



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Tista García Edgar Ing.

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 5

No de Práctica(s): 1

Integrante(s): Téllez González Jorge Luis

*No. de Equipo de
cómputo empleado:* 5

No. de Lista o Brigada: X

Semestre: 2020-2

Fecha de entrega: 09/02/2020

Observaciones:

CALIFICACIÓN: _____



Índice

1. Introducción	2
2. Objetivos	2
3. Análisis inicial de Insertion-Sort y Selection-Sort	2
3.1. Comentarios y análisis de los headers	3
3.2. Desarrollo del entorno de pruebas	5
3.3. Pruebas de ejecución	6
4. Ordenamiento en Java	8
4.1. Comentarios y análisis	8
5. Complejidad computacional	10
5.1. Resultados obtenidos	11
5.2. Análisis de resultados	13
6. Conclusiones	14

1. Introducción

El ordenamiento de datos es una actividad humana común, la cual implica la reagrupación o reorganización de un conjunto de datos de cualquier tamaño.

Entre las aplicaciones que esta actividad tiene en nuestra época contemporánea se encuentra el uso de algoritmos de ordenamiento en ambientes computacionales, el manejo de grandes volúmenes de información de forma eficiente y, principalmente, facilitar la búsqueda de esta información.

Se define al **ordenamiento** como el reacomodo de una colección de elementos que cumplen un criterio definido con el fin de establecer una secuencia lógica entre un conjunto de elementos. Este ordenamiento, además, puede seguir tanto un orden ascendente como descendente.



Figura 1: *Proceso de ordenamiento de un conjunto de datos.*

En el siguiente reporte escrito se analizarán dos algoritmos de ordenamiento: Insertion-Sort y Selection-Sort. Se trabajará sobre un conjunto de archivos proporcionados y se realizarán análisis y pruebas de eficacia en ambos. Así mismo, se representará gráficamente la complejidad computacional de ambos algoritmos y, con base en ello, se determinará cuál de ellos posee un mejor comportamiento (es decir, que requiera menor cantidad de operaciones con el fin de lograr su propósito).

2. Objetivos

- El estudiante identificará la estructura de los algoritmos de ordenamiento por selección y por inserción en C y Java.
- Realizar un programa funcional con el fin de probar la eficacia de ambos algoritmos.
- Verificar la complejidad computacional de *Insertion-Sort* y *Selection-Sort* por medio de la cantidad de operaciones realizadas.

3. Análisis inicial de Insertion-Sort y Selection-Sort

El concepto de las capas de abstracción permite a un programador elaborar sus propias bibliotecas basado en la solución de problemas específicos para poder elaborar soluciones más complejas.

Con el fin de iniciar el análisis se nos han proporcionado dos bibliotecas: **utilerias.h** y **ordenamientos.h**. La primera biblioteca posee en su interior herramientas útiles para la ejecución de un programa que utilice los algoritmos de *Insertion-Sort* y *Selection-Sort*; cuya implementación en C se encuentra presente en la segunda biblioteca.

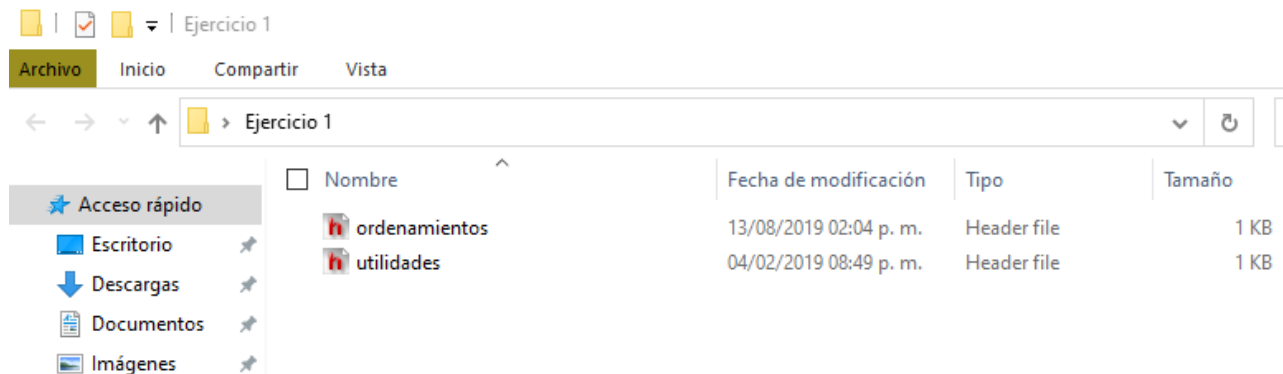


Figura 2: Archivos header proporcionados.

