	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
No de Práctica(s):	PRACTICA #1 – ALGORITMOS DE ORDENAMIENTO PARTE 1
Integrante(s):	CARRILLO CERVANTES IVETTE ALEJANDRA
No. de Equipo de cómputo empleado:	TRABAJO EN CASA
No. de Lista o Brigada:	NO APLICA
Semestre:	2022 - 1
Fecha de entrega:	16 SEPTIEMBRE 2021
Observaciones:	

CALIFICACIÓN: _____

PRACTICA #1 ALGORITMOS DE ORDENAMIENTO PARTE 1

OBJETIVO GENERAL: El estudiante identificará la estructura de los algoritmos de ordenamiento InsertionSort, SelectionSort y HeapSort

OBJETIVO DE CLASE: El alumno revisará el uso de bibliotecas como capas de abstracción, así como realización de pruebas de los algoritmos para diferentes instancias de colecciones para conocer la complejidad de estos.

Actividades para el desarrollo de la práctica

Ejercicio 1. Análisis Inicial

Archivo ordenamientos.c

En este archivo se plantean 3 funciones diferentes las cuales corresponden a los algoritmos de ordenamiento vistos en clase:

- *Selection-Sort*

Es un algoritmo de ordenamiento por selección, consiste en revisar todos los elementos de una colección y seleccionar el valor más pequeño de la colección para intercambiarlo con el primero; este proceso que se repite hasta tener toda la colección ordenada. La complejidad de este algoritmo en el mejor caso, caso promedio y el peor caso es de $O(n^2)$.

La función selectionSort tiene como parámetro el arreglo y el número de elementos que tiene este, su complejidad se debe a que cuenta con dos ciclos “for” anidados, los cuales van recorriendo el arreglo haciendo comparaciones mediante un “if” para ver que elemento es el menor de la colección. Finalizando este proceso, se un intercambio del elemento menor por el elemento en el que se encuentra el primer for, esto se logra llamando a la función swap.

- *Insertion-Sort*

Es un algoritmo de ordenamiento por inserción, consiste en ir revisando elemento por elemento de una colección e irlos comparando con aquellos que se encuentran a su izquierda (los que ya están ordenados). La complejidad de este algoritmo en el mejor de los casos es de $O(n)$, mientras que en el caso promedio y el peor de los casos es de $O(n^2)$.

La función insertionSort tiene como parámetro el arreglo y el número de elementos que tiene este, su complejidad en el caso promedio y el peor caso se debe a que esta función tiene un ciclo while dentro de un ciclo for; esta función recorre la colección mediante el ciclo for y mientras que j sea mayor a cero y la variable auxiliar sea mejor al arreglo en la posición j-1, se cambiará el valor del elemento por el elemento a su izquierda. Este proceso se repetirá en toda la lista hasta que se obtenga la lista ordenada.

- *Heap-Sort*

Es un algoritmo de ordenamiento por selección, consiste en crear una estructura no lineal (árbol), y en cada iteración ir eliminando la raíz, conservando la integridad de Heap. Este algoritmo es el más eficiente, ya que tiene para el mejor caso, caso promedio y el peor de los casos una complejidad de $O(n \log(n))$.

La función HeapSort tiene como parámetros el arreglo y el número de elementos que tiene este; dentro de esta función, primero se manda a llamar a otra llamada “BuildHeap” la cual tiene como función crear un Heap; después, tiene solamente un ciclo “*for*” el cual se irá encargando de cambiar el head del Heap por el ultimo de este, así eliminando la raíz; finalmente, igual dentro de este “*for*” se llama a la función “*Heapify*” la cual reacomoda el heap. Esto se va repitiendo hasta que la colección se ordene.

Cabe destacar que este programa utiliza sus propias bibliotecas, las cuales las manda a llamar desde un principio del programa:

```
1   #include "ordenamientos.h"
2   #include "utilidades.h"
3   #include <stdio.h>
```

bibliotecas

Archivo odenamientos.h

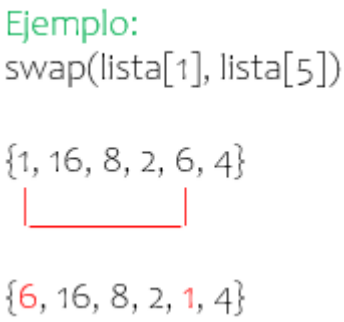
En este archivo se definen las funciones que se encuentran en el archivo “ordenamientos.c”

Archivo utilidades.c

Dentro de este archivo se encuentran algunas funciones que se van a mandar a llamar en los algoritmos de ordenamiento, dichas funciones son las siguientes:

- *Swap*

Tiene como función intercambiar en el arreglo dos elementos, los cuales se reciben como parámetros.



