHeap* de lo único que se encarga es de preparar los valores necesarios para que *Heapify()* funcione adecuadamente.

Toma dos argumentos que son un apuntador **A** y una variable entera que contenga el tamaño del arreglo dado **size**.

Después de esto ocupa la variable global **heapSize** para depositar el valor en **size** menos una unidad, acto seguido declara una variable entera **i** que sirve como contador de un *for-loop* que iterara los diferentes índices invocando a la función *Heapify()* en cada una de los ciclos.

La variable **i** se iguala al tamaño del arreglo menos una unidad, dividido entre dos e itera hasta que el valor en **i** sea menor o igual a cero. En cada iteración se va restando uno a la variable **i**.

Esto hace entonces que el algoritmo vaya iterando dentro de cada uno de los índices del arreglo, desde la mitad hasta el primero, osea, va hacia la izquierda. Es por esta razón que se va restando uno del valor en **i**.

Al finalizar, la función simplemente imprime el texto "*Termino de construir el HEAP*"

Después de esta breve explicación sobre lo que las sub-funciones del algoritmo realizan, explicaremos lo que hace el algoritmo *HeapSort()*.

Al ingresar a la función *HeapSort* se invoca a la función *BuildHeap* y se le transmiten como argumentos el arreglo **A** y la variable **size**. Después de que este termina de su función, se declara una variable entera **i** para que funcione como contador dentro de un *for-loop* y se le asigna a **i** el valor en **size** menos una unidad y va reduciendo el valor en **i** en una unidad por cada iteración, hasta que llegue a cero.

Dentro de cada ciclo del *for-loop* se utiliza la función *swap()* para intercambiar el ultimo elemento en el arreglo por el primero. Esto se hace

porque este ciclo lo que hará, será el proceso de eliminar la raíz del árbol binario para arreglar los elementos en orden ascendente.

La variable **heapSize** ya había sido modificada previamente por la función *BuildHeap* y corresponde al valor de la longitud del arreglo menos una unidad. Por cada *swap()* que el ciclo ejecuta, se resta una unidad al valor en **heapSize**.

Después la función imprime la leyenda *Iteración: HS* para marcar que acaba de suceder un ciclo y ya que el arreglo principal ha sido modificado, se invoca a la función *Heapify()* para reordenar el arreglo.

Cuando el ciclo termina, la última invocación a la función *printArray* mostrará el arreglo completamente ordenado de forma ascendente.

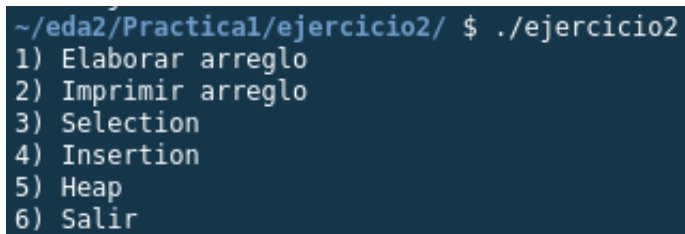
2.2 Ejercicio 2

El programa que hice incluye las bibliotecas `time.h` y `stdlib.h` que se usaran para usar la función `rand()`, `malloc()` y `realloc()`. De igual manera se incluyen las bibliotecas proporcionadas en el preámbulo del programa.

Después de ingresar a la función `main()`, se usa la función `srand(time(NULL))` para poder crear números al azar. Así mismo se declara un arreglo dinámico de enteros `arr` y se usa la función `malloc()` para darle el tamaño de uno inicialmente. De igual forma se declaran dos variables globales enteras `n` y `option`, la primera sirve para indicar el tamaño del arreglo que se desea obtener, la segunda para depositar la opción que se elija en el menu del usuario.

El programa costa de dos *do-while loops*. En el primero se integra el programa completo que no rompe el ciclo hasta que se ingresa la opción 6 que corresponde a la finalización del programa. El segundo encierra el menu del usuario que se repite mientras el usuario no seleccione una opción valida del menu.

Las opciones a escoger en el menu son *Elaborar arreglo*, *Imprimir arreglo*, *Selection*, *Insertion*, *Heap* y *Salir*.



```
~/.eda2/Practical/ejercicio2/ $ ./ejercicio2
1) Elaborar arreglo
2) Imprimir arreglo
3) Selection
4) Insertion
5) Heap
6) Salir
```

Figure 1: Menu principal

Al momento de seleccionar cualquier opción, el programa utiliza la función `switch()` para ejecutar el código indicado por el usuario.

2.2.1 Opción 1: Elaborar arreglo

La primera opción que es la de elaborar arreglo, pide inicialmente el numero de elementos a elegir dado esto, utiliza la función `realloc()` para crear espacio en el arreglo dinámico en función de los elementos requeridos por el usuario.

Después entra en un *for-loop* que itera desde `0` hasta `n - 1` veces el espacio demandado y en cada ciclo se genera un numero al azar con la función `rand()` y la cual se obtiene el modulo entre 100, lo que nos proporciona los últimos dos dígitos del numero creado ya que `rand()` genera enteros de 8 dígitos

Cada uno de estos números se acomoda de acuerdo a al indice del contador `i` en el arreglo dinámico


```
~/eda2/Practical/ejercicio2/ $ ./ejercicio2
1) Elaborar arreglo
2) Imprimir arreglo
3) Selection
4) Insertion
5) Heap
6) Salir
1
No. elementos: 20
```

Figure 2: Seleccionando opción 1 e ingresando numero de elementos deseados.