Para el desarrollo de las siguientes actividades en un ambiente idóneo se utilizará el IDE *NetBeans* con soporte C/C++ y el compilador *Cygwin*. A continuación, se procede a abrir ambos archivos en el IDE con el fin de observar su estructura y contenido.

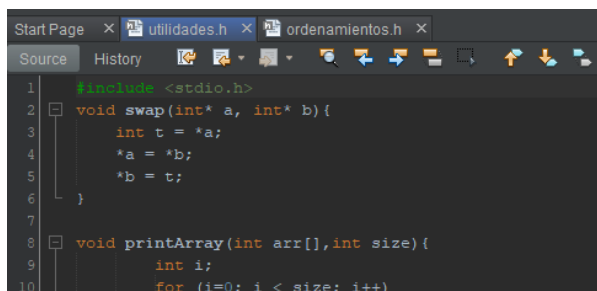


Figura 3: Vista general de los headers en NetBeans.

3.1. Comentarios y análisis de los headers

Como se comentó previamente, el archivo **utilerias.h** está conformado por tres funciones declaradas: `swap`, `printArray` y `printSubArray`:

- `void swap(int* a, int* b)`: Esta función tiene como propósito recibir por referencia dos variables de tipo de dato entero. Al inicio, la dirección de 'a' será almacenada en la variable temporal 't', posteriormente, al apuntador *a le será asignada la dirección de la variable 'b', mientras que a *b le será asignada la dirección original que guardaba el apuntador *a; intercambiando efectivamente las direcciones de memoria. La función no devuelve nada al término de su ejecución.

- `void printArray(int arr[], int size)`: Esta función que recibe un arreglo y su tamaño indicado por un entero imprime el arreglo especificado por medio de un ciclo *for*. Es indispensable para seguir el flujo de los algoritmos de ordenamiento y verificar que se encuentren realizando los pasos correctos. Tampoco devuelve nada tras ejecutarse.
- `void printSubArray(int arr[], int low, int high)`: A diferencia de la función anterior, esta recibe una cota inferior *low* y una cota superior *high* para imprimir únicamente una parte especificada del arreglo y no todo el arreglo con cada uno de sus elementos.

```
printf("Sub array : ");  
for (i=low; i <= high; i++)  
    printf("%d ", arr[i]);  
printf("\n");
```

Figura 4: Cota superior e inferior de impresión del arreglo.

A continuación, se abordarán las implementaciones en C de Insertion-Sort y Selection-Sort presentes en el archivo **ordenamientos.h**:

- La implementación del algoritmo Selection-Sort está compuesta por una función *selectionSort* que recibe un arreglo y un entero *n* que representa la longitud del arreglo recibido. Se declaran dos variables de control 'i' y 'j' para su uso en dos ciclos *for* anidados que tienen como objetivo iterar y recorrer el arreglo de forma sucesiva.

Por cada iteración en 'i' la función se encontrará realizando comparaciones repetidas que preguntarán si `arreglo[j] < arreglo[indiceMenor]`. Si la condición es cierta, *indiceMenor* cambiará su valor, caso contrario, no ocurrirá nada. Finalmente, tras haber hecho todas las comparaciones iteradas sobre el arreglo, intercambiará posiciones entre el arreglo[i] y el arreglo[indiceMenor].

```
for(j = i+1; j<n; j++){  
    if(arreglo[j]<arreglo[indiceMenor])  
        indiceMenor=j;  
}  
if (i!=indiceMenor){  
    swap(&arreglo[i],&arreglo[indiceMenor]); //Función ejecutada para intercambiar lugares en el arreglo.
```

Figura 5: Proceso de comparación e intercambio.

Es importante observar que esta operación tan solo ha ordenado un elemento, por lo tanto, harán falta una mayor cantidad de iteraciones para ordenar el arreglo dependiendo de su tamaño.

- La función *insertionSort* recibe un arreglo y su tamaño definido por un entero y, a diferencia de la función anterior, realiza comparaciones sucesivas de tal forma que, conforme avanza en el recorrido del arreglo con un ciclo *for*, comienza a comparar con un ciclo *while* anidado un dato denominado *indice* con los datos que se encuentran a su izquierda; siempre y cuando se cumpla que $j > 0$. Se tiene

como consideración inicial en cada caso que los elementos que se encuentran a su izquierda ya se encuentran ordenados.

En caso de que una de las comparaciones tengan valor falso, el arreglo permanece inalterado y el algoritmo continua iterando. Cuando encuentra un valor que hace que ambas proposiciones sean verdaderas, la función copia el primer elemento a la izquierda de la posición del índice actual. Este proceso se repite sucesivamente hasta que el índice finalmente queda en su lugar correspondiente, que en esta implementación corresponde al primer elemento en el arreglo. Es importante señalar que el valor de *índice* no se pierde ya que se encuentra almacenado durante todo el proceso de copiado.

```
24 | ☐ | for(i=1; i<n; i++){  
25 | ☐ |     j=i;  
26 | ☐ |     aux=a[i];  
27 | ☐ |     while (j>0 && aux < a[j-1]){
```

Figura 6: Condición de entrada al ciclo while en insertionSort.