- *printArray*

Tiene como función imprimir el arreglo completo. Recibe como parámetros el arreglo y el tamaño de este.

- *printSubArray*

Tiene como función imprimir los sub-arreglos que se forman al dividir un arreglo en 2 partes.

Archivo utilidades.h

En este archivo se definen las funciones que se encuentran en el archivo “utilidades.c”

Ejercicio 2. Probando los ordenamientos

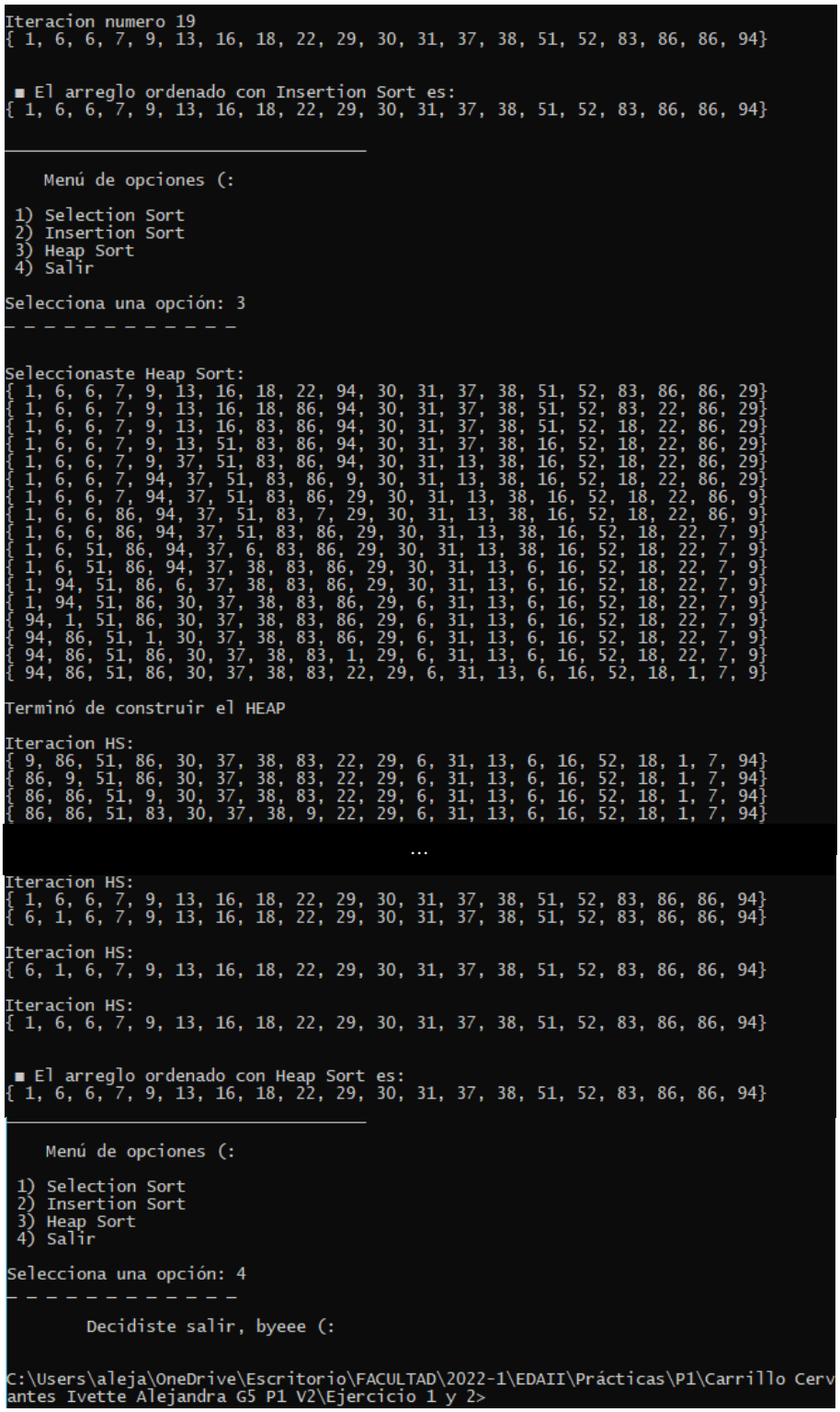
Este programa tiene como función principal probar todos los algoritmos de ordenamiento, para ello se creó un nuevo archivo llamado "*ejercicio2.c*" en la carpeta "*Ejercicio 1 y 2*", en el cual se mandó a llamar a la biblioteca "*ordenamientos.c*" (en esta biblioteca se encuentran las funciones de cada algoritmo de ordenamiento). Para probar cada algoritmo se inicializó un arreglo con 20 elementos aleatorios con ayuda de la función srand() (esta se encuentra en la biblioteca "*stdlib.h*") y un ciclo "*for*" el cual fue guardando en el arreglo cada elemento aleatorio que se generó en un rango del 0 al 100. Después, se realizó un menú que se va repitiendo hasta que el usuario decida salir, este menú tiene las siguientes opciones para utilizar con el arreglo:

- Selection Sort
- Insertion Sort
- Heap Sort
- Salir

Finalmente, la salida del programa se ve de la siguiente manera:

```
                    Algoritmos de ordenamiento (:

Se generó un arreglo aleatorio con 20 elementos:
{ 6, 18, 86, 6, 38, 16, 86, 52, 51, 94, 83, 1, 9, 7, 13, 22, 30, 29, 37, 31}

    Menú de opciones (:

 1) Selection Sort
 2) Insertion Sort
 3) Heap Sort
 4) Salir

Selecciona una opción: 1

Seleccionaste Selection Sort:

Iteracion numero 1
{ 1, 18, 86, 6, 38, 16, 86, 52, 51, 94, 83, 6, 9, 7, 13, 22, 30, 29, 37, 31}

Iteracion numero 2
{ 1, 6, 86, 18, 38, 16, 86, 52, 51, 94, 83, 6, 9, 7, 13, 22, 30, 29, 37, 31}

Iteracion numero 3
                                    ...
Iteracion numero 19
{ 1, 6, 6, 7, 9, 13, 16, 18, 22, 29, 30, 31, 37, 38, 51, 52, 83, 86, 86, 94}

 ■ El arreglo ordenado con Selection Sort es:
{ 1, 6, 6, 7, 9, 13, 16, 18, 22, 29, 30, 31, 37, 38, 51, 52, 83, 86, 86, 94}

    Menú de opciones (:

 1) Selection Sort
 2) Insertion Sort
 3) Heap Sort
 4) Salir

Selecciona una opción: 2

Seleccionaste Insertion Sort:

Iteracion numero 1
{ 1, 6, 6, 7, 9, 13, 16, 18, 22, 29, 30, 31, 37, 38, 51, 52, 83, 86, 86, 94}

Iteracion numero 2
{ 1, 6, 6, 7, 9, 13, 16, 18, 22, 29, 30, 31, 37, 38, 51, 52, 83, 86, 86, 94}

Iteracion numero 3
{ 1, 6, 6, 7, 9, 13, 16, 18, 22, 29, 30, 31, 37, 38, 51, 52, 83, 86, 86, 94}
                                    ...
```

NOTA: Para que no fuera una captura muy larga, se hizo el recordé de algunas iteraciones.



./Ejercicio 1 y 2/ejercicio2.c

Ejercicio 3. Análisis de Complejidad computacional

En este ejercicio se generó un nuevo archivo llamado "*ejercicio3.c*" en la carpeta "*Ejercicio3*", se realizó el mismo procedimiento que en el ejercicio anterior; sin embargo, se hicieron algunas modificaciones, esta vez se realizó un menú con las mismas opciones que el ejercicio anterior, pero ahora se le preguntaba al usuario cuantos elementos quería que tuviera el arreglo y con base a la respuesta se generaban los números aleatorios ahora en un rango de 0 a 1000. Esta pregunta se le hacia al usuario cada vez que se elegia un algoritmo de ordenamiento.

Luego, con respecto al archivo "*ordenamientos.c*", se hicieron las modificaciones necesarias para que se fuera contando cada operación de comparación, intercambio e inserción, con el fin de realizar el análisis de complejidad temporal de cada algoritmo; esto se realizó con ayuda de una variable llamada contador inicializada en cero para cada función la cual iba aumentando 1 cada que se hacia una operación de las mencionadas anteriormente; sin embargo, para el caso del algoritmo HeapSort, la cual llama a su vez a las funciones "*Heapify*" y "*BuildHeap*", en estas ultimas se hizo lo mismo pero se retorno el número de operaciones (por lo tanto se volvieron en funciones int, en vez de void), por lo tanto en la función HeapSort cada vez que se llama a una de estas dos funciones, se guarda el valor de retorno y se le sumaba este valor al contador principal de HeapSort. Al final de cada función se imprime el número de operaciones realizadas.