2.2.2 Opción 2: Imprimir arreglo

La opción 2 simplemente imprime el arreglo que se acaba de crear. Utiliza un *for-loop* para que de igual manera corre desde **0** hasta **n - 1** y simplemente utiliza la función *printf()* dentro de cada iteración para mostrar en pantalla cada elemento dentro de la variable **arr**.

```
1) Elaborar arreglo
2) Imprimir arreglo
3) Selection
4) Insertion
5) Heap
6) Salir
2
24 63 21 47 91 12 31 78 34 2 95 32 12 43 74 67 22 89 25 42
```

Figure 3: Arreglo impreso después de seleccionar opción 2

2.2.3 Opción 3: Selection Sort

Esta opción ejecuta la función *selectionSort()* de la biblioteca **ordenamientos.c**. Al momento de ejecutarla toma como argumentos el arreglo dinámico creado **arr** así como la variable **n** que contiene el valor de la cantidad de elementos dentro del arreglo.

La función itera **n - 1** veces hasta ordenar el arreglo que originalmente se encontraba en desorden.

```
Iteracion numero 1
2 63 21 47 91 12 31 78 34 24 95 32 12 43 74 67 22 89 25 42

Iteracion numero 2
2 12 21 47 91 63 31 78 34 24 95 32 12 43 74 67 22 89 25 42

Iteracion numero 3
2 12 12 47 91 63 31 78 34 24 95 32 21 43 74 67 22 89 25 42

Iteracion numero 4
2 12 12 21 91 63 31 78 34 24 95 32 47 43 74 67 22 89 25 42

Iteracion numero 5
2 12 12 21 22 63 31 78 34 24 95 32 47 43 74 67 91 89 25 42

Iteracion numero 6
2 12 12 21 22 24 31 78 34 63 95 32 47 43 74 67 91 89 25 42

Iteracion numero 7
2 12 12 21 22 24 25 78 34 63 95 32 47 43 74 67 91 89 31 42

Iteracion numero 8
2 12 12 21 22 24 25 31 34 63 95 32 47 43 74 67 91 89 78 42

Iteracion numero 9
2 12 12 21 22 24 25 31 32 63 95 34 47 43 74 67 91 89 78 42

Iteracion numero 10
2 12 12 21 22 24 25 31 32 34 95 63 47 43 74 67 91 89 78 42

Iteracion numero 11
2 12 12 21 22 24 25 31 32 34 42 63 47 43 74 67 91 89 78 95

Iteracion numero 12
2 12 12 21 22 24 25 31 32 34 42 43 47 63 74 67 91 89 78 95
```

Figure 4: Se muestra como se modifica el arreglo en los primeros 12 ciclos.

```
Iteracion numero 13
2 12 12 21 22 24 25 31 32 34 42 43 47 63 74 67 91 89 78 95

Iteracion numero 14
2 12 12 21 22 24 25 31 32 34 42 43 47 63 74 67 91 89 78 95

Iteracion numero 15
2 12 12 21 22 24 25 31 32 34 42 43 47 63 67 74 91 89 78 95

Iteracion numero 16
2 12 12 21 22 24 25 31 32 34 42 43 47 63 67 74 91 89 78 95

Iteracion numero 17
2 12 12 21 22 24 25 31 32 34 42 43 47 63 67 74 78 89 91 95

Iteracion numero 18
2 12 12 21 22 24 25 31 32 34 42 43 47 63 67 74 78 89 91 95

Iteracion numero 19
2 12 12 21 22 24 25 31 32 34 42 43 47 63 67 74 78 89 91 95
```

Figure 5: Se muestra como se modifica el arreglo en últimos ciclos.

2.2.4 Opcion 4: Insertion Sort

La opción numero 4 corresponde al *insertionSort()* que al igual que el *selectionSort* usa una de las funciones alojadas en la biblioteca `utilidades.h` para imprimir en pantalla cada una de las modificaciones del arreglo que se pretende ordenar.

Modificamos el arreglo pasado para crear uno nuevo con la misma cantidad de elementos que el anterior y poder visualizar los ciclos de *selectionSort* que de igual forma corre desde **0** hasta **n - 1** iteraciones por lo cual nos da de igual forma 20 arreglos impresos.

```
No. elementos: 20
1) Elaborar arreglo
2) Imprimir arreglo
3) Selection
4) Insertion
5) Heap
6) Salir
2
64 68 39 17 54 21 38 24 57 19 70 33 34 43 81 26 55 12 4 90
```

Figure 6: Creando nuevo arreglo e imprimiendo sus valores

Después de esto, procedemos a seleccionar la opción 4 y nos muestra el programa como se va modificando el arreglo en cada ciclo del algoritmo.

```
1) Elaborar arreglo
2) Imprimir arreglo
3) Selection
4) Insertion
5) Heap
6) Salir
4

Iteracion numero 1
64 68 39 17 54 21 38 24 57 19 70 33 34 43 81 26 55 12 4 90

Iteracion numero 2
39 64 68 17 54 21 38 24 57 19 70 33 34 43 81 26 55 12 4 90

Iteracion numero 3
17 39 64 68 54 21 38 24 57 19 70 33 34 43 81 26 55 12 4 90

Iteracion numero 4
17 39 54 64 68 21 38 24 57 19 70 33 34 43 81 26 55 12 4 90

Iteracion numero 5
17 21 39 54 64 68 38 24 57 19 70 33 34 43 81 26 55 12 4 90

Iteracion numero 6
17 21 38 39 54 64 68 24 57 19 70 33 34 43 81 26 55 12 4 90

Iteracion numero 7
17 21 24 38 39 54 64 68 57 19 70 33 34 43 81 26 55 12 4 90

Iteracion numero 8
17 21 24 38 39 54 57 64 68 19 70 33 34 43 81 26 55 12 4 90
```

Figure 7: Primeras 8 iteraciones de la opción 4

```
Iteracion numero 9
17 19 21 24 38 39 54 57 64 68 70 33 34 43 81 26 55 12 4 90

Iteracion numero 10
17 19 21 24 38 39 54 57 64 68 70 33 34 43 81 26 55 12 4 90

Iteracion numero 11
17 19 21 24 33 38 39 54 57 64 68 70 34 43 81 26 55 12 4 90

Iteracion numero 12
17 19 21 24 33 34 38 39 54 57 64 68 70 43 81 26 55 12 4 90

Iteracion numero 13
17 19 21 24 33 34 38 39 43 54 57 64 68 70 81 26 55 12 4 90

Iteracion numero 14
17 19 21 24 33 34 38 39 43 54 57 64 68 70 81 26 55 12 4 90

Iteracion numero 15
17 19 21 24 26 33 34 38 39 43 54 57 64 68 70 81 55 12 4 90

Iteracion numero 16
17 19 21 24 26 33 34 38 39 43 54 55 57 64 68 70 81 12 4 90

Iteracion numero 17
12 17 19 21 24 26 33 34 38 39 43 54 55 57 64 68 70 81 4 90

Iteracion numero 18
4 12 17 19 21 24 26 33 34 38 39 43 54 55 57 64 68 70 81 90

Iteracion numero 19
4 12 17 19 21 24 26 33 34 38 39 43 54 55 57 64 68 70 81 90
```

Figure 8: Ultimas 11 iteraciones de la opción 4

2.2.5 Opcion 5: Heap Sort

Repetimos la misma dinámica que con la opción 4, creamos un nuevo arreglo para apreciar el funcionamiento de *heapSort()* y acto seguido procedemos a correr el algoritmo seleccionando la opción 5 de el programa.

Se puede como el algoritmo hace uso primeramente de la función *buildHeap* y una vez terminado comienza a remover la raíz del HEAP hasta que el arreglo queda completamente ordenado.

```
No. elementos: 20

1) Elaborar arreglo
2) Imprimir arreglo
3) Selection
4) Insertion
5) Heap
6) Salir
2

66 99 22 78 94 48 98 68 89 75 10 53 43 1 23 49 22 13 73 31
```

Figure 9: Creando nuevo arreglo para HeapSort

```

1) Elaborar arreglo
2) Imprimir arreglo
3) Selection
4) Insertion
5) Heap
6) Salir
5

66 99 22 78 94 53 98 68 89 75 10 48 43 1 23 49 22 13 73 31
66 99 22 89 94 53 98 68 78 75 10 48 43 1 23 49 22 13 73 31
66 99 98 89 94 53 22 68 78 75 10 48 43 1 23 49 22 13 73 31
66 99 98 89 94 53 23 68 78 75 10 48 43 1 22 49 22 13 73 31
99 66 98 89 94 53 23 68 78 75 10 48 43 1 22 49 22 13 73 31
99 94 98 89 66 53 23 68 78 75 10 48 43 1 22 49 22 13 73 31
99 94 98 89 75 53 23 68 78 66 10 48 43 1 22 49 22 13 73 31
Termin de construir el HEAP
Iteracion HS:
31 94 98 89 75 53 23 68 78 66 10 48 43 1 22 49 22 13 73 99
98 94 31 89 75 53 23 68 78 66 10 48 43 1 22 49 22 13 73 99
98 94 53 89 75 31 23 68 78 66 10 48 43 1 22 49 22 13 73 99
98 94 53 89 75 48 23 68 78 66 10 31 43 1 22 49 22 13 73 99
Iteracion HS:
73 94 53 89 75 48 23 68 78 66 10 31 43 1 22 49 22 13 98 99
94 73 53 89 75 48 23 68 78 66 10 31 43 1 22 49 22 13 98 99
94 89 53 73 75 48 23 68 78 66 10 31 43 1 22 49 22 13 98 99
94 89 53 78 75 48 23 68 73 66 10 31 43 1 22 49 22 13 98 99
Iteracion HS:
13 89 53 78 75 48 23 68 73 66 10 31 43 1 22 49 22 94 98 99
89 13 53 78 75 48 23 68 73 66 10 31 43 1 22 49 22 94 98 99
89 78 53 13 75 48 23 68 73 66 10 31 43 1 22 49 22 94 98 99
89 78 53 73 75 48 23 68 13 66 10 31 43 1 22 49 22 94 98 99

```

Figure 10: Se crea el heap y después se comienza a remover las raíces


```

Iteracion HS:
49 75 53 73 66 48 23 68 13 22 10 31 43 1 22 78 89 94 98 99
75 49 53 73 66 48 23 68 13 22 10 31 43 1 22 78 89 94 98 99
75 73 53 49 66 48 23 68 13 22 10 31 43 1 22 78 89 94 98 99
75 73 53 68 66 48 23 49 13 22 10 31 43 1 22 78 89 94 98 99
Iteracion HS:
22 73 53 68 66 48 23 49 13 22 10 31 43 1 75 78 89 94 98 99
73 22 53 68 66 48 23 49 13 22 10 31 43 1 75 78 89 94 98 99
73 68 53 22 66 48 23 49 13 22 10 31 43 1 75 78 89 94 98 99
73 68 53 49 66 48 23 22 13 22 10 31 43 1 75 78 89 94 98 99
Iteracion HS:
1 68 53 49 66 48 23 22 13 22 10 31 43 73 75 78 89 94 98 99
68 1 53 49 66 48 23 22 13 22 10 31 43 73 75 78 89 94 98 99
68 66 53 49 1 48 23 22 13 22 10 31 43 73 75 78 89 94 98 99
68 66 53 49 22 48 23 22 13 1 10 31 43 73 75 78 89 94 98 99
Iteracion HS:
43 66 53 49 22 48 23 22 13 1 10 31 68 73 75 78 89 94 98 99
66 43 53 49 22 48 23 22 13 1 10 31 68 73 75 78 89 94 98 99
66 49 53 43 22 48 23 22 13 1 10 31 68 73 75 78 89 94 98 99
Iteracion HS:
31 49 53 43 22 48 23 22 13 1 10 66 68 73 75 78 89 94 98 99
53 49 31 43 22 48 23 22 13 1 10 66 68 73 75 78 89 94 98 99
53 49 48 43 22 31 23 22 13 1 10 66 68 73 75 78 89 94 98 99
Iteracion HS:
10 49 48 43 22 31 23 22 13 1 53 66 68 73 75 78 89 94 98 99
49 10 48 43 22 31 23 22 13 1 53 66 68 73 75 78 89 94 98 99
49 43 48 10 22 31 23 22 13 1 53 66 68 73 75 78 89 94 98 99
49 43 48 22 22 31 23 10 13 1 53 66 68 73 75 78 89 94 98 99
Iteracion HS:
1 43 48 22 22 31 23 10 13 49 53 66 68 73 75 78 89 94 98 99
48 43 1 22 22 31 23 10 13 49 53 66 68 73 75 78 89 94 98 99
48 43 31 22 22 1 23 10 13 49 53 66 68 73 75 78 89 94 98 99

```

Figure 11: Continua iterando el algoritmo

```

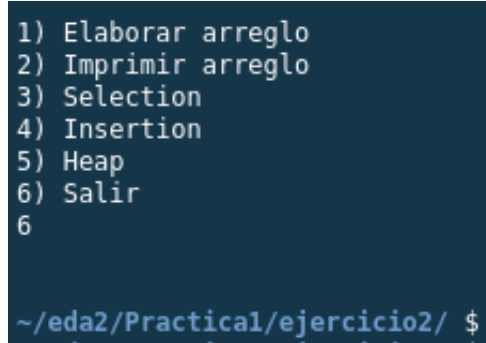
Iteracion HS:
13 43 31 22 22 1 23 10 48 49 53 66 68 73 75 78 89 94 98 99
43 13 31 22 22 1 23 10 48 49 53 66 68 73 75 78 89 94 98 99
43 22 31 13 22 1 23 10 48 49 53 66 68 73 75 78 89 94 98 99
Iteracion HS:
10 22 31 13 22 1 23 43 48 49 53 66 68 73 75 78 89 94 98 99
31 22 10 13 22 1 23 43 48 49 53 66 68 73 75 78 89 94 98 99
31 22 23 13 22 1 10 43 48 49 53 66 68 73 75 78 89 94 98 99
Iteracion HS:
10 22 23 13 22 1 31 43 48 49 53 66 68 73 75 78 89 94 98 99
23 22 10 13 22 1 31 43 48 49 53 66 68 73 75 78 89 94 98 99
Iteracion HS:
1 22 10 13 22 23 31 43 48 49 53 66 68 73 75 78 89 94 98 99
22 1 10 13 22 23 31 43 48 49 53 66 68 73 75 78 89 94 98 99
22 22 10 13 1 23 31 43 48 49 53 66 68 73 75 78 89 94 98 99
Iteracion HS:
1 22 10 13 22 23 31 43 48 49 53 66 68 73 75 78 89 94 98 99
22 1 10 13 22 23 31 43 48 49 53 66 68 73 75 78 89 94 98 99
22 13 10 1 22 23 31 43 48 49 53 66 68 73 75 78 89 94 98 99
Iteracion HS:
1 13 10 22 22 23 31 43 48 49 53 66 68 73 75 78 89 94 98 99
13 1 10 22 22 23 31 43 48 49 53 66 68 73 75 78 89 94 98 99
Iteracion HS:
10 1 13 22 22 23 31 43 48 49 53 66 68 73 75 78 89 94 98 99
Iteracion HS:
1 10 13 22 22 23 31 43 48 49 53 66 68 73 75 78 89 94 98 99

```

Figure 12: Finaliza el algoritmo y se aprecia al final la lista ordenada ascendentemente

2.2.6 Opcion 6: Salir

Para salir del programa, solo se escoge la opción 6 y automáticamente se rompen los ciclos y se finaliza el programa.



```
1) Elaborar arreglo
2) Imprimir arreglo
3) Selection
4) Insertion
5) Heap
6) Salir
6

~/eda2/Practical/ejercicio2/ $
```

Figure 13: Finalización del programa

2.3 Ejercicio 3

Para este ejercicio modifique el programa de tal forma que se automatizaron los tamaños de los arreglos de acuerdo a lo requerido por el ejercicio y se hicieron pruebas con 3 formas de arreglos distintas por cada uno de los algoritmos de ordenamientos.

Se probaron 5 tamaños distintos de arreglos:

- 10
- 100
- 500
- 1000
- 2000

Y se usaron 3 tipos de arreglos para cada uno de esos tamaños:

- Ascendente
- Descendente
- Aleatorio

Se ejecutaron 45 pruebas en total y esta fue la tabla general de resultados:

	Cantidad de Elementos en el Arreglo				
Selection	10	100	500	1000	2000
Ascendete	45	4950	124750	499500	1999000
Descendente	45	4950	124750	499500	1999000
Aleatorio	45	4950	124750	499500	1999000
Insertion	10	100	500	1000	2000
Ascendete	9	99	499	999	1999
Descendente	45	4950	124750	499500	1999000
Aleatorio	18	2373	64213	253279	999661
Heap	10	100	500	1000	2000
Aleatorio	14	149	749	1499	2999
Descendente	14	149	749	1499	2999
Aleatorio	14	149	749	1499	2999

Figure 14: Total de resultados

2.3.1 Selection Sort:

Para el algoritmo de *Selection Sort* se puede apreciar como debido a que su tiempo de complejidad es de $O(n^2)$ para todos los casos, los puntos de la grafica son constantes independientemente del tamaño del arreglo o de la forma en que se encuentre arreglado. De hecho en la grafica no se alcanza a apreciar los puntos rojos y amarillos puesto que son los mismos que los azules.

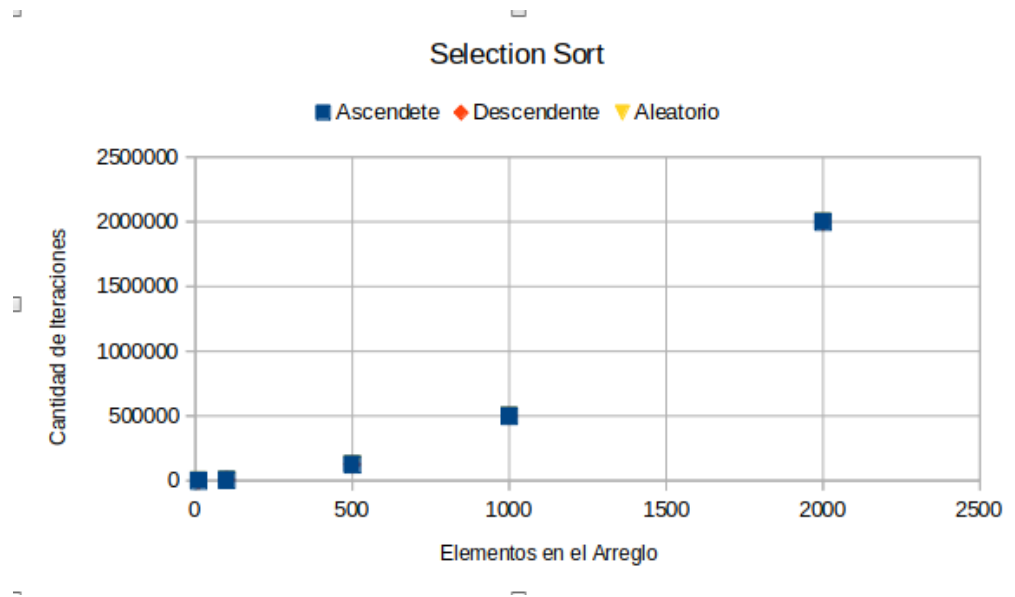


Figure 15: Grafica de Selection Sort

2.3.2 Insertion Sort:

El algoritmo de *Insertion Sort* a diferencia del anterior, tiene 3 tipos diferentes de complejidad temporal dependiendo de la forma en la que los arreglos se encuentren ordenados.

Estas serian:

- Para el peor caso: $O(n^2)$
- Para el caso promedio: $\Theta(n^2)$
- Para el mejor caso: $\Omega(n^2)$

Podemos notar que para el caso de las listas aleatorias, no necesariamente se cumple esto sino es mas un caso promedio, puesto que no están en orden descendente lo que implica que el algoritmo tendría que hacer iteraciones a través de toda la lista.

