



## Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

### Laboratorios de computación salas A y B

*Profesor:* TISTA GARCÍA EDGAR

*Asignatura:* ESTRUCTURA DE DATOS Y ALGORITMOS II

*Grupo:* 5

*Número de Práctica(s):* Guía práctica de estudio 2: Algoritmos de Ordenamiento de Datos II

*Integrante(s):* MURRIETA VILLEGAS ALFONSO

*Núm. De Equipo de  
cómputo empleado :* 32

*Semestre :* 2019 - 1

*Fecha de entrega:* 21 DE AGOSTO DE 2018

*Observaciones:*

**CALIFICACIÓN:** \_\_\_\_\_

## ALGORITMOS DE ORDENAMIENTO DE DATOS. PARTE II

### INTRODUCCIÓN

En el mundo de la programación existen ciertas características que hacen a éste metódico, para poder resolver cualquier problema siempre se debe plantear un algoritmo el cual se encargará de dar ya sea a primera instancia una solución cualquiera, o la mejor solución, posteriormente este se lleva a través de un lenguaje de programación para poder representarlo en lo que conocemos como programa.

Pero antes, se le conoce como **algoritmo** a un conjunto finito de instrucciones libre de ambigüedades que sirven para realizar una tarea o acción específica. Para hacer un algoritmo debe tener algunas características particulares como son:

- 1) Debe ser exacta la descripción de las actividades a realizar.
- 2) Debe excluir cualquier ambigüedad
- 3) Debe ser invariante en el tiempo como en lugar.

Por otro lado, entendemos el concepto de “ordenar” como la acción de reagrupar o reorganizar un conjunto de datos u objetos en una secuencia específica, la cual nos permite manejar información de manera más eficiente.

Dicho lo anterior, podemos definir que un algoritmo de ordenamiento es aquel que cumple con factores como la cantidad de memoria o el entorno de software para reacomodar colecciones de elementos de tal forma que todos sus elementos cumplan con una lógica respecto al criterio global ya sea ascendente o descendente.

Por último, las condiciones que deben cumplirse para que se pueda considerar un algoritmo de ordenamiento de datos son las siguientes:

- 1) **Verificación:** El algoritmo siempre devuelve los elementos en orden correcto.
- 2) **Tiempo de ejecución:** Para calcular el tiempo es importante contabilizar comparaciones, intercambios, o acceso al arreglo – lista.
- 3) **Memoria:** Se debe considerar la cantidad de memoria extra necesaria para realizar operaciones.

### *Clasificación de algoritmos de ordenamientos*

Pese existen muchos criterios para clasificar los algoritmos de ordenamiento de datos, estos se pueden clasificar en 4 grupos:

#### *1) Algoritmos de inserción:*

Se consideran un elemento a la vez y cada elemento insertado en la posición apropiada con respecto a los demás elementos que ya han sido ordenados.

Ejemplos: Shell sort, inserción binaria y hashing.

## 2) Algoritmos de intercambio:

En estos algoritmos se trabaja con parejas de elementos que se van comparando e intercambiando si no están en el orden adecuado. El proceso se realiza hasta que se han revisado todos los elementos del conjunto a ordenar.

Ejemplos: Bubble Sort y Quicksort.

## 3) Algoritmos de selección:

En estos algoritmos se selecciona el elemento mínimo o el máximo de todo el conjunto a ordenar y se coloca en la posición apropiada. Esta selección se realiza con todos los elementos restantes del conjunto.

## 4) Algoritmos de enumeración:

Se compara cada elemento con todos los demás y se determina cuántos son menores que él. La información del conteo para cada elemento indicará su posición de ordenamiento. Otras formas de clasificar el tipo este tipo de algoritmos es a través de la memoria usada como es el caso de: Ordenamiento interno y Ordenamiento externo. A su vez también se pueden clasificar por su estructura: Directo (Sobre el mismo elemento) e Indirecto (A través de otros elementos).

En el caso del ordenamiento interno podemos destacar que se caracteriza por el rápido acceso aleatorio a la memoria, mientras que en el ordenamiento externo se necesita recursos de la memoria secundaria.

