|  |  |  |
| --- | --- | --- |
|  | **Carátula para entrega de prácticas** | |
| Facultad de Ingeniería | | Laboratorio de docencia |

Laboratorios de computación

salas A y B

|  |  |
| --- | --- |
| *Profesor:* | M.I Edgar Tista García |
| *Asignatura:* | Estructura de Datos y Algoritmos II |
| *Grupo:* | 9 |
| *No de Práctica(s):* | 1 |
| *Integrante(s):* | Díaz Hernández Marcos Bryan |
| *No. de Equipo de cómputo empleado:* | Elaborado en casa/Equipo personal. |
| *No. de Lista o Brigada:* | 9 |
| *Semestre:* | 2021-1 |
| *Fecha de entrega:* | 3 de octubre de 2020 |
| *Observaciones:* |  |
|  |  |

CALIFICACIÓN: \_\_\_\_\_\_\_\_\_\_

**Objetivo de la practica**

El estudiante identificará la estructura de los algoritmos de ordenamiento por selección y por inserción.

**Introducción**

En esta práctica usted podrá visualizar la forma en que interprete los algoritmos de ordenamiento, los analice, y los traducí a la forma en que es mas simple para mi entenderlos, espero sea de su agrado.

**Ejercicios de la practica:**

* Ejercicio 1:

El primer ejercicio consiste en analizar los códigos y las bibliotecas que se proporcionan para resolver el segundo ejercicio.

* Dificultades en el código

Al querer ejecutar los códigos c, surge un problema con la compilación y por lo tanto no se ejecuta (Imagen 1).

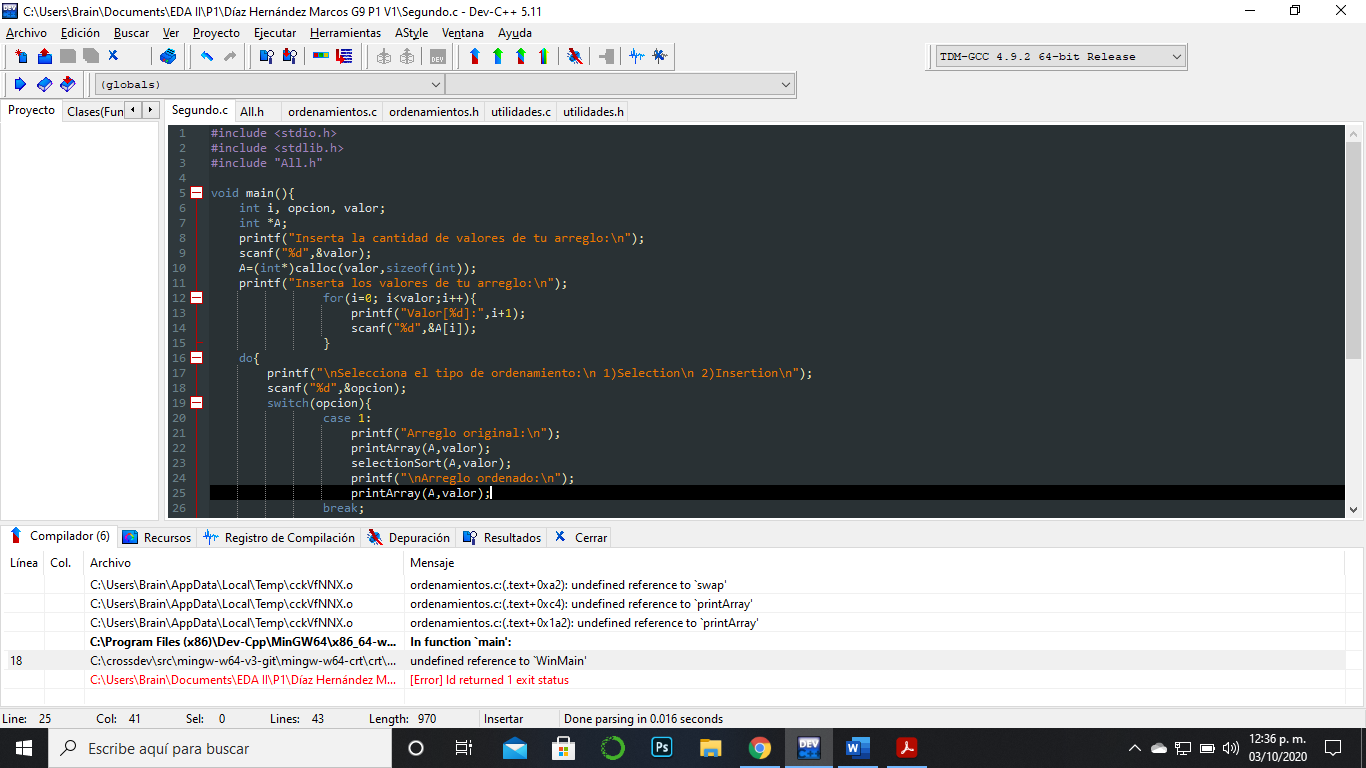


Imagen 1

Ahora el problema es que las librerías que se dan por default para el primer ejercicio tienen una forma bastante extraña de declaración de las funciones que se van a utilizar, porque se menciona en el error que no están definidas las funciones y esto es porque en las librerías (.h) se encuentra únicamente el prototipo de las funciones y en el código (.c), se encuentra el desarrollo completo de las sentencias, por lo que cuando se invocan las bibliotecas solo conocen el prototipo pero no saben cuáles son las sentencias, por lo tanto no están definidas (Imagen [2,3]).

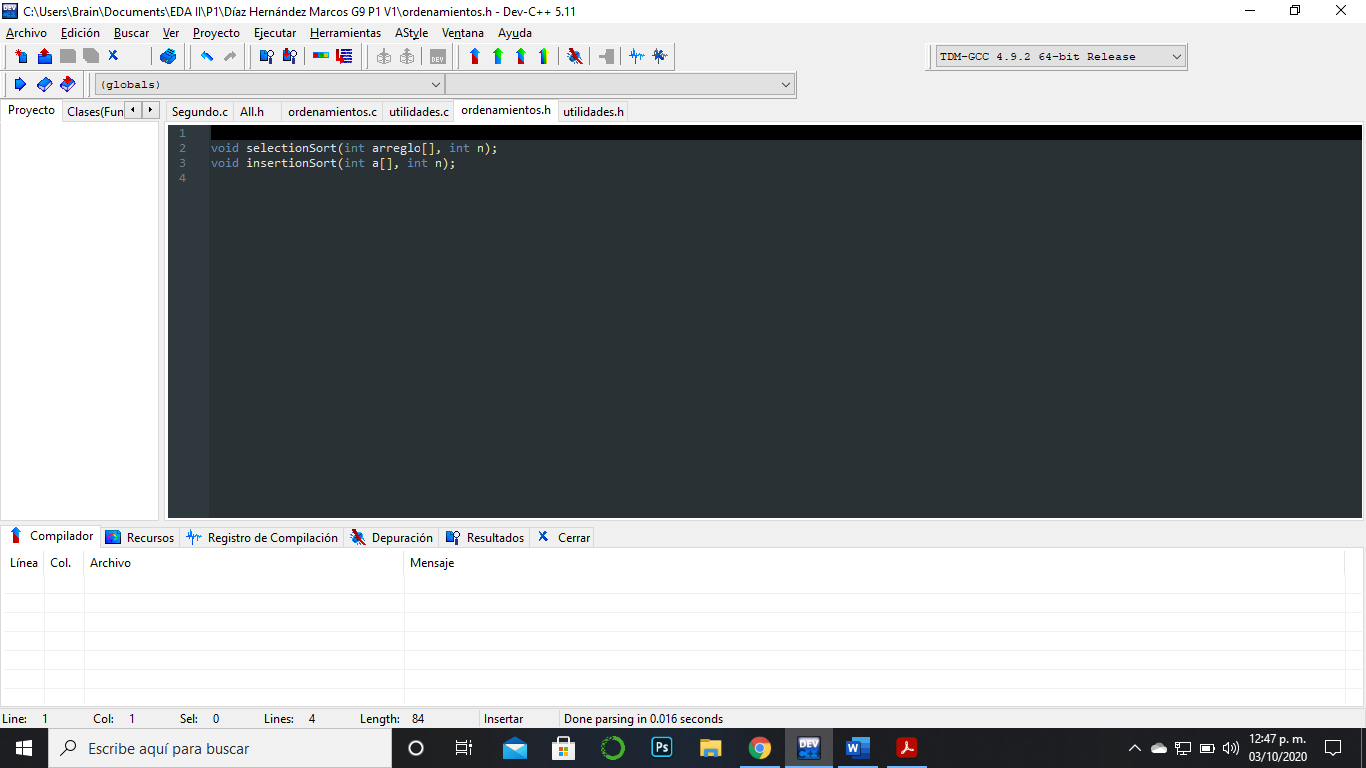


Imagen 2

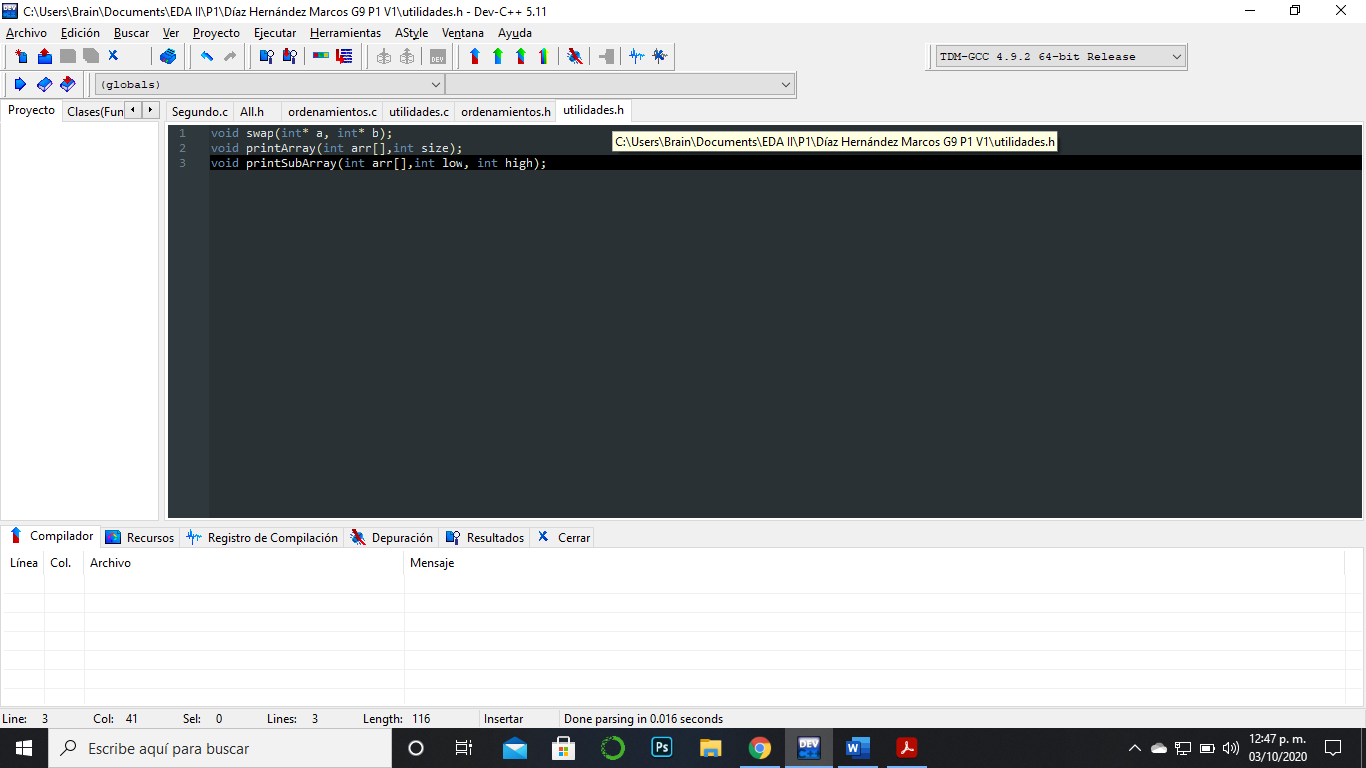


Imagen 3

* Solución de las dificultades:

Para poder resolverlos únicamente es necesario el decirles a las funciones que tienen que hacer y de ahí el vincular el programa principal con las librerías creadas y de esta forma el problema se elimina y podemos continuar con la ejecución de las demás actividades.

Además lo que hice como primera opción el colocar los prototipos y las sentencias de cada función en una sola biblioteca para que funcionase, esto es una alternativa y se puede aplicar lo mismo con cada biblioteca (utilidades y ordenamientos), como una segunda opción.

Es decir en utilidades colocar todas las funciones completas y en ordenamientos, colocar la biblioteca “utilidades.h”, y ahí definir las otras funciones de ordenamiento. Como están ligadas las bibliotecas al solicitarlas en el siguiente ejercicio, simplemente irán en busca de una dentro de la otra, y al estar definidas ahí las encontrara. (Imagen 1)

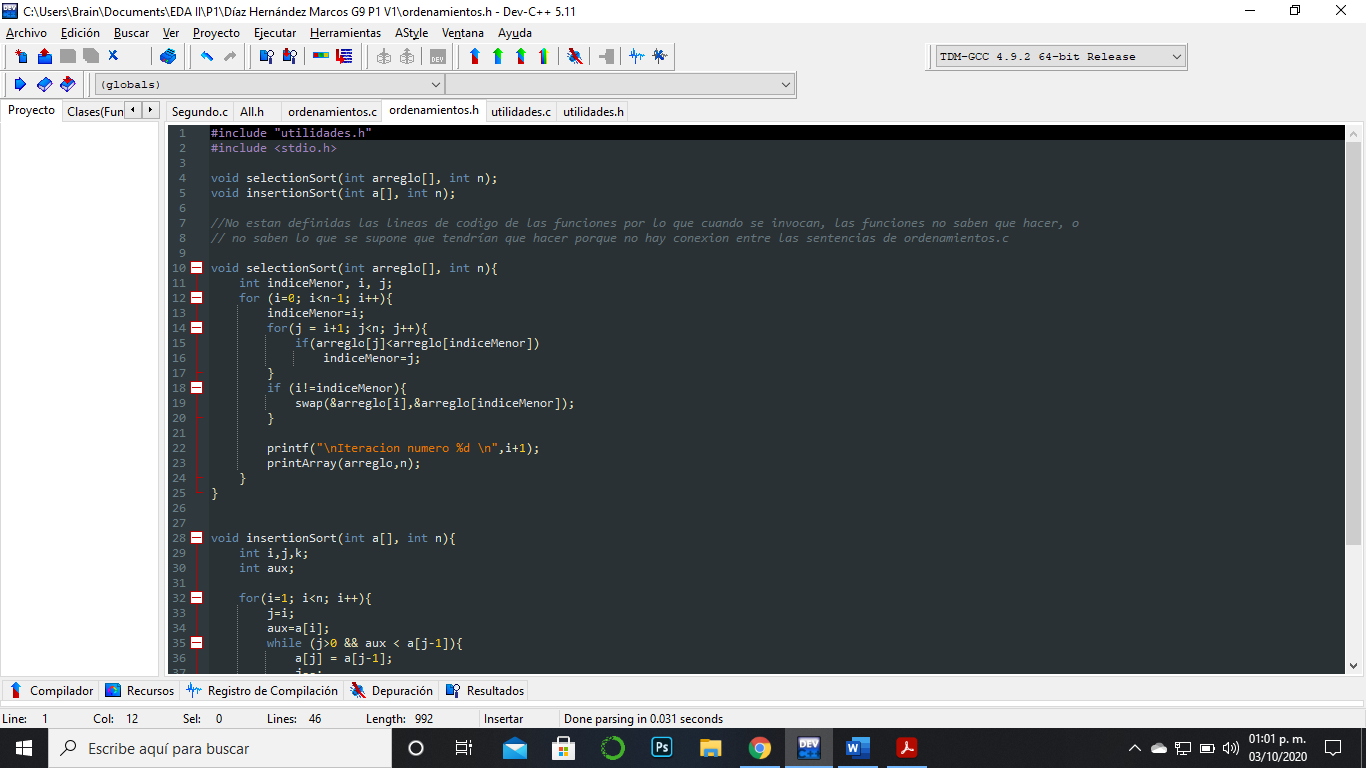


Imagen 1

* Diagrama de funcionamiento:

Las funciones más importantes fueron las de Insertion y Selection porque son las que utilizan las utilidades, de la siguiente forma:

SegundoEjercicio

De forma sencilla se explica la jerarquía de las funciones, donde se llama primero a Ordenamientos() y de ahí las funciones Insertion() y Selection() mandan a llamar las funciones de Utilidades.h

“Ordenamientos.h”

Insertion()

Selection()

“Utilidades.h”

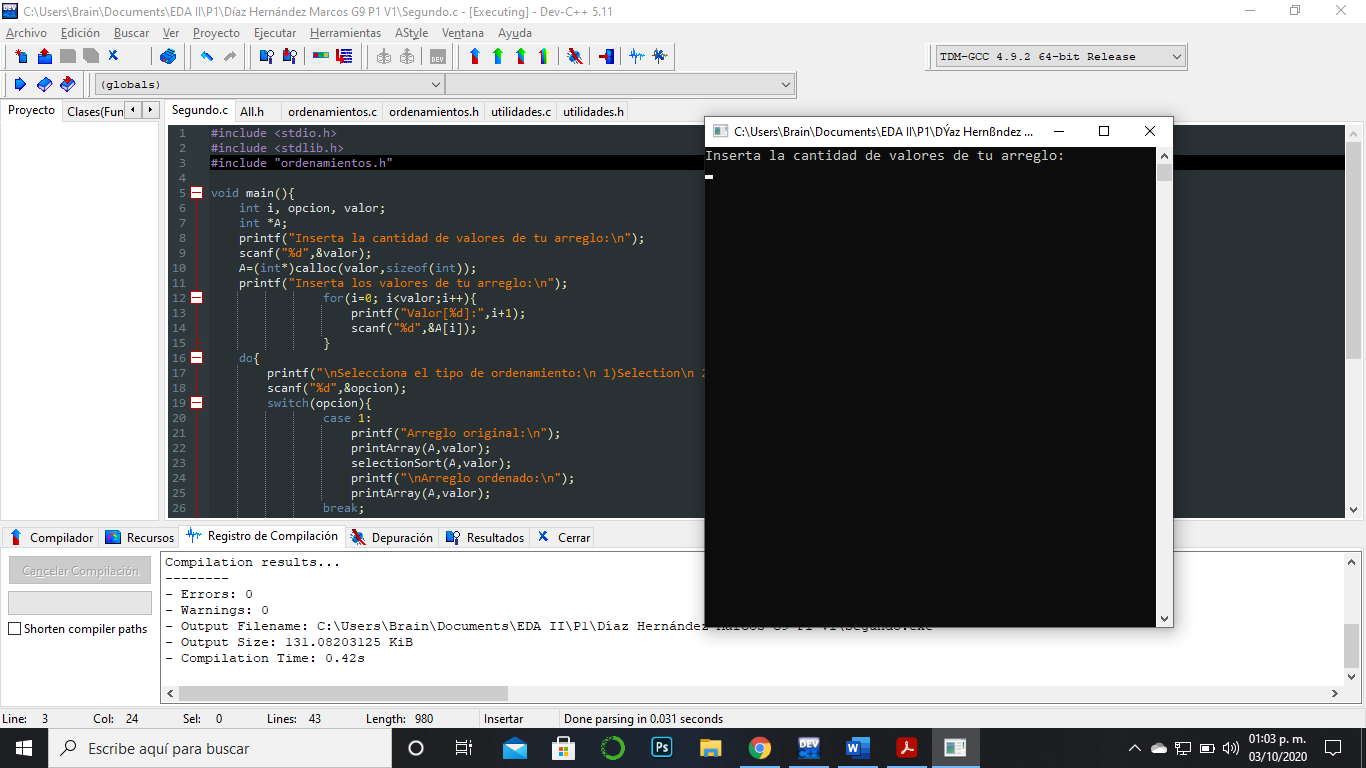
Swap()

PrintArray()

* Relación con la teoría:

Esto es más que nada un repaso de las funciones y de las librerías que se pueden crear, además de entender el funcionamiento de los algoritmos solamente con el análisis, de ahí que se puede descubrir el error de las librerías, lo que siento fue el principal reto de aquí y saber cómo se relacionan para poder utilizarlas en el segundo ejercicio.

* Evidencia de implementación



* Ejercicio 2:

El segundo ejercicio consistía en realizar un pequeño menú para que el usuario pudiera seleccionar e ingresar una cantidad de elementos en un arreglo y con la selección del algoritmo de ordenamiento de su preferencia obtener la lista ordenada. Por supuesto mostrando los cambios que se realizaron al arreglo y mostrando la forma inicial y la final del algoritmo.

* Dificultades en el código

El problema principal que encontré dentro de la clase fue que no detectaba las funciones, pero eso era un problema que se tenía que resolver en el primer ejercicio, y como mencionaba anteriormente los resolví de esa manera que mencioné. Ahora con las características que se requerían tuve que repasar algunos elementos para elaborar la selección y la creación del arreglo dinámico, pero los demás no fue difícil, de hecho estuvo bastante sencillo, pero esa parte del primer ejercicio influyo mucho en este ejercicio (Imagen 1).

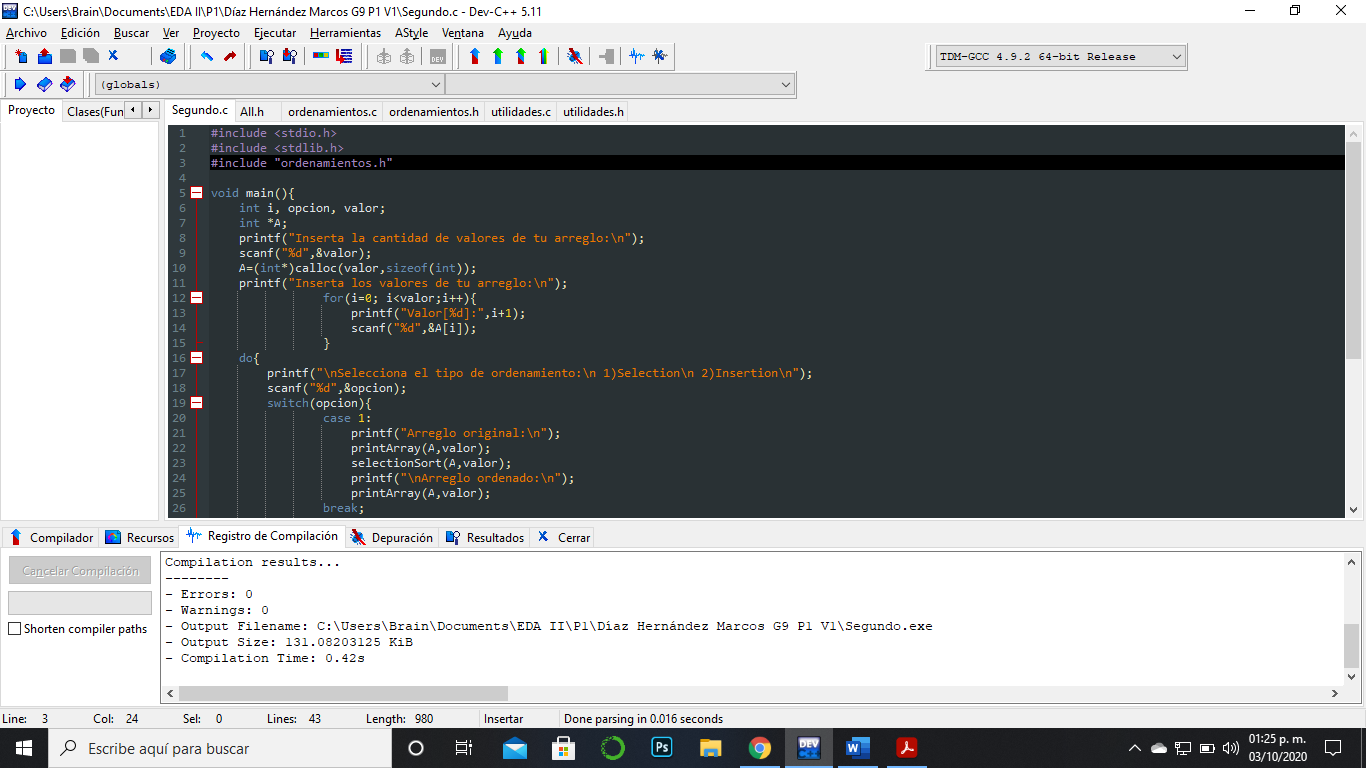


Imagen 1

* Diagrama de funcionamiento:

1. Creación y llenado del arreglo.

A=(int\*)calloc(5,sizeof(int));

Arreglo

1. SelectionSort

Se utiliza el arreglo

Se llena el arreglo: for(i=0;i<5;i++)

Creación de Arreglo = int \*A

|  |
| --- |
| 10 |
| 15 |
| 9 |
| 18 |
| 16 |

SelectionSort

For(i=0;i<4;i++)

Selection(A,valor)

IndiceM=0

If(A[1]<A[0])

For(j=1;j<5;j++)

If(15<10) = No

If(A[2]<A[0])

For(j=2;j<5;j++)

If (9<10) = Si

Se busca el elemento más pequeño para colocarlo en la primera posición por medio de comparaciones, y esto se repite.

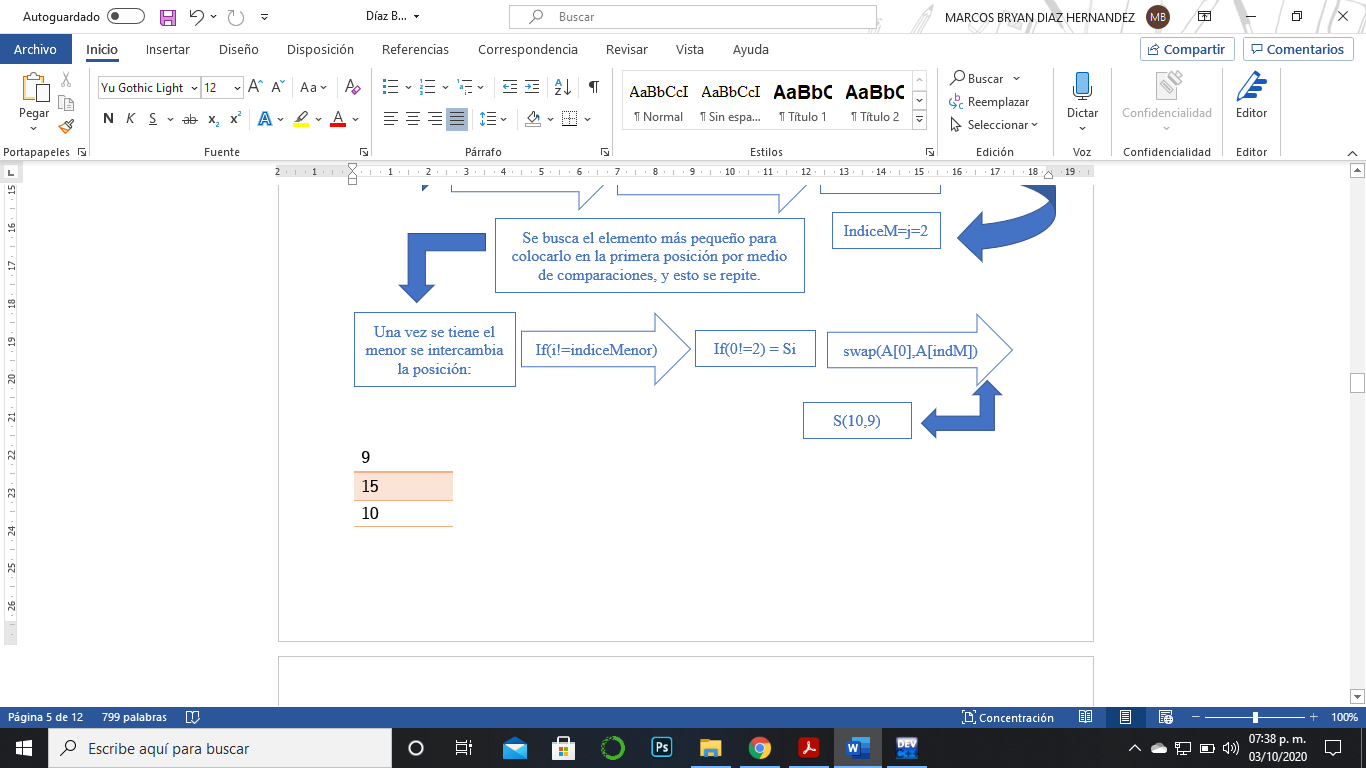
IndiceM=j=2

swap(A[0],A[indM])

If(0!=2) = Si

If(i!=indiceMenor)

Una vez se tiene el menor se intercambia la posición:



Es un intercambio de posiciones y se repite hasta estar en orden el arreglo.

Por motivos didácticos utilizare el mismo arreglo como ejemplo para ambos algoritmos de ordenamiento.

1. InsertionSort

while (j>0 && aux < A[j-1])

j=i=1

aux=A[i]=10

For(i=1;i<5;i++)

InsertionSort

For(i=2;i<5;i++)

A[j]=aux /A[1]=10

While(1>0 && 10<algo) = No

El proceso compara con los elementos de la izquierda, uno por uno hasta que encuentra valor mayor al del aux, y sigue comprobando hasta estar seguro de tener la posición de los elementos más grande y pequeño de la izquierda.

while (3>0 && 9<15)=Si

For(i=3;i<5;i++)

j=i=3

aux=A[3]=9

while (2>0 && 9<10)=Si

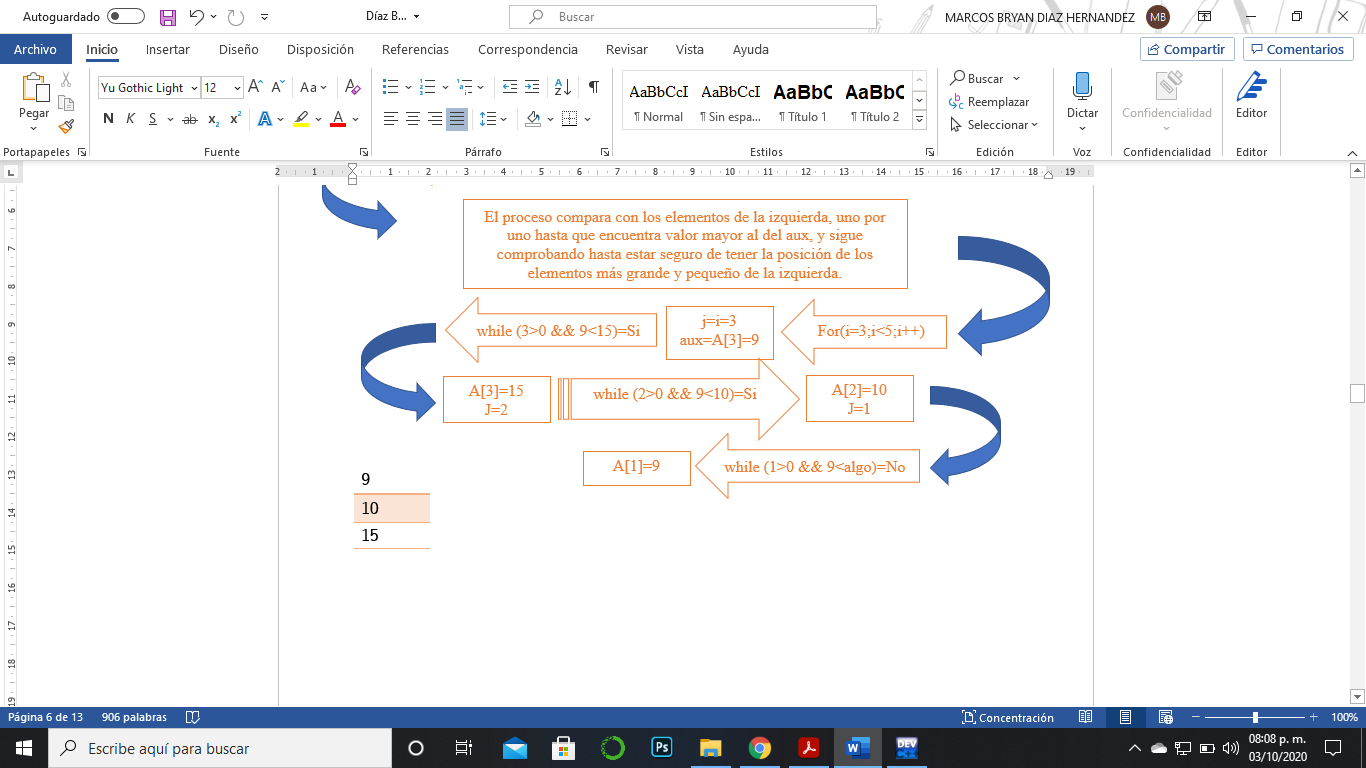
A[2]=10

J=1

A[3]=15

J=2

while (1>0 && 9<algo)=No



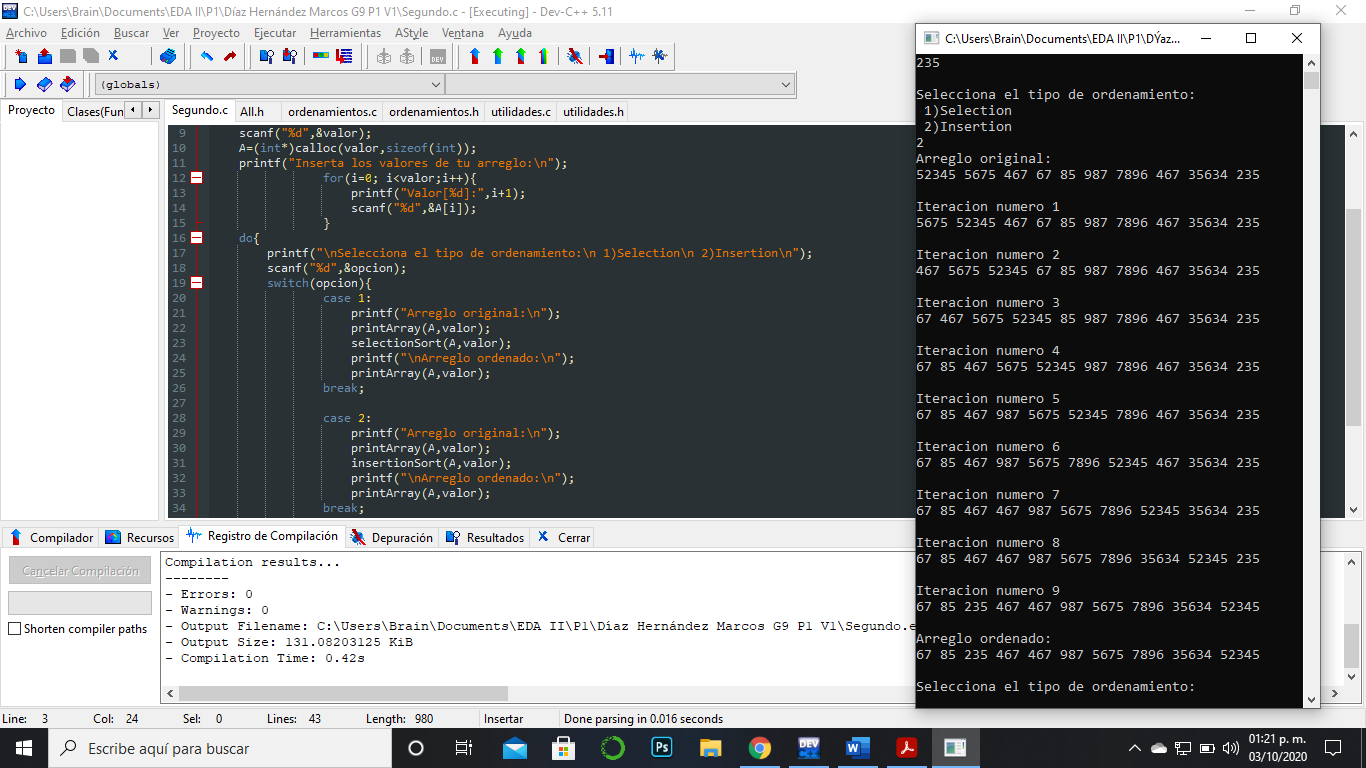
A[1]=9

Es un ordenamiento de posiciones y se consideran los valores del arreglo por lo que es más tardado, pero más eficiente en cantidades grandes.

* Relación con la teoría:

Estos algoritmos fueron dos los primeros que vimos en la clase y tienen sus puntos buenos por supuesto, pero estos diagramas y el ejercicio me ayudaron a comprender mejor como es su funcionamiento e incluso describirlo por medio de un diagrama por lo que está íntimamente relacionado con la teoría que vimos, y más con el código que se ponía de cada uno de estos.

* Evidencia de implementación



* Ejercicio 3:

En este ejercicio, se pide el ejecutar el proyecto correspondiente a Ejercicio 3, Practica1, bueno es describir lo que sucede en el programa y añadir los comentarios pertinentes a la ejecución y a elementos que me parezcan interesante o desconocidos.

* Diagrama de funcionamiento:

En esta sección del ejercicio pondría los diagramas pero ya los he elaborado anteriormente aunque solo voy a colocarlos con las diferencias que tienen entre Java y C, para que tenga yo una idea más clara de los que sucede en el programa.

SelectionSort

SelectionSort

For(i=0;i<4;i++)

Selection(A,valor)

IndiceM=0

If(A[1]<A[0])

For(j=1;j<5;j++)

If(15<10) = No

If(A[2]<A[0])

For(j=2;j<5;j++)

If (9<10) = Si

IndiceM=j=2

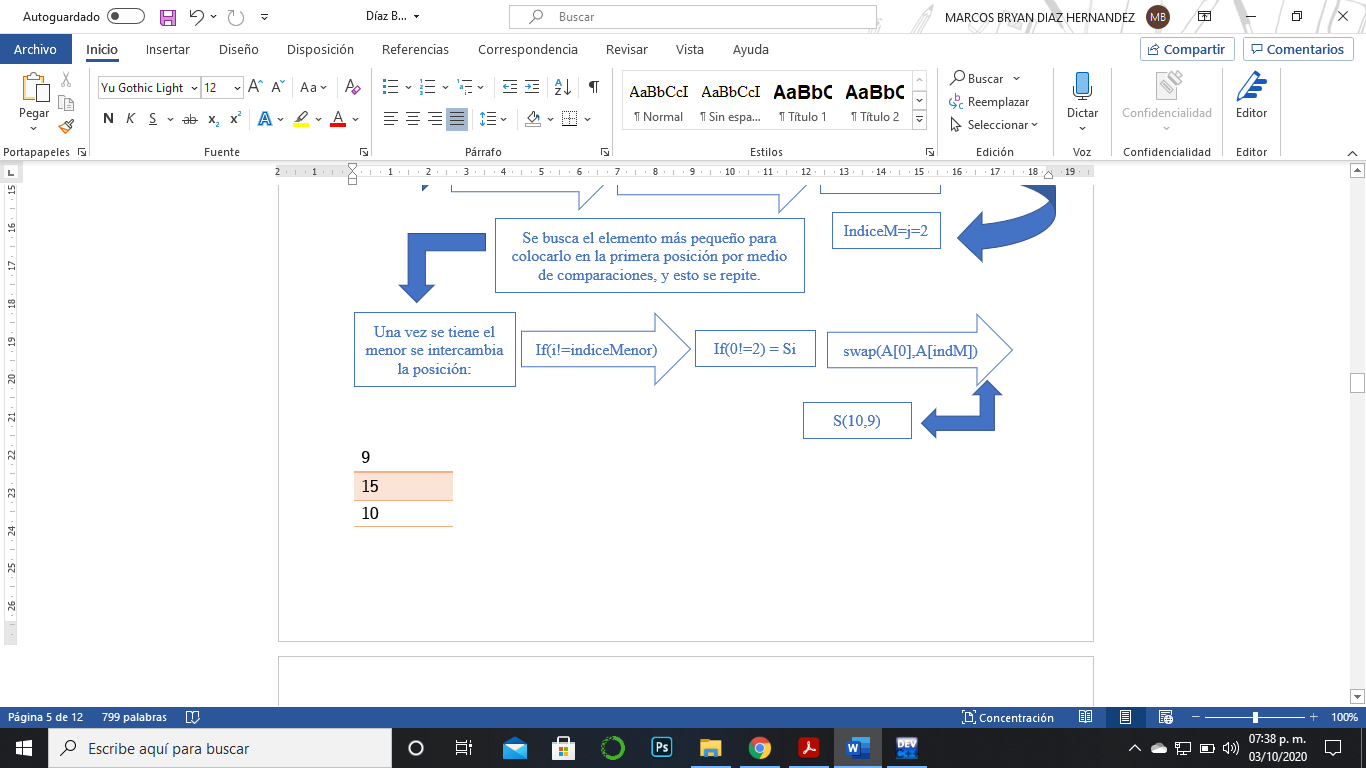
Se busca el elemento más pequeño para colocarlo en la primera posición por medio de comparaciones, y esto se repite.

swap(A[0],A[indM])

If(0!=2) = Si

If(i!=indiceMenor)

Una vez se tiene el menor se intercambia la posición:



Es un intercambio de posiciones y se repite hasta estar en orden el arreglo.

N=A.length (Guarda la longitud del arreglo / similar a Python)

Utilerias.intercambiar(A, i, min)

Utiliza el operador de las estructuras que podemos crear en C.

Está dividido por clases lo que me recuerda que pueden ser accesibles a las otras solo si son públicas.

Elementos distintos:

InsertionSort

while (j>0 && aux < A[j-1])

j=i=1

aux=A[i]=10

For(i=1;i<5;i++)

InsertionSort

For(i=2;i<5;i++)

A[j]=aux /A[1]=10

While(1>0 && 10<algo) = No

El proceso compara con los elementos de la izquierda, uno por uno hasta que encuentra valor mayor al del aux, y sigue comprobando hasta estar seguro de tener la posición de los elementos más grande y pequeño de la izquierda.

while (3>0 && 9<15)=Si

For(i=3;i<5;i++)

j=i=3

aux=A[3]=9

while (2>0 && 9<10)=Si

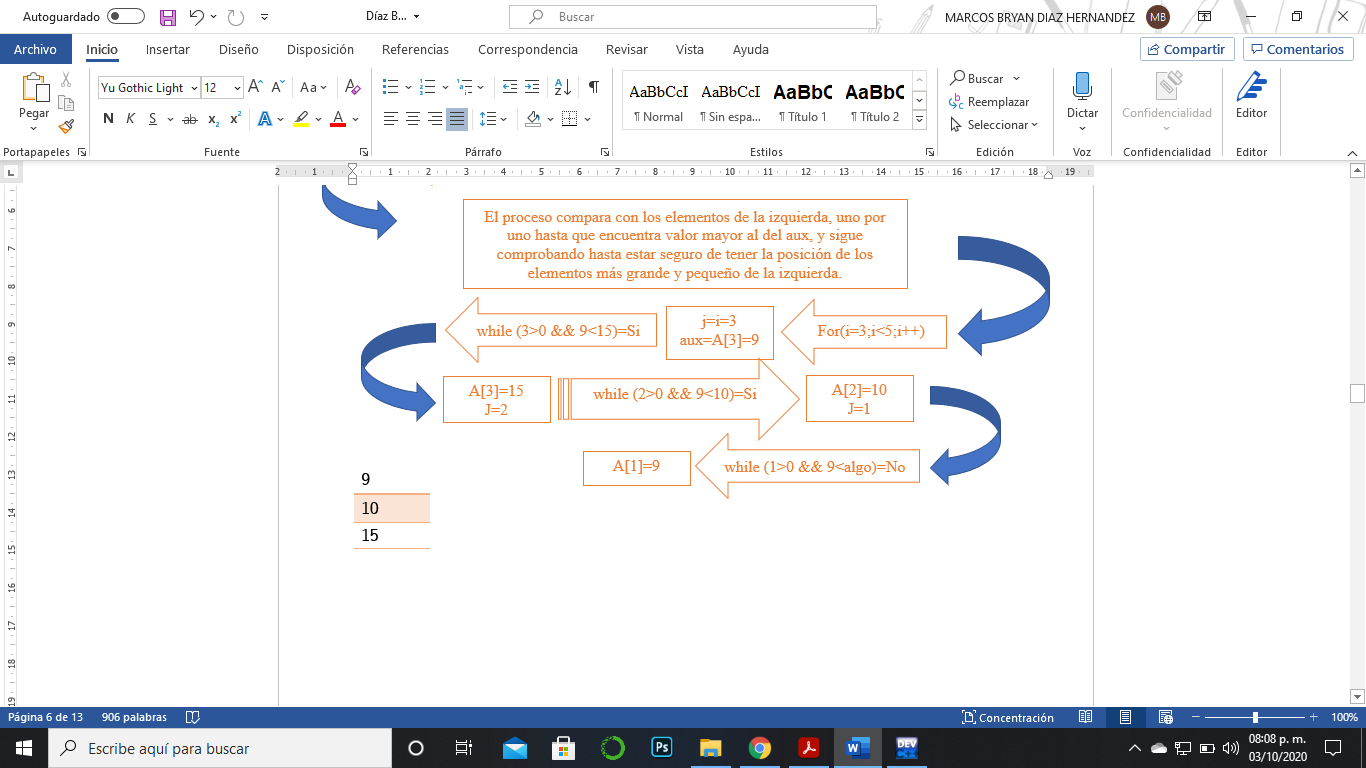
A[2]=10

J=1

A[3]=15

J=2

while (1>0 && 9<algo)=No



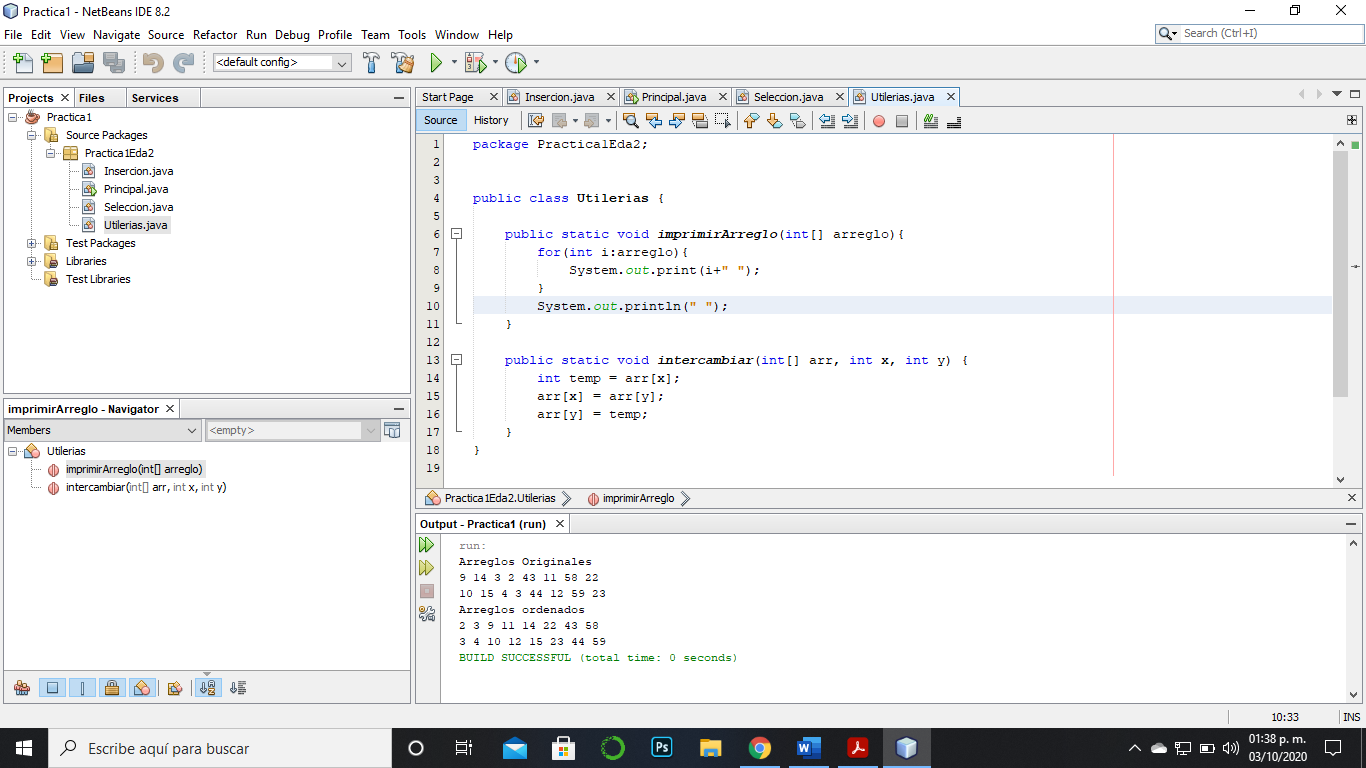
A[1]=9

* Relación con la teoría:

Bueno esto es como un viaje entre dimensiones paralelas, lo que sucede es que las dimensiones no necesariamente tienen una interpretación del tiempo ni de los hechos iguales, y lo digo no porque vea muchas series o algo así, sino que esto es parte de POO y EDA, la situación aquí es que en mi clase de POO, casi no hemos tocado JAVA, además que vamos atrasados una clase, pero pronto voy a disponer de mi tiempo libre a estudiar y programar en JAVA.

Sin embargo, esto es totalmente relacionado con los algoritmos de ordenamiento solo que implementados en dos paradigmas distintos y siendo honesto el tener varias pestañas que separen las funciones es mucho más agradable que estar bajando sobre líneas de código.

* Evidencia de implementación



* Ejercicio 4:

El último ejercicio consistía en colocar contadores/instrucciones que nos ayudaran a poder llevar un control de las diferentes operaciones que se llevan a cabo en el programa al momento de la ejecución además de que todas estas instrucciones serán usadas para el análisis de complejidad de cada algoritmo.

* Dificultades en el código

Hubo una parte donde me entretuve como una hora pensando, entre buscando como solucionarlo y haciendo otras cosas para poder despejar la mente y tener más claridad, para poder encontrar la solución, pero no la encontré.

El problema vino en el algoritmo de Insertion, porque quería contar las veces que los números cambiaban de posición, pero mi error fue el considerar los algoritmos como similares, y al final me di cuenta de que el Insertion hace inserciones y para cada número que cambia de posición hace dos operaciones, por lo que este último detalle me llevo a una forma de estimar los números que se movían, no decir cuales eran, sino cuantas veces se intercambiaron de lugar.

Igual intente el usar otro arreglo y compararlos uno que fingiera ser el original sin cambios y el otro ya ordenado, pero al comparar las posiciones no me dieron lo esperado, además que los números aleatorios no me permitían tener un estudio más claro de lo que sucedía en cada ejecución.

* Diagrama de funcionamiento:

El programa solo contiene contadores, para poder verificar las operaciones realizadas, pero voy a agregar la función de números aleatorios aquí para que se pueda ver como realice la función y que ayuda bastante a la solución de los casos de 50,100,500 y 1000, elementos.

La función al funciona para crear una formación de números en un rango extenso que se limita por medio del módulo, y evita la repetición de los valores aleatorios.

Srand(time(NULL))

for (i=0; i<=valor; i++)

A[i] = rand()%10000000;

Contadores añadidos

Cambios de índice: contadorM

Número de comparaciones entre elementos: contadorC

Número de cambio del indiceMenor: contadorI

Número de intercambios: contadorS

Número de posibles intercambios: contadorF

SelectionSort

Número de inserciones/comparaciones totales: (UNO + DOS)

Número inserciones que llevaron a la posición preliminar: UNO

Número de inserciones en posición final: (UNO + DOS)/2

InsertionSort

* Análisis de la complejidad
  + SelectionSort

|  |  |
| --- | --- |
| ENTRADA | TIEMPO(s) |
| 50 | 13.33 |
| 100 | 9.618 |
| 500 | 47.74 |
| 1000 | 100.8 |

|  |  |
| --- | --- |
| Operaciones | *DATOS* |
| 1255 | *50* |
| 4950 | *100* |
| 124750 | *500* |
| 499500 | *1000* |

En base a las gráficas y a las pruebas realizadas, en algunas ocasiones los tiempos de ejecución tendían a ser menores, y esto es debido al procesador de la computadora y el sistema operativo de la misma, además del rendimiento.

En este caso SelectionSort parece tener una complejidad lineal, que tiende a ser cuadrática, esto va en función de los valores que tiene que ordenar y al se aleatorios pues se tiene un caso promedio, con O(n2), eso en la primera gráfica, ya que los tiempos de ejecución se duplican en base a la cantidad de la entrada.

En la segunda gráfica se tienen los datos que se dan y las operaciones que se realizan con estos, de hecho, igual se puede ver un crecimiento equivalente de 50 a 100, pero cuando llega a 500 miembros del arreglo, la proporción se pierde y comienza a ser mas compleja, por lo que en grandes cantidades es más complejo de utilizar.

* + InsertionSort

|  |  |
| --- | --- |
| ENTRADA | TIEMPO(s) |
| 50 | 5.544 |
| 100 | 11.24 |
| 500 | 32.5 |
| 1000 | 94.64 |

|  |  |
| --- | --- |
| Operaciones | *DATOS* |
| 515 | *50* |
| 2285 | *100* |
| 62772 | *500* |
| 250001 | *1000* |

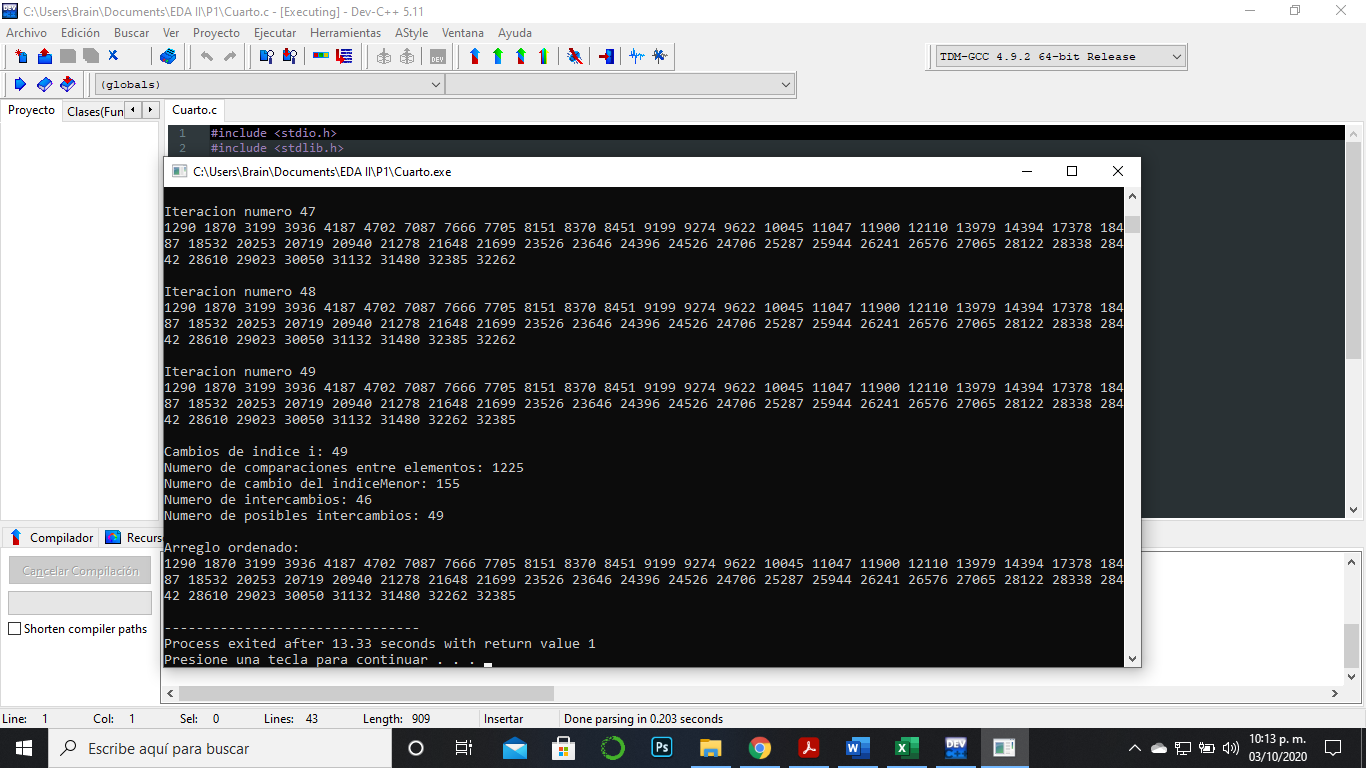
En la primer grafica es posible ver la tendencia lineal/cuadrática de la función, debido a que los tiempos en base a la entrada, son proporcionales, de hecho son mas uniformes que los del SelectionSort, ya que considera a los elementos anteriores ya ordenados, por lo que en cantidades pequeñas puede obtener ventaja, pero la grafica de Insertion esta mas inclinada a ser cuadrática por lo que la complejidad tiende a ser 0(n2).

En cuanto a la segunda grafica que se basa en las operaciones por entrada, estas se cuatriplican aproximadamente por cada duplicación (50-100), pero de 100 a 500, se vuelve 30 veces lo de 100 elementos de entrada, y al doble de 500, se cuatriplica, por lo que las operaciones se multiplican al subir la entrada, con un tendencia lineal, lo que sería algo lógico si se considera que hace comparaciones entre los miembros que ya están ordenados.