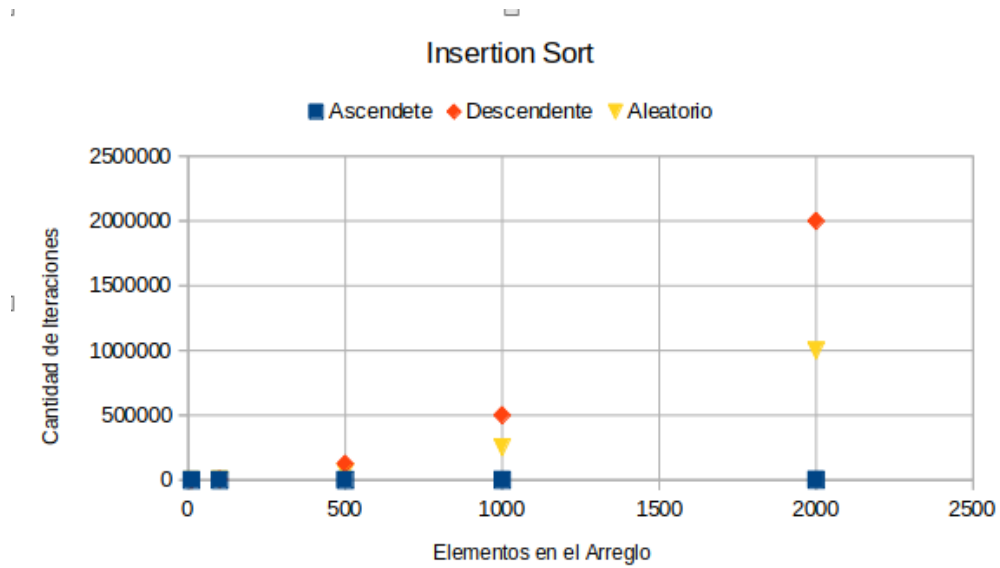


Figure 16: Grafica de Insertion Sort

2.3.3 Heap Sort:

Por ultimo, es notable como la grafica de *Heap Sort* cumple con el tiempo logarítmico de $O(n \log n)$ en la grafica que es casi una linea recta. Es un algoritmo que a pesar de no utilizar una estructura de datos lineal para ejecutarse, en realidad la forma en la que se ejecuta en el código es mas lineal de lo que parece, pues no usa muchos loops y su función principal *Heapify* simplemente se reduce a hacer *swaps* en el arreglo que utiliza que es lo que le ahorra mucho tiempo y memoria.

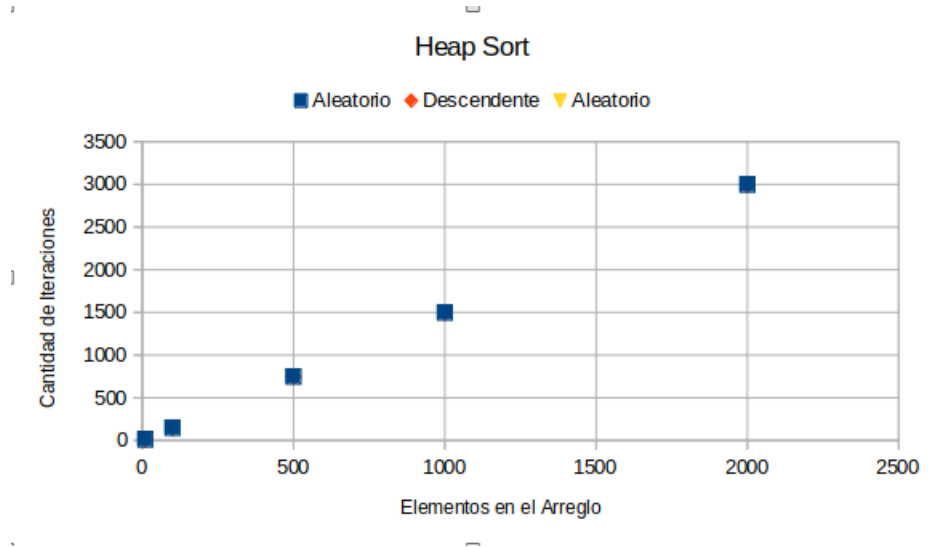


Figure 17: Grafica de Heap Sort

2.3.4 Capturas de Ejecucion

Aquí solo se incluyen las capturas de la ejecución del programa como lo pide el ejercicio.

```
~/eda2/practical/ejercicio3/ $ ./ejercicio3
Tipo: Ascendente, Elementos: 10
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 45

Tipo: Descendente, Elementos: 10
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 45

Tipo: Aleatorio, Elementos: 10
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 45

Tipo: Ascendente, Elementos: 100
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 4950

Tipo: Descendente, Elementos: 100
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 4950

Tipo: Aleatorio, Elementos: 100
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 4950

Tipo: Ascendente, Elementos: 500
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 124750

Tipo: Descendente, Elementos: 500
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 124750

Tipo: Aleatorio, Elementos: 500
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 124750

Tipo: Ascendente, Elementos: 1000
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 499500

Tipo: Descendente, Elementos: 1000
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 499500

Tipo: Aleatorio, Elementos: 1000
  Selection Sort:
    Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 499500
```

Figure 18: Output de programa automatizado


```

Tipo: Ascendente, Elementos: 2000
Selection Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 1999000

Tipo: Descendente, Elementos: 2000
Selection Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 1999000

Tipo: Aleatorio, Elementos: 2000
Selection Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...Total: 1999000

Tipo: Ascendente, Elementos: 10
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 9

Tipo: Descendente, Elementos: 10
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 45

Tipo: Aleatorio, Elementos: 10
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 28

Tipo: Ascendente, Elementos: 100
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 99

Tipo: Descendente, Elementos: 100
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 4950

Tipo: Aleatorio, Elementos: 100
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 2549

Tipo: Ascendente, Elementos: 500
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 499

Tipo: Descendente, Elementos: 500
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 124750

```

Figure 19: Output de programa automatizado

```

Tipo: Aleatorio, Elementos: 500
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 62960

Tipo: Ascendente, Elementos: 1000
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 999

Tipo: Descendente, Elementos: 1000
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 499500

Tipo: Aleatorio, Elementos: 1000
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 244709

Tipo: Ascendente, Elementos: 2000
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 1999

Tipo: Descendente, Elementos: 2000
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 1999000

Tipo: Aleatorio, Elementos: 2000
Insertion Sort:
Big O:  $n^2 \rightarrow (n^2 - n)/2$ 
Corriendo...
Total: 988700

Tipo: Ascendente, Elementos: 10
Heap Sort:
Big O:  $n \log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 5, HeapSort: 9, Total = 14

Tipo: Descendente, Elementos: 10
Heap Sort:
Big O:  $n \log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 5, HeapSort: 9, Total = 14

Tipo: Aleatorio, Elementos: 10
Heap Sort:
Big O:  $n \log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 5, HeapSort: 9, Total = 14

```

Figure 20: Output de programa automatizado

```

Tipo: Ascendente, Elementos: 100
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 50, HeapSort: 99, Total = 149

Tipo: Descendente, Elementos: 100
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 50, HeapSort: 99, Total = 149

Tipo: Aleatorio, Elementos: 100
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 50, HeapSort: 99, Total = 149

Tipo: Ascendente, Elementos: 500
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 250, HeapSort: 499, Total = 749

Tipo: Descendente, Elementos: 500
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 250, HeapSort: 499, Total = 749

Tipo: Ascendente, Elementos: 1000
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 500, HeapSort: 999, Total = 1499

Tipo: Descendente, Elementos: 1000
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 500, HeapSort: 999, Total = 1499

Tipo: Aleatorio, Elementos: 1000
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 500, HeapSort: 999, Total = 1499

Tipo: Ascendente, Elementos: 2000
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 1000, HeapSort: 1999, Total = 2999

Tipo: Descendente, Elementos: 2000
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 1000, HeapSort: 1999, Total = 2999

```

Figure 21: Output de programa automatizado

```

Tipo: Aleatorio, Elementos: 2000
Heap Sort:
Big O:  $n\log(n) \rightarrow (3n - 1)/2$ 
Corriendo...
BuildHeap: 1000, HeapSort: 1999, Total = 2999

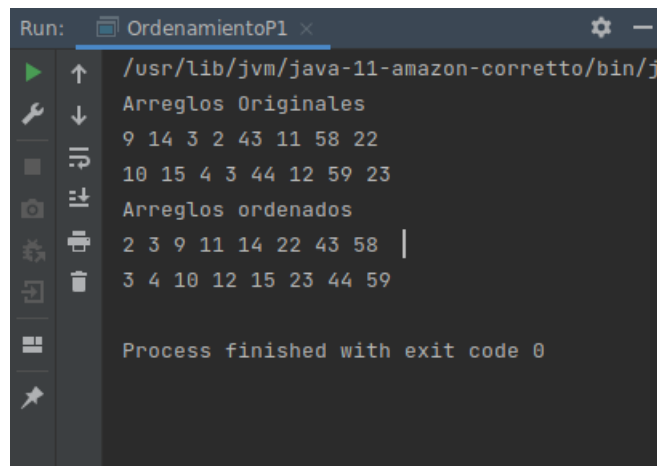
~/eda2/practical1/ejercicio3/ $ █

```

Figure 22: Output de programa automatizado

2.4 Ejercicio 4

2.4.1 Ejecución OrdenamientoP1



```
Run: OrdenamientoP1 x
/usr/lib/jvm/java-11-amazon-corretto/bin/j...
Arreglos Originales
9 14 3 2 43 11 58 22
10 15 4 3 44 12 59 23
Arreglos ordenados
2 3 9 11 14 22 43 58 |
3 4 10 12 15 23 44 59

Process finished with exit code 0
```

Figure 23: Ejecución de OrdenamientoP1 en IntelliJIDE

2.4.2 Revisión del Código

Aun soy algo nuevo en esto de Java y sinceramente no entiendo muy bien como funciona el código que se encuentra en el programa, pero me recuerda mucho a los tipos de datos *struct* que usamos en la clase de Estructura de Datos 1.