## OBJETIVOS DE LA PRÁCTICA

- El estudiante identificará la estructura de algoritmos de ordenamiento *Selectionsort*, *Insertionsort* y *Heapsort*.

## DESARROLLO

### Actividad 1. Análisis del manual de prácticas

Es esta práctica los 3 métodos que se lleva se llevarán a cabo serán SelectionSort, Insertionsort y Heapsort.

#### 1.1 Selection Sort

##### Generalización

Selection Sort es una de las formas más sencillas de ordenar datos, consiste en seleccionar el valor más pequeño o más grande del arreglo o lista e intercambiarlo con el primer dato, al checar los datos se realizarán los cambios cuando se determinen el mayor (descendente) o menor (ascendente).

El procedimiento se repetirá con los siguientes datos faltantes hasta terminar con el arreglo o lista.

Por otro lado, como todo algoritmo de ordenamiento tiene tanto ventajas como desventajas, por un lado, sus desventajas son que el algoritmo no toma en cuenta el orden de la entrada de los datos, provocando que pueda tardar al revisar las comparaciones entre el menor dato actual y el primer dato.

NOTA: Toma el mismo tiempo ordenar una lista ya sea que esté o no arreglada.

## Ejemplificación

A continuación, se muestra el pseudocódigo de Selection sort hecho en OneNote:

```
ordenamientoSeleccion(lista)
n ← longitud(lista)
para i ← 1 hasta n-1
    mínimo ← 1;
    para j ← i+1 hasta n
        si lista[j] < lista[mínimo]
            mínimo ← j
    intercambiar(lista[i], lista[mínimo])
regresar(lista)
```

Imagen 1: Algoritmo de Selection Sort

## Análisis de complejidad

En el caso de Selection Sort tenemos un algoritmo de ordenamiento de datos que realmente no importa si tenemos el peor o mejor caso debido a la forma en que este evalúa a la lista o arreglo, por lo que siempre tenemos que su grado de complejidad es  $O(n^2)$

## Actividad (Python) (EXTRA)

Como parte extra (No hay nada de este algoritmo en el manual) decidí llevar el algoritmo de Selection Sort en Python solo como práctica de programación y como pasatiempo. A continuación, se muestra la salida ordenada de la siguiente lista = [54,26,93,17,77,31,44,55,20]:

```
In [2]: runfile('C:/Users/pon_c/OneDrive/Documentos/
Universidad/3° Semestre/EDA II/Prácticas_LAB/2_practice/
Códigos_Manual/selectionsort.py', wdir='C:/Users/pon_c/
OneDrive/Documentos/Universidad/3° Semestre/EDA II/
Prácticas_LAB/2_practice/Códigos_Manual')
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Imagen 2: Salida ordenada de la lista previa llevada a cabo en Selection Sort.

## 1.2 Heap Sort

### Previo

Para poder comprender este apartado es necesario saber con antelación que es un “HEAP”. Heap es un árbol binario donde cualquiera de sus nodos siempre tiene un nodo padre o dicho de otra forma el de mayor valor (Ver imagen 3).

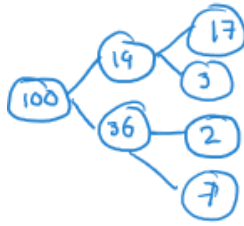


Imagen 3: Esquema de un Heap

## Generalización

Se encuentra entre los mejores algoritmos de ordenamiento más eficientes, consiste en almacenar los datos de un arreglo o lista en un Heap. Este algoritmo depende del tipo de ordenamiento como se guardarán los datos, en el caso de que el nodo padre sea mayor a sus 2 nodos hijos se denotará como montículo máximo y si es menor a sus hijos se llamará montículo mínimo.

Para poder llevar a cabo este algoritmo de ordenamiento de datos se da principalmente en 2 dos pasos:

### 1) Construcción:

Se inserta un nuevo elemento en la primera posición disponible, posteriormente se verifica si el valor es mayor al del padre en tal caso se intercambian los elementos, en caso contrario se finaliza la acción.

NOTA: Así se realiza sucesivamente hasta terminar con todos los elementos

### 2) Eliminación

Se reemplaza la raíz con el último elemento que ocupa la última posición en el Heap, posteriormente se verifica que el valor más grande entre sus hijos, si eso ocurre se realiza un intercambio en caso contrario finaliza el proceso.

## Análisis de complejidad

En el caso de Heap Sort tenemos un algoritmo de ordenamiento de datos eficiente, donde a pesar de no depender de la recursividad se logra una complejidad  $O(n \log n)$ .

## Actividad (Python) (EXTRA)

Descripción del código:

La función *OrdenarHeapSort* se le pasa toda la lista de números además de su tamaño, posteriormente para poder ordenar la lista se llama a la función *construirHeapMaxIni* la cual crea el árbol binario con los datos del arreglo, primero con el ciclo que el índice tomará los valores del a mitad del arreglo menos 1 a 0 dando la disminución de 1, y con estos valores se llamará a la función *MaxHeapify*.

NOTA 1: Las notas del programa se encuentran comentadas en el código.

A continuación, se muestra la salida del programa, es importante destacar que realmente el primer elemento es tomado como un "pivote" de manera en que se pueda llevar a cabo el código:

```
In [9]: runfile('C:/Users/pon_c/OneDrive/Documentos/
Universidad/3° Semestre/EDA II/Prácticas_LAB/2_practice/
Códigos_Manual/HeapSort.py', wdir='C:/Users/pon_c/
OneDrive/Documentos/Universidad/3° Semestre/EDA II/
Prácticas_LAB/2_practice/Códigos_Manual')
[0, 34, 78, 45, 12, 98, 67]
[0, 34, 98, 45, 12, 78, 67]
[0, 98, 34, 45, 12, 78, 67]
[0, 98, 78, 45, 12, 34, 67]
[0, 78, 67, 45, 12, 34, 98]
[0, 67, 34, 45, 12, 78, 98]
[0, 45, 34, 12, 67, 78, 98]
[0, 34, 12, 45, 67, 78, 98]
[0, 12, 34, 45, 67, 78, 98]
```

*Imagen 4: Salida de la lista ordenada a través del uso de Heap Sort*

Por otro lado, y como actividad, para poder llevar a cabo la versión del Heap Sort Máximo (Inversión de la lista) solo debemos cambiar la lógica de ordenamiento, dicho de otra forma, invertir los signos de comparación.

```
if( L<=sizeHeap and A[L]<A[i]):
    posMax=L
else:
    posMax=i

if(R<=sizeHeap and A[R]<A[posMax]):
    posMax=R
```

A continuación, se muestra la salida del programa, es importante destacar que realmente el primer elemento es tomado como un “pivot” de manera en que se pueda llevar a cabo el código:

```
In [1]: runfile('C:/Users/pon_c/OneDrive/Documentos/
Universidad/3° Semestre/EDA II/Prácticas_LAB/2_practice/
Códigos_Manual/HeapSort_invertida.py', wdir='C:/Users/
pon_c/OneDrive/Documentos/Universidad/3° Semestre/EDA II/
Prácticas_LAB/2_practice/Códigos_Manual')
[1, 34, 78, 45, 12, 98, 67]
[1, 34, 12, 45, 78, 98, 67]
[1, 12, 34, 45, 78, 98, 67]
[1, 34, 67, 45, 78, 98, 12]
[1, 45, 67, 98, 78, 34, 12]
[1, 67, 78, 98, 45, 34, 12]
[1, 78, 98, 67, 45, 34, 12]
[1, 98, 78, 67, 45, 34, 12]
```

*Imagen 5: Líneas modificadas en el algoritmo “Heap Sort” para invertir el orden de la lista.*

## Actividad 2. Ejercicios de laboratorio

### 2.1 Selection Sort e Insertion Sort

Para este problema se proporcionaron previamente los códigos de selection sort e insertion sort.

#### 2.1.1) Descripción acerca de Insertion Sort

Insertion sort es un algoritmo de ordenamiento de datos donde a través de “la inserción” y la comparación dentro de las iteraciones, es como este logra ordenar una lista o arreglo.

Al igual que en los algoritmos de ordenamiento previamente vistos, se necesita de un arreglo y lista, además de la longitud (Tamaño) de este ya que en base a esto es como realmente se limita las evaluaciones que se realizarán (En el ciclo for).

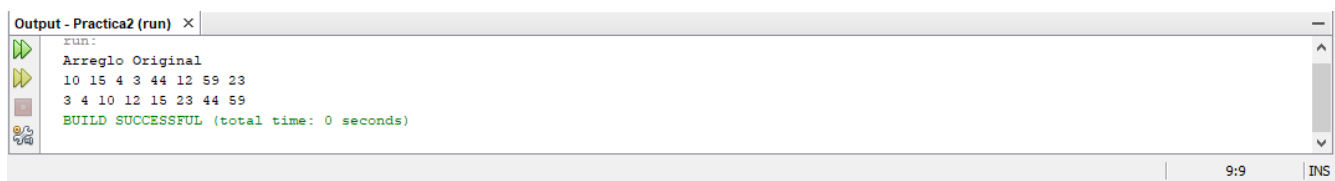
Por otra parte, donde realmente se lleva toda lógica del algoritmo es en este ciclo while:

```
while ( ( i > -1) && ( array [i] > key ) ) {  
    array [i+1] = array [i];  
    i--;  
}  
array[i+1] = key;
```

Imagen 6: Parte principal del algoritmo Insertion Sort.

Lo que sucede en estas líneas es la inserción de un elemento en el caso de que sea mayor al que le sigue esto con el fin de que conforme se vaya comparando cada uno de los elementos del arreglo o lista del lado izquierdo o del lado donde ya pasó el índice (en el caso del código era j ) ya se encuentren ordenados de menor a mayor los elementos.

A continuación, se muestra la salida de una lista (arreglo) ordenado a través del uso de Insertion Sort.



```
Output - Practica2 (run) x  
run:  
Arreglo Original  
10 15 4 3 44 12 59 23  
3 4 10 12 15 23 44 59  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Imagen 7: Salida de una lista ordenada a través de Insertion Sort

#### 2.1.2) Diferencia entre la invocación de insertion Sort y selection Sort

Algo importante a destacar es que la forma en que se invoca “insertion sort” respecto a la forma en que se invoca “selection sort”, se debe sobre todo a un concepto relacionado con programación orientada a objetos en específico con la privacidad que tiene los métodos ya sean públicos, privados o estáticos.

En este caso, insertion sort al ser un método estático no requiere o no necesita invocarse sobre un objeto simplemente puede utilizarse de manera directa en la función main.

```
public class InsertionSort{
    public static void insertionSort(int array[]) {
```

Por otro lado, en el caso de selection sort al ser una función(Método) no estática forzosamente se debe previamente declarar o crear un objeto que contenga o con el que se pueda utilizar los métodos de la clase "Selection Sort".

```
public void selectionSort(int[] arr){
    for (int i = 0; i < arr.length - 1; i++){
```

A continuación, se muestran la línea de código donde se crea un objeto del tipo Selection Sort para posteriormente ser invocada para el ordenamiento de una lista (arreglo).

```
SelectionSort seleccion = new SelectionSort();
seleccion.selectionSort(arr1);

System.out.println("Arreglo ordenado");
imprimirArreglo(arr2);
```

### 2.1.3) Descripción acerca de Selection Sort

Selection sort es una de las formas más sencillas de ordenar datos, consiste en seleccionar el valor más pequeño del arreglo e intercambiarlo con el primero (El más pequeño se encontrará al inicio) y esto repetidamente con los elementos restantes.