3.2. Desarrollo del entorno de pruebas

A continuación se muestra el programa realizado con el fin de poner a prueba los algoritmos de ordenamiento propuestos:

- En primer lugar, se ha creado un menú que, mediante una variable entera de selección y un ciclo while, permite al usuario elegir qué algoritmo desea utilizar para ordenar un arreglo generado aleatoriamente.

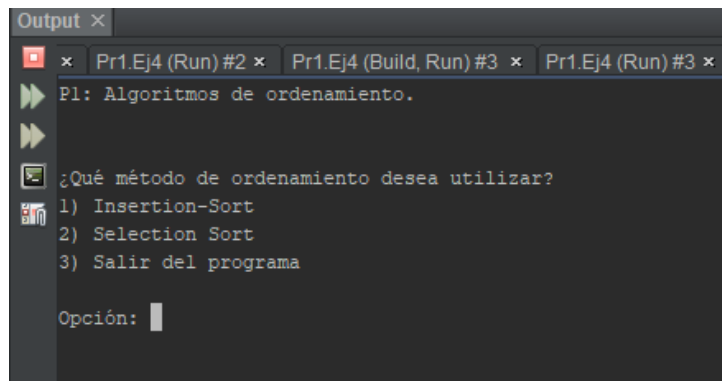
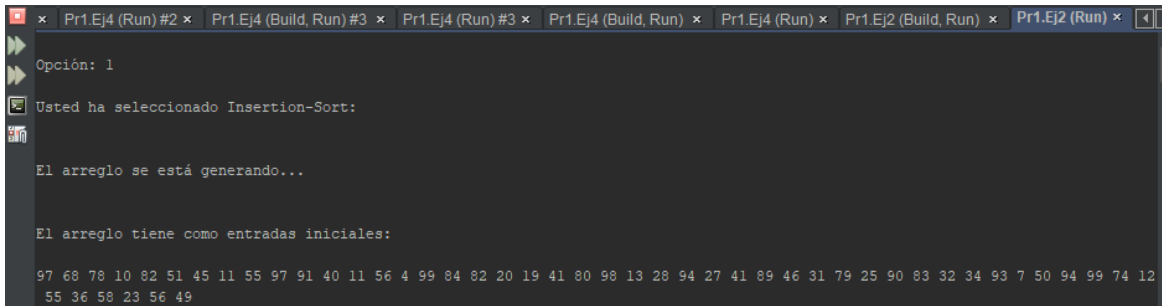


Figura 7: Menú inicial del programa.

Una vez que el usuario selecciona una opción, el programa continuará por medio de un switch con el código correspondiente a cada algoritmo. Así mismo, se puede detener la ejecución del programa si así se desea.

- Independientemente del método de ordenamiento seleccionado, el programa continuará utilizando una función denominada *llenarArreglo* que, teniendo en este caso como entrada un arreglo de 50 elementos, rellenará cada uno de sus espacios con números aleatorios haciendo uso de la función *srand* y *rand*. La primera cambiará la semilla de generación de números aleatorios con base en el reloj del sistema y la segunda será usada para generar un número aleatorio de dos dígitos que sea asignado a un espacio del arreglo. Finalmente se llama a la función *printArray* que imprime en pantalla el arreglo generado.



```
Opción: 1
Usted ha seleccionado Insertion-Sort:
El arreglo se está generando...
El arreglo tiene como entradas iniciales:
97 68 78 10 82 51 45 11 55 97 91 40 11 56 4 99 84 82 20 19 41 80 98 13 28 94 27 41 89 46 31 79 25 90 83 32 34 93 7 50 94 99 74 12
55 36 58 23 56 49
```

Figura 8: Generación del arreglo aleatorio.

```
case 1:
    printf("\nUsted ha seleccionado Insertion-Sort:\n\n");
    llenarArreglo(arreglo, N);
    printf("\nEl arreglo tiene como entradas iniciales: \n\n");
    printArray(arreglo, N);
    printf("\nAhora, ordenando el arreglo...\n\n");
    insertionSort(arreglo, N);
    printf("\n¡Felicitaciones! Su arreglo está ordenado. :) \n\n");
    printArray(arreglo, N);

break;
case 2:
    printf("\nUsted ha seleccionado Selection-Sort:\n\n");
    llenarArreglo(arreglo, N);
    printf("\nEl arreglo tiene como entradas iniciales: \n\n");
    printArray(arreglo, N);
    printf("\nAhora, ordenando el arreglo...\n\n");
    selectionSort(arreglo, N);
    printf("\n¡Felicitaciones! Su arreglo está ordenado. :) \n\n");
    printArray(arreglo, N);
```

Figura 9: Estructura general de ambos programas.

3.3. Pruebas de ejecución

La primera implementación que será puesta a prueba será la correspondiente a Insertion-Sort. Dado un arreglo de 50 elementos generados aleatoriamente, el programa mostrará las iteraciones realizadas junto con los cambios hechos en cada una. Finalmente, mostrará el arreglo completamente ordenado:

```

Output x
Pr1.Ej2 (Build, Run) #2 x Pr1.Ej2 (Build, Run) x Pr1.Ej2 (Run) x
El arreglo tiene como entradas iniciales:
30 28 23 76 18 18 46 74 24 10 33 34 27 65 24 17 13 51 21 79 36 96 27 27 55 10 29 64 11 36 92 74 48 5 3 60 78 99 24 28 0 80 67 28 82 96 29 74 98 2
Ahora, ordenando el arreglo...

Iteracion 1:
28 30 23 76 18 18 46 74 24 10 33 34 27 65 24 17 13 51 21 79 36 96 27 27 55 10 29 64 11 36 92 74 48 5 3 60 78 99 24 28 0 80 67 28 82 96 29 74 98 2
Iteracion 2:
23 28 30 76 18 18 46 74 24 10 33 34 27 65 24 17 13 51 21 79 36 96 27 27 55 10 29 64 11 36 92 74 48 5 3 60 78 99 24 28 0 80 67 28 82 96 29 74 98 2
Iteracion 3:
23 28 30 76 18 18 46 74 24 10 33 34 27 65 24 17 13 51 21 79 36 96 27 27 55 10 29 64 11 36 92 74 48 5 3 60 78 99 24 28 0 80 67 28 82 96 29 74 98 2
Iteracion 4:
18 23 28 30 76 18 18 46 74 24 10 33 34 27 65 24 17 13 51 21 79 36 96 27 27 55 10 29 64 11 36 92 74 48 5 3 60 78 99 24 28 0 80 67 28 82 96 29 74 98 2

```

Figura 10: *Primeras iteraciones de insertionSort.*

Una vez que se han realizado las iteraciones correspondientes, el programa muestra en pantalla una línea indicando que el arreglo ha sido ordenado y lo imprime:

```

Iteracion 49:
0 2 3 5 10 10 11 13 17 18 18 21 23 24 24 24 27 27 27 28 28 28 29 29 30 33 34 36 36 46 48 51 55 60 64 65 67 74 74 74 76 78 79 80 82 92 96 96 98 99
¡Felicidades! Su arreglo está ordenado. :)
0 2 3 5 10 10 11 13 17 18 18 21 23 24 24 24 27 27 27 28 28 28 29 29 30 33 34 36 36 46 48 51 55 60 64 65 67 74 74 74 76 78 79 80 82 92 96 96 98 99

```

Figura 11: *Resultado obtenido con el primer programa.*

El segundo programa funciona de forma similar, con la única diferencia de que ahora se utilizará el algoritmo de ordenamiento *Selection-Sort*:

```

El arreglo tiene como entradas iniciales:
26 83 16 40 12 92 62 36 33 46 33 18 60 96 67 6 27 59 44 86 77 18 76 91 62 9 98 49 22 31 9 13 2 96 28 72 33 8 45 65 82 74 57 36 34 92 15 71 1 20
Ahora, ordenando el arreglo...

Iteracion 1:
1 83 16 40 12 92 62 36 33 46 33 18 60 96 67 6 27 59 44 86 77 18 76 91 62 9 98 49 22 31 9 13 2 96 28 72 33 8 45 65 82 74 57 36 34 92 15 71 26 20
Iteracion 2:
1 2 16 40 12 92 62 36 33 46 33 18 60 96 67 6 27 59 44 86 77 18 76 91 62 9 98 49 22 31 9 13 83 96 28 72 33 8 45 65 82 74 57 36 34 92 15 71 26 20
Iteracion 3:
1 2 6 40 12 92 62 36 33 46 33 18 60 96 67 16 27 59 44 86 77 18 76 91 62 9 98 49 22 31 9 13 83 96 28 72 33 8 45 65 82 74 57 36 34 92 15 71 26 20

```

Figura 12: *Primeras iteraciones de selectionSort.*


```

Iteracion 49:
1 2 6 8 9 9 12 13 15 16 18 18 20 22 26 27 28 31 33 33 33 34 36 36 40 44 45 46 49 57 59 60 62 62 65 67 71 72 74 76 77 82 83 86 91 92 92 96 96 98
;Felicitades! Su arreglo está ordenado. :)
1 2 6 8 9 9 12 13 15 16 18 18 20 22 26 27 28 31 33 33 33 34 36 36 40 44 45 46 49 57 59 60 62 62 65 67 71 72 74 76 77 82 83 86 91 92 92 96 96 98

```

Figura 13: Arreglo ordenado obtenido.

4. Ordenamiento en Java

La tercera actividad consistió en el análisis del proyecto *Práctica 1* en NetBeans. Al abrir por primera vez el proyecto se puede observar la existencia de 4 clases presentes en un único paquete:

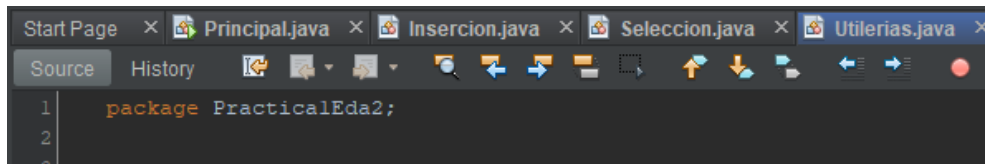


Figura 14: Paquete que contiene las 4 clases a analizar.

4.1. Comentarios y análisis

- La primera clase en analizar será *Principal.java*: Dentro de su método principal se declaran 2 arreglos de 8 elementos ya predefinidos. A continuación, se imprimen ambos arreglos llamando al método **imprimirArreglo** presente en la clase *Utilerias*; enviando directamente como parámetro ambos arreglos. Luego, se manda el primer arreglo directamente al método **insertionSort** de la clase *Insercion*.

Existe una característica notable en las siguientes líneas, pues introduce la creación de un objeto de tipo *Seleccion* que es utilizado para invocar el método **selectionSort** y enviarle el segundo arreglo.

Para finalizar la ejecución del programa, se imprime un mensaje en pantalla indicando que los arreglos han sido ordenados junto con su respectiva impresión con el uso de **imprimirArreglo**.

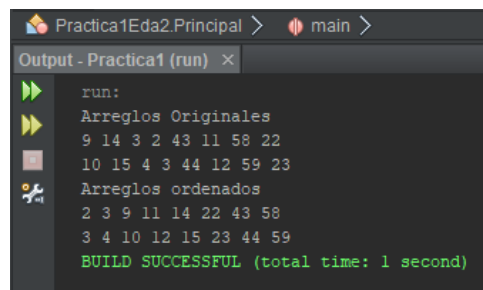


Figura 15: Ejecución del programa.

- A continuación se tiene a la clase *Insercion*: su estructura general es similar a la función *insertionSort* presente en C. Sin embargo, existen diferencias notables: No se recibe entre los parámetros el nombre del arreglo, en cambio, se extrae su tamaño accediendo a su propiedad 'length'; la cual parece que ya se encuentra definida para cada arreglo que se declare.

```
public static void insertio
    int n = array.length;
```

Figura 16: Atributo del arreglo que especifica su tamaño. El valor se almacena en una variable entera.

Otro detalle importante y que me causó cierta confusión fue la forma en que fue implementado el ciclo *for* y *while*, ya que en cierta forma parece como si estuviese 'al revés' con respecto a la implementación de C. Además, la condición $i > -1$ me pareció poco intuitiva y un tanto extraña de seguir a la hora de tratar de hacer una prueba de escritorio.

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

Figura 17: La condición de la discordia.

- La clase *Seleccion* posee a primera vista una estructura casi idéntica a la función *selectionSort* de C aunque con un diferencia notable: tras la ejecución del ciclo *for* no se tiene la condicional *if* y se salta directamente al método **intercambiar** de la clase *Utilerias*.

```
//Aquí se carece de la condicional del if y
//¿Por qué aquí se envían tres parámetros?
Utilerias.intercambiar(arr, i,min);
```

Figura 18: Método de intercambio.

En un inicio me generó dudas el hecho de que se enviaran tres parámetros al método. Estas dudas quedaron un poco despejadas al proseguir con el análisis de la última clase.

- Para finalizar se tiene la clase *Utilerias* la cual posee dos métodos declarados y definidos: **imprimirArreglo** e **intercambiar**. En el primer método llamó mucho mi atención la forma en que está escrito el ciclo *for* para imprimir el arreglo; pues solo introduce una variable entera acompañada del texto ':arreglo'. Es una notación que nunca había visto anteriormente.

```
for(int i:arreglo){ //Esta f
    System.out.print(i+" ");
}
```

Figura 19: Notación del ciclo *for* en **imprimirArreglo**

Posteriormente, fue notable observar que el funcionamiento de **intercambiar** es bastante intuitivo y sencillo de seguir: únicamente guarda en una variable temporal el valor presente en la posición 'x' del arreglo que recibe, copia el valor de la posición 'y' en el lugar de 'x' y finalmente el valor original de la posición 'x' guardado en 'temp' es almacenado en la posición 'y' del arreglo.

```
int temp = arr[x];  
//Tan solo guarda e  
arr[x] = arr[y];  
arr[y] = temp; //F
```

Figura 20: Operación de intercambio de posiciones en el arreglo recibido.

El hecho de que Java maneje los apuntadores de tal forma que su uso sea implícito al enviar variables a los métodos sin duda hace más sencilla la lectura del código fuente.

5. Complejidad computacional

Un tema común en el análisis de algoritmos se encuentra en el total de operaciones que realiza con el fin de lograr su objetivo junto con el tiempo que le toma realizar tales operaciones. La notación **Big-O** es útil para acotar por arriba el tiempo de ejecución de un programa. Si un tiempo de ejecución es $O(f(n))$, entonces para una n suficientemente grande, el tiempo de ejecución es a lo más $k * f(n)$ para alguna constante k . [1]

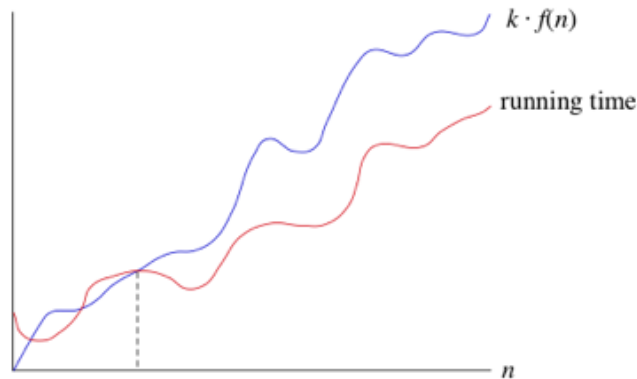


Figura 21: Tiempo de ejecución de un programa acotado por una función $k * f(n)$.

Con el final de observar el comportamiento de ambos algoritmos de ordenamiento frente a una cantidad determinada de entradas, se agregó en el archivo fuente *main.c* dos variables denominadas 'combinaciones' e 'intercambios' con el fin de contabilizar la cantidad de operaciones que realizan ambos algo-

ritmos durante cada iteración. Posteriormente, estas variables fueron enviadas por referencia a las funciones *insertionSort* y *selectionSort* para comenzar a contar en cada iteración el tipo de operación realizada:

```
13 |                                     if(arreglo[j]<arreglo[indiceMenor])
14 |                                         indiceMenor=j;
15 |                                     }
16 |                                     *comparaciones=*comparaciones+1;
17 |                                     if (i!=indiceMenor){
18 |                                         *intercambios=*intercambios+1;
19 |                                         swap (&arreglo[i], &arreglo[indiceMenor]);
20 |                                     }
21 |                                     printf("\nIteracion %i:\n\n", i+1);
```

Figura 22: Líneas de código añadidas para contar las comparaciones e intercambios.

La metodología usada para añadir los contadores fue la siguiente:

1. Se añade un aumento en la variable 'comparaciones' antes de cada ciclo *for* para contar la comparación que arroja un valor falso y hace que no se ejecute el ciclo. Esta misma condición se aplica al ciclo *while* y el condicional *if*.
2. El aumento anterior también se añade en el interior de los ciclos *for* y *while* con el fin de contar las iteraciones donde las condiciones son verdaderas y se ingresa a los ciclos mencionados.
3. La variable 'intercambios' aumentará al ser invocada la función *swap* y al ser realizado el proceso de copiado y pegado en *insertionSort*.

```
{Felicitades! Su arreglo está ordenado. :)}
0 0 0 2 3 5 6 6 7 14 25 27 27 31 31 31 32 37 38 40 43 45 46 48 55 55 60 63 68 68 70 71 72 73 79 79 81 84 84 86 86 89 90 91 91 92 94 94 96 96
Número total de comparaciones: 2598
Número total de intercambios : 44
```

Figura 23: Número total de operaciones obtenido para Selection-Sort con una entrada de 50 números.

5.1. Resultados obtenidos

A continuación se han probado las modificaciones hechas al código fuente para verificar el comportamiento de ambos algoritmos con una entrada de 50, 100, 500 y 1000 números modificando la constante 'N' definida en el código fuente del programa. El programa arrojará el número y el tipo de operación realizada al final de la ejecución del algoritmo de ordenamiento, como puede verse en [23].

Con el fin de mostrar gráficamente los datos obtenidos se ha recurrido al uso de tablas y gráficas, las cuales fueron realizadas por medio del software Microsoft Excel© 2016:

	Insertion-Sort		Selection-Sort	
Entrada	Comparaciones I	Intercambios I	Comparaciones J	Intercambios J
50	1478	665	2598	45
100	5268	2485	10198	93
500	121434	59968	250998	493
1000	491152	244077	1001998	984

Figura 24: Resultados obtenidos para una cantidad variable de entradas.