Lo que veo en el archivo es que hay 3 archivos de clases y el archivo `OrdenamientoP1.java` es el que contiene a la función *main*, y un par de arreglos.

Después la clase, imprime la leyenda *Arreglos Originales* y después de esa línea, lo que noto es que el programa a través de `Utilerias.imprimirArreglo()`, lo que hace es llamar al otro archivo donde se encuentra la clase *Utilerias* y con un punto invoca a una función dentro de la clase que es la de `imprimirArreglo()` la cual toma un argumento de un arreglo de números enteros y lo que veo es que al parecer en Java, se puede copiar un arreglo con un signo de `=` como en *Python*.

De hecho es muy similar el programa a cuando en *Python* se importan bibliotecas completas y para invocar sus funciones se usa el nombre de la biblioteca, seguido de un punto más el nombre de la función que se desea ejecutar.

Veo como de hecho los archivos-clases tienen distintos tipos de funciones dentro de sí como por ejemplo, las clases de *Inserción* y *Selection* que contienen los algoritmos utilizados y que a su vez, de igual manera, invocan a las funciones incluidas en la clase *Utilerias* que en ese caso es el método `intercambiar()`.

Por otro lado parece ser que al llamar al algoritmo de selección, lo que en realidad sucede es que se crea una nueva **instancia** de la clase *Selection* que se llama igual, pero con minúscula al principio y a través de esta instancia se llama al algoritmo. Me imagino que esta instancia no se encuentra en el archivo de la clase como tal, sino ya en la memoria RAM de la computadora.

Para finalizar se muestran los arreglos ordenados por medio del archivo-clase *Utilerias* y el método *imprimirArreglo()*.

3 Conclusiones

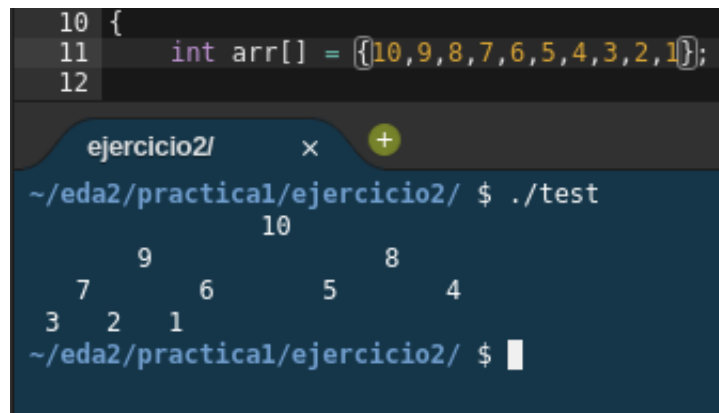
Esta practica estuvo muy larga pero me gusto para ser la primera practica de la clase.

Ya había dejado de trabajar con C y la practica afortunadamente me hizo recordar bastantes cosas que ya hace tiempo que no usaba y pude programar un poco, los algoritmos que usamos y experimentar con ellos.

Los algoritmos me parecen interesantes, pero me gustaría indagar un poco mas en ellos, pues no creo que unicamente sirvan para ordenar números, sino que supongo que también deben de ser útiles cosas mas complejas como ordenar grandes cantidades de archivos, cadenas, palabras, etc. Me gustaría mas indagar en esa parte y saber como es que se utilizan en los programas que utilizamos a diario.

El algoritmo de ordenamiento que mas me gusto fue *HeapSort* porque se me hizo muy interesante este tipo de estructura no-lineal con la que trabaja y que adema es muy sencilla. Al principio me parecía que las estructuras de árbol, eran mas complicadas, pero entendiéndolas me di cuenta que la verdad son muy sencillas, inclusive podría decir que hasta mas sencillas que las demás estructuras de datos.

De hecho, me di a la tarea de tratar de crear un algoritmo para imprimir los arreglos en forma de árbol y me quedo mas o menos así



```
10 {
11   int arr[] = {10,9,8,7,6,5,4,3,2,1};
12
ejercicio2/ × +
~/eda2/practical/ejercicio2/ $ ./test
      10
     9  8
    7  6  5  4
   3  2  1
~/eda2/practical/ejercicio2/ $
```

Figure 24: Impresión automática de arreglo en forma de árbol

La verdad es que mi algoritmo aun no es muy eficiente, pero me gusto experimentar con la forma de esa estructura.

Otra cosa que me agrado bastante fue que este es el primer documento que

hago en L^AT_EXy aprendí mucho como usar el programa, mucha de la sintaxis del *typeset*, insertar imágenes, formulas, accidentario, etc, ademas de que no sabia que también fue inventado por Donald Knut. (Quien fuera como el!)

Por ultimo es la primera vez que experimento con *Java*, pero creo que no me fue tan complicado ver como funciona la lógica del programa, creo que ya me va quedando claro que es un **objeto** y como se comportan. Recientemente adquirí el curso de Tim Buchalka para aprender a usarlo bien y es algo que he estado haciendo estos días Seguiré practicando para volverme mejor.

Gracias por leer mi practica!