Al igual que todos los algoritmos de ordenamiento previos se necesita de un ciclo for para poder realizar las comparaciones necesarias al momento de ir ordenando la lista. A continuación, se muestra el ciclo for principal y como en este es necesario pasarle el tamaño del arreglo o lista para poder limitar las iteraciones del ciclo for. Además, es necesario destacar que este tipo de algoritmo de ordenamiento de datos se caracteriza o se basa en la posición de las comparaciones por lo tanto se hace uso de un variable que es utilizada como mínimo.

```
for (int i = 0; i < arr.length - 1; i++){
    System.out.println("Iteración " + (i+1));
    int min = i;
```

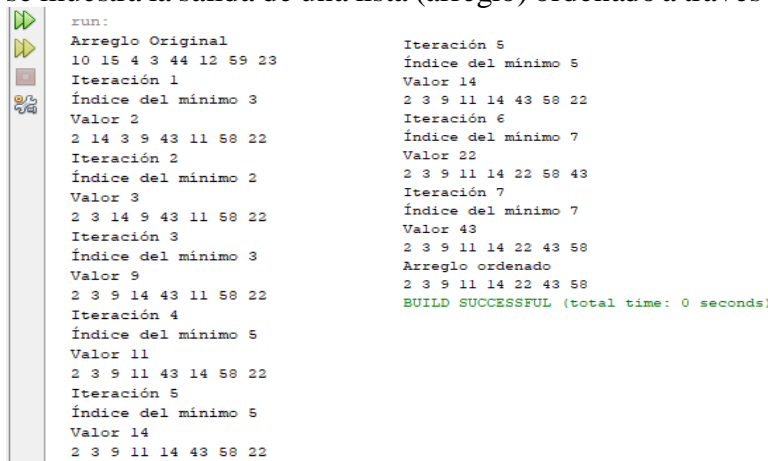
Por otro lado, en la parte donde realmente se lleva la lógica del algoritmo es en las siguientes líneas (Ver imagen posterior):

```
for (int j = i + 1; j < arr.length; j++){
    if (arr[j] < arr[min]){
        min = j;
    }
}
int smallerNumber = arr[min];
arr[min] = arr[i];
arr[i] = smallerNumber;
```



En el ciclo for lo que se hace es ir comparando los valores de la lista (En específico donde se encuentra el valor de j) con respecto a “min” o mínimo que es como dice su nombre el valor mínimo tenido en el momento (su índice), esto con el propósito de poder llevar a cabo la selección de posición de los elementos. Además, también se considera (Posterior al ciclo for) el cambio a través del resultado previo arrajado por el ciclo.

A continuación, se muestra la salida de una lista (arreglo) ordenado a través del uso de Insertion Sort.



```

run:
Arreglo Original
10 15 4 3 44 12 59 23
Iteración 1
Índice del mínimo 3
Valor 2
2 14 3 9 43 11 58 22
Iteración 2
Índice del mínimo 2
Valor 3
2 3 14 9 43 11 58 22
Iteración 3
Índice del mínimo 3
Valor 9
2 3 9 14 43 11 58 22
Iteración 4
Índice del mínimo 5
Valor 11
2 3 9 11 43 14 58 22
Iteración 5
Índice del mínimo 5
Valor 14
2 3 9 11 14 43 58 22
Iteración 6
Índice del mínimo 7
Valor 22
2 3 9 11 14 22 58 43
Iteración 7
Índice del mínimo 7
Valor 43
2 3 9 11 14 22 43 58
Arreglo ordenado
2 3 9 11 14 22 43 58
BUILD SUCCESSFUL (total time: 0 seconds)

```

Imagen 7: Salida de una lista ordenada a través de Insertion Sort

#### 2.1.4) Revisión y descripción acerca de detalles o peculiaridades de los códigos

Para concluir con este apartado de análisis, quiero destacar que realmente la forma en que se puede llevar a cabo estos algoritmos en Java realmente es muy agradable sobre todo porque el llamar clases (En específico sus métodos) desde otras clases es realmente fácil, cabe destacar que pese a ello se debe considerar algunas peculiaridades del paradigma al que está orientado ya que el hecho de definir el tipo de método es más que relevante como fue el caso de estático o no estático.

Por otro lado, en el código principal solamente se incluyen 2 métodos que son el **principal** donde se llevará solamente el llamado de los métodos que se necesitarán para poder ordenar las listas y el método imprimirArreglo que al ser un método que ambos algoritmos tienen en común es realmente conveniente dejarlo en la clase principal sobre todo para poder ahorrar algunas líneas de código.

## 2.2 Heap Sort

### 2.2.1) Heap Sort

Para este apartado se dio previamente el código en C de Heap el cual a continuación será descrito función por función:

La función “**printArray**” es la encargada solamente de hacer las impresiones del arreglo es por ello que para que funcione necesita tanto el arreglo como una variable que incluya el tamaño de este, para llevar a cabo la impresión simplemente usa un ciclo for encargado de la iteración de la impresión del arreglo.

La función “**main**” simplemente es la encargada de llevar a cabo tanto la declaración del arreglo como el llamado de la función **Heapsort**.

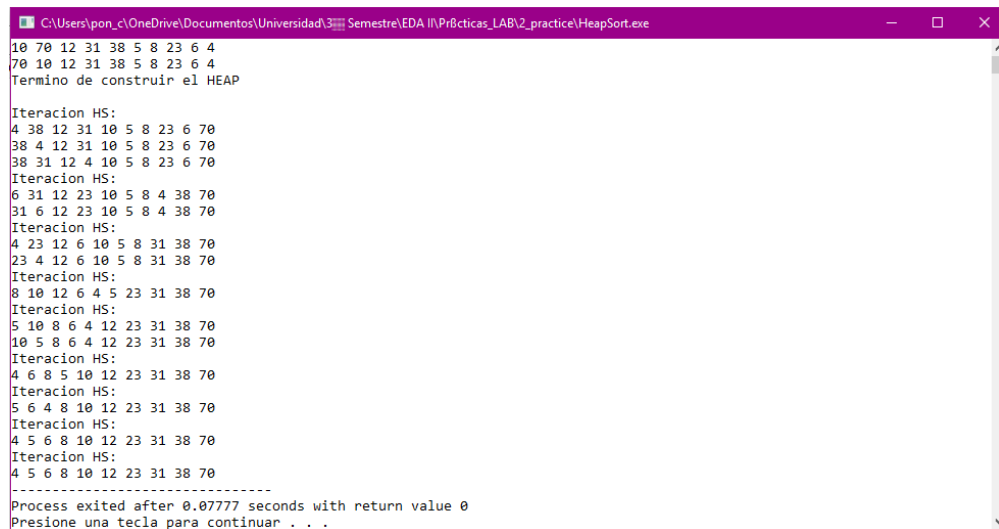
La función “**Heapsort**” es la encargada de llevar a cabo tanto la construcción del HEAP a través de otra función llamada BuildHEAP y de la parte de muestra y ordenamiento del arreglo una vez creado el HEAP a través de heapify. Para que esta función funcione simplemente se le pasa un apuntador al arreglo previamente establecido.

La función “**BuildHeap**” es la encargada de crear el HEAP a través de un ciclo for donde a dentro de este llama a la función heapify.

NOTA: Cabe destacar que además utiliza una variable auxiliar que sirve para el tamaño del heap donde esta emplea el tamaño del arreglo -1, esto con el fin de ir construyendo cada uno de las ramas del HEAP (árbol).

Por último, la función “**Heapify**” es donde realmente se lleva a cabo la lógica tanto del ordenamiento como de la construcción del HEAP para ello al igual que en el código en Python se necesitan de varias variables donde cada una sirve tanto para las condiciones como para determinar la cantidad de elemento que hay en el ordenamiento (Posterior al HEAP).

A continuación, se muestra la salida del programa utilizando heap sort:



```
C:\Users\pon_c\OneDrive\Documentos\Universidad\3er Semestre\EDA II\Prácticas_LAB\2_practice\HeapSort.exe
10 70 12 31 38 5 8 23 6 4
70 10 12 31 38 5 8 23 6 4
Termino de construir el HEAP

Iteracion HS:
4 38 12 31 10 5 8 23 6 70
38 4 12 31 10 5 8 23 6 70
38 31 12 4 10 5 8 23 6 70
Iteracion HS:
6 31 12 23 10 5 8 4 38 70
31 6 12 23 10 5 8 4 38 70
Iteracion HS:
4 23 12 6 10 5 8 31 38 70
23 4 12 6 10 5 8 31 38 70
Iteracion HS:
8 10 12 6 4 5 23 31 38 70
Iteracion HS:
5 10 8 6 4 12 23 31 38 70
10 5 8 6 4 12 23 31 38 70
Iteracion HS:
4 6 8 5 10 12 23 31 38 70
Iteracion HS:
5 6 4 8 10 12 23 31 38 70
Iteracion HS:
4 5 6 8 10 12 23 31 38 70
Iteracion HS:
4 5 6 8 10 12 23 31 38 70
-----
Process exited after 0.07777 seconds with return value 0
Presione una tecla para continuar . . .
```