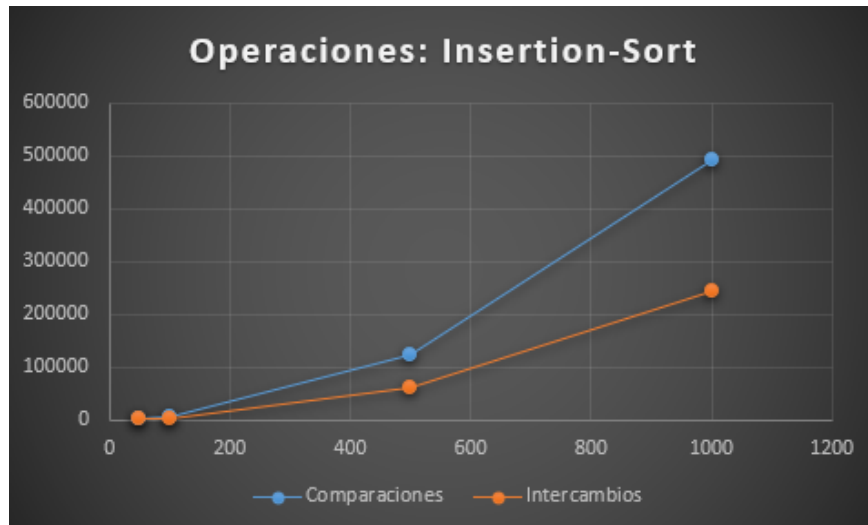


Figura 25: Operaciones realizadas por Insertion-Sort.

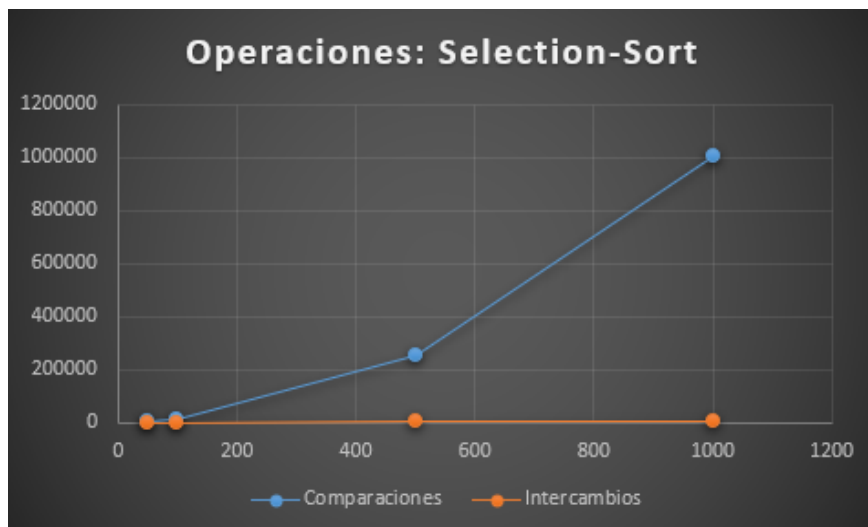


Figura 26: Operaciones realizadas por Selection-Sort.

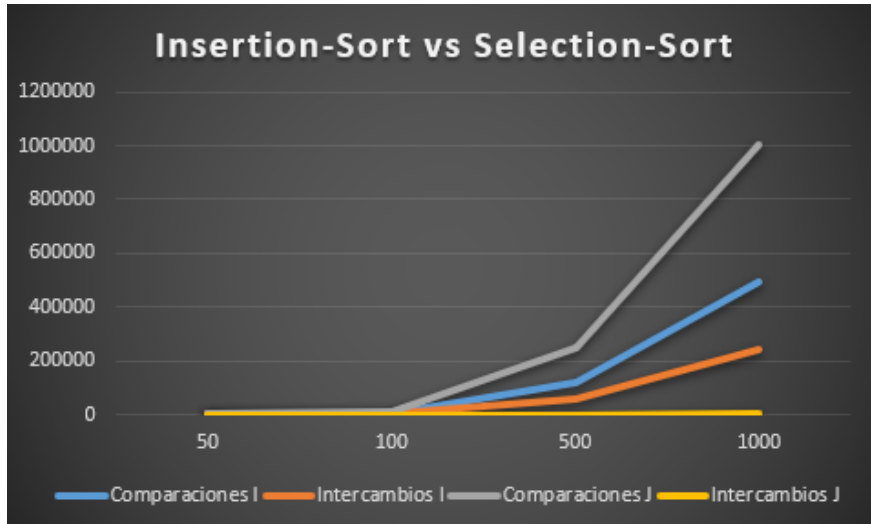


Figura 27: Comparación de las operaciones realizadas por ambos algoritmos.

5.2. Análisis de resultados

Como puede observarse en [25], tanto el número de comparaciones como de intercambios comienzan a crecer a un ritmo rápido; con las comparaciones siendo el mayor tipo de operación realizada. En cambio, puede apreciarse en [26] que el número de intercambios crece a un nivel muy bajo. Por el contrario, el número de comparaciones se eleva notablemente conforme se aumenta el tamaño de la entrada.

Finalmente, en la comparación de los algoritmos mostrada en [28] puede observarse que las operaciones realizadas por Insertion-Sort quedan 'acotadas' por las operaciones de Selection-Sort. Esto tiene como posible explicación el hecho de que ambos algoritmos, en el peor de sus casos, poseen una cota superior de complejidad polinomial $O(n^2)$. Por lo anterior, es esperable que estos algoritmos posean un comportamiento 'semejante' conforme aumenta el tamaño de la entrada.

Como comentario, es importante señalar que el tiempo de ejecución aumenta conforme se incrementa la entrada; aunque de una forma aceptable. En el caso de Insertion-Sort, $N = 50$ no supera un segundo, mientras que $N = 1000$ incrementa el tiempo de espera a 5 segundos aproximadamente. Los tiempos de ejecución para $N = 50$ y $N = 1000$ en Selection-Sort son prácticamente igual.

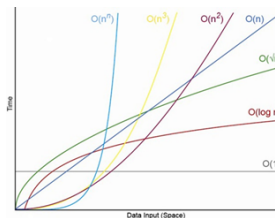


Figura 28: Un tiempo de ejecución polinomial es ideal en la mayoría de situaciones.

El ganador de esta comparativa puede resultar difícil de elegir debido a la similitud de operaciones realizadas por ambos algoritmos. Sin embargo, considerando que Selection-Sort posee una complejidad del orden $O(n^2)$ en todos sus casos y que Insertion-Sort reduce su complejidad a $O(n)$ en su mejor caso, la balanza se inclina finalmente al algoritmo de ordenamiento por inserción.

6. Conclusiones

El desarrollo de las actividades ha resultado ser una actividad muy interesante y, a su vez, retadora. El flujo de trabajo seguido en el desarrollo de esta práctica ha sido completamente diferente al que he estado acostumbrado en cursos previos y me ha exigido un nivel superior.

Hay ciertos puntos que deseo abordar acerca de las dificultades a vencer en esta práctica:

1. He requerido volver a estudiar mis antecedentes de sintaxis en C con el fin de trabajar ágilmente y evitar errores de compilación al trabajar el programa realizado con los archivos brindados.
2. Mi flujo de trabajo se entorpeció notablemente durante el laboratorio debido a la relativa poca habilidad que poseo en el S.O. macOS y en sistemas Unix en general: tuve fuertes problemas para incluir los archivos `.h` en mi programa. Tal dificultad fue superada haciendo uso de Windows 10 e incluyendo los archivos en la carpeta *include* del compilador *Cygwin*.

Esta es una deficiencia a vencer con el fin de mejorar mi flujo de trabajo en el laboratorio de la asignatura.

3. Esta práctica marcó un cambio notable en la presentación de mis reportes de prácticas, pues es la primera que realicé completamente en \LaTeX . La experiencia ha sido satisfactoria y espero poder obtener retroalimentación con el fin de mejorar cada vez más la presentación del trabajo realizado.

Las actividades realizadas resultaron una aplicación directa de los algoritmos analizados en pseudocódigo durante la clase teórica. Además, los ejercicios hechos en clase han demostrado ser muy útiles a la hora de comenzar el análisis de las implementaciones en C y Java, lo cual enriquece mucho el conocimiento adquirido teóricamente. El *bonus* de Java también resultó agradable y una buena introducción al lenguaje que será utilizado próximamente durante el resto del curso.

Finalmente, considerando que se ha realizado un análisis completo de la estructura de las implementaciones en ambos lenguajes, se ha logrado crear un programa funcional y correcto que pone a prueba tales implementaciones y que, además, se ha verificado el número de operaciones que ambos algoritmos realizan para compararlos, es posible concluir que los objetivos propuestos al inicio de este reporte han sido cumplidos con éxito. Así mismo, se ha cumplido en su totalidad con los ejercicios propuestos.

Sin duda, considero que la experiencia obtenida de esta práctica tendrá un impacto profundo en los próximos reportes y trabajos que realice en esta asignatura o en otras asignaturas de mi carrera.



Referencias

- [1] Notación O grande (Big-O). Recuperado de: <https://es.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation>. Fecha de consulta: 08/02/2020.

Los créditos de las fotografías pertenecen a sus respectivos autores. ©