Finalmente, se hicieron pruebas con cada algoritmo de ordenamiento con arreglos de diferente tamaño: 50, 100, 200, 500, 1000, 2000. Se hicieron 5 ejecuciones por cada tamaño y se realizó una tabla general con el promedio del número de operaciones de cada uno. (Los datos de las 5 ejecuciones de cada tamaño se encuentran en el archivo "*Tablas.xlsx*" en la carpeta "*Ejercicio3*").

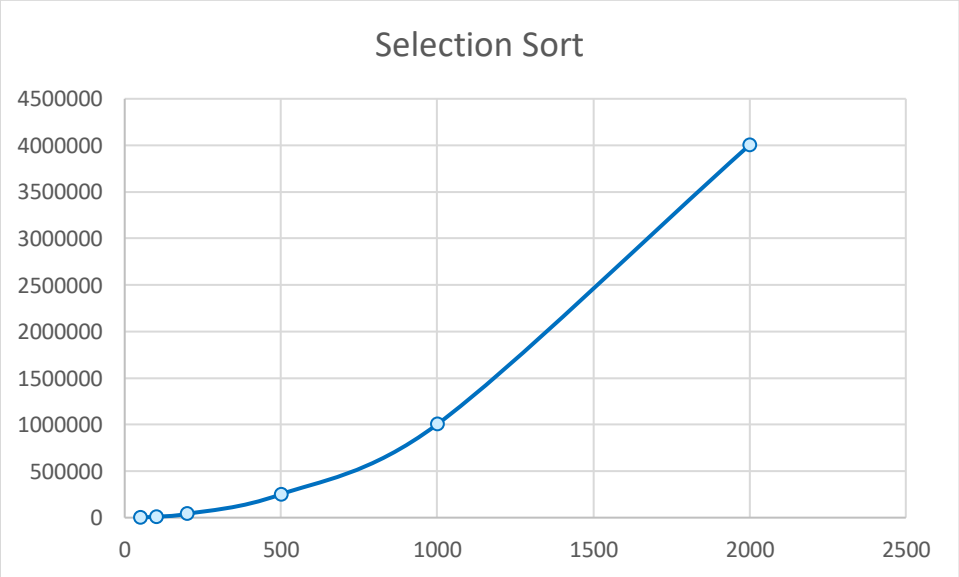
A continuación, se muestran las tablas en general, al igual que las gráficas de cada algoritmo.

Selection Sort

Complejidad de este algoritmo en el mejor caso, caso promedio y el peor caso es de $O(n^2)$.

SELECTION SORT	
Valor	Operaciones
50	2594.2
100	10192.8
200	40392.4
500	250989.2
1000	1001990.6
2000	4003997

Tabla Selection-Sort



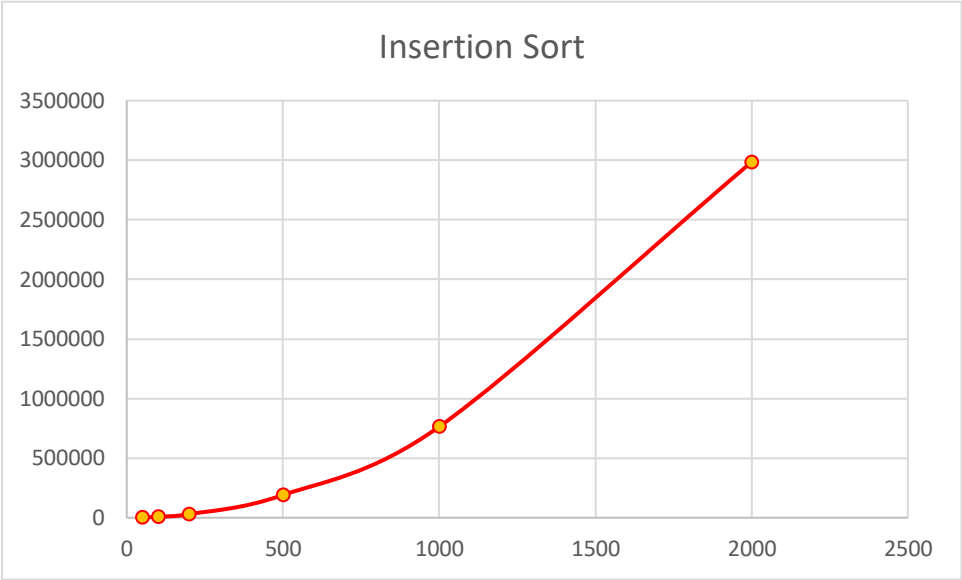
Gráfica Selection-Sort

Insertion Sort

Complejidad de este algoritmo en el mejor de los casos es de $O(n)$, mientras que en el caso promedio y el peor de los casos es de $O(n^2)$.