Imagen 8: Salida de una lista ordenada a través de Insertion Sort

### 2.2.2) Heap Sort con ordenamiento descendente

Para llevar a cabo de forma inversa el ordenamiento lo primero que se tuvo que hacer fue analizar detenidamente que hacía cada función (Análisis previo), a lo que inmediatamente identifiqué que la función encargada de ese apartado era realmente “**Heapify**” por lo tanto lo único que se tuvo que hacer al igual que en su versión en Python fue simplemente cambiar 2 signos de comparación en las condiciones de la función.

Dicho lo anterior, a continuación, se muestra la salida del programa:

```
C:\Users\pon_c\OneDrive\Documentos\Universidad\3er Semestre\EDA II\Prácticas_LAB\2_practice\HeapSort_Inversa.exe
Termino de construir el HEAP
Iteracion HS:
70 6 5 10 38 12 8 23 31 4
5 6 70 10 38 12 8 23 31 4
Iteracion HS:
31 6 8 10 38 12 70 23 5 4
6 31 8 10 38 12 70 23 5 4
6 10 8 31 38 12 70 23 5 4
Iteracion HS:
31 10 8 23 38 12 70 6 5 4
8 10 31 23 38 12 70 6 5 4
Iteracion HS:
70 10 12 23 38 31 8 6 5 4
10 70 12 23 38 31 8 6 5 4
Iteracion HS:
31 23 12 70 38 10 8 6 5 4
Iteracion HS:
38 23 31 70 12 10 8 6 5 4
Iteracion HS:
70 38 31 23 12 10 8 6 5 4
Iteracion HS:
70 38 31 23 12 10 8 6 5 4
Iteracion HS:
70 38 31 23 12 10 8 6 5 4
-----
Process exited after 0.06793 seconds with return value 0
Presione una tecla para continuar . . .
```

Imagen 9: Salida de una lista ordenada de forma descendente a través de Insertion Sort

## Conclusiones

Durante esta práctica se conocieron 3 algoritmos de ordenamiento de datos por intercambio, fue el caso de Selectionsort, Insertionsort y Heapsort. Además, en esta práctica realmente resultó cómodo el poder ver algunas ventajas que nos ofrece Java y en específico los lenguajes orientados a objetos ya que el poder llamar los métodos (funciones) de otras clases sin necesidad más que de invocarlas (Salvo algunas excepciones donde previamente debe crearse un objeto para el uso) realmente facilita el paso e interacción entre el mismo programa.

Por otro lado, es necesario mencionar que algo que realmente resultó algo distinto, fue que en el caso de Heap sort (En Python o sea el del manual) fue necesario un dato extra para poder llevar a cabo el ordenamiento de la lista que se le pasó.

Por último, realmente me pareció un poco larga la práctica sobre todo por la parte del análisis que se tuvo que realizar a todos los códigos, sin embargo, en el último ejercicio del laboratorio que fue invertir o dejar de manera descendente al arreglo me resultó muy efectivo haber hecho un análisis previo de que hacía cada función sobre todo porque no tuve que recurrir al típico prueba y error sino ir directamente a la función que se encargaba del ordenamiento.

## REFERENCIAS

Elba Karen Saenz García. (2017). *Manual de Prácticas del laboratorio de Estructuras de Datos y algoritmos II*. UNAM, Facultad de Ingeniería.

Bradley N. Miller y David L. Ranum, Franklin, Beedle & Associates. (2011). *Problem Solving with Algorithms and Data Structures using Python*. Segunda Edición.

Mark J. Guzdial, Barbara Ericson. (2015). *Introducción a la computación y programación con Python*. 3ra Edición. Madrid España.