INSERTION SORT	
Valor	Operaciones
50	1977
100	8284.2
200	29615.4
500	190508.4
1000	763548.6
2000	2984930.4

Tabla Insertion-Sort



Gráfica Insertion-Sort

Heap Sort

Complejidad de este algoritmo para el mejor caso, caso promedio y el peor de los casos una complejidad de $O(n\log(n))$.

HEAP SORT	
Valor	Operaciones
50	1678.8
100	3944.2
200	9171.8
500	27303.6
1000	60956.4
2000	134599.8

Tabla Heap-Sort

Gráfica Heap-Sort

Se observó que los algoritmos se tardaban mucho tiempo al ejecutarse, ya que se imprimía cada iteración que se realizaba; por lo tanto, se quitaron dichas impresiones para que fuera más rápido obtener los datos. Sin embargo, se hicieron algunas ejecuciones midiendo el tiempo tomando en cuenta las impresiones de cada iteración (estos datos se encuentran en el archivo "*Tablas.xlsx*" en la carpeta "*Ejercicio3*").

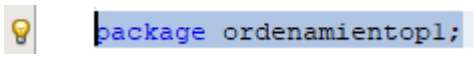
La salida del programa en cada ejecución es la siguiente:

```
Numero de operaciones: 134638
Tiempo de ejecución: 0.1000 segundos
```

./Ejercicio 3/ejercicio3.c

Ejercicio 4. Ahora en Java

Para este ultimo ejercicio, se utilizó ahora el lenguaje de programación Java y el IDE NetBeans, igual que los ejercicios anteriores, este tiene como función principal probar los algoritmos de ordenamiento; sin embargo, en vez de bibliotecas, este programa contiene clases, las cuales no hubo necesidad de llamar una por una, si no que se especificó que se encontraban en el directorio "*ordenamientop1*" en el cual se tienen los archivos de los algoritmos de ordenamiento "*selection sort*" e "*insertion sort*".



Se manda a llamar el directorio

Para realizar las pruebas necesarias se inicializaron dos arreglos, en el primer arreglo se mando a llamar al método "insertionSort" de la clase "Insertion" con el fin de que se ordenara el arreglo con este método; mientras que, en el segundo arreglo se mando a llamar al método "*selectionSort*" de la clase "*Inserción*". Finalmente se manda a llamar al método "*imprimriArreglo*" de la clase "*Utilerias*" dos veces para imprimir el arreglo ordenado de cada arreglo.

```
run:
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
BUILD SUCCESSFUL (total time: 0 seconds)
```

./OrdenamientoP1/OrdenamientoP1.class

Conclusiones

Se cumplieron los objetivos de esta práctica, ya que se probaron los algoritmos de ordenamiento InsertionSort, SelectionSort y HeapSort vistos en clase, identificamos el funcionamiento de cada uno, así como su estructura en lenguaje C y dos de estos algoritmos en lenguaje Java. También, retomamos el concepto de capas de abstracción con el fin de realizar pruebas para diferentes instancias de colecciones. Y mediante el ejercicio 3, observamos la complejidad que tienen estos algoritmos con respecto al número de operaciones que hizo cada uno de estos. Se llegó a la conclusión de que el algoritmo más eficiente es HeapSort, ya que cuenta con una complejidad de $O(nlog(n))$, este algoritmo fue el más tardado debido a las impresiones de cada iteración, pero una vez quitando estas impresiones, siguió siendo el más eficiente.

Se cumplieron con todas las actividades de esta práctica, y a pesar de que fue un tanto tediosa por tantas ejecuciones de cada algoritmo, considero que cada ejercicio estuvo muy bien planteado para comprender este tema. Además, fue la primera práctica en la materia de Estructura de Datos y Algoritmos II en la cual se trabajó con el lenguaje de programación Java y se vio el uso de arreglos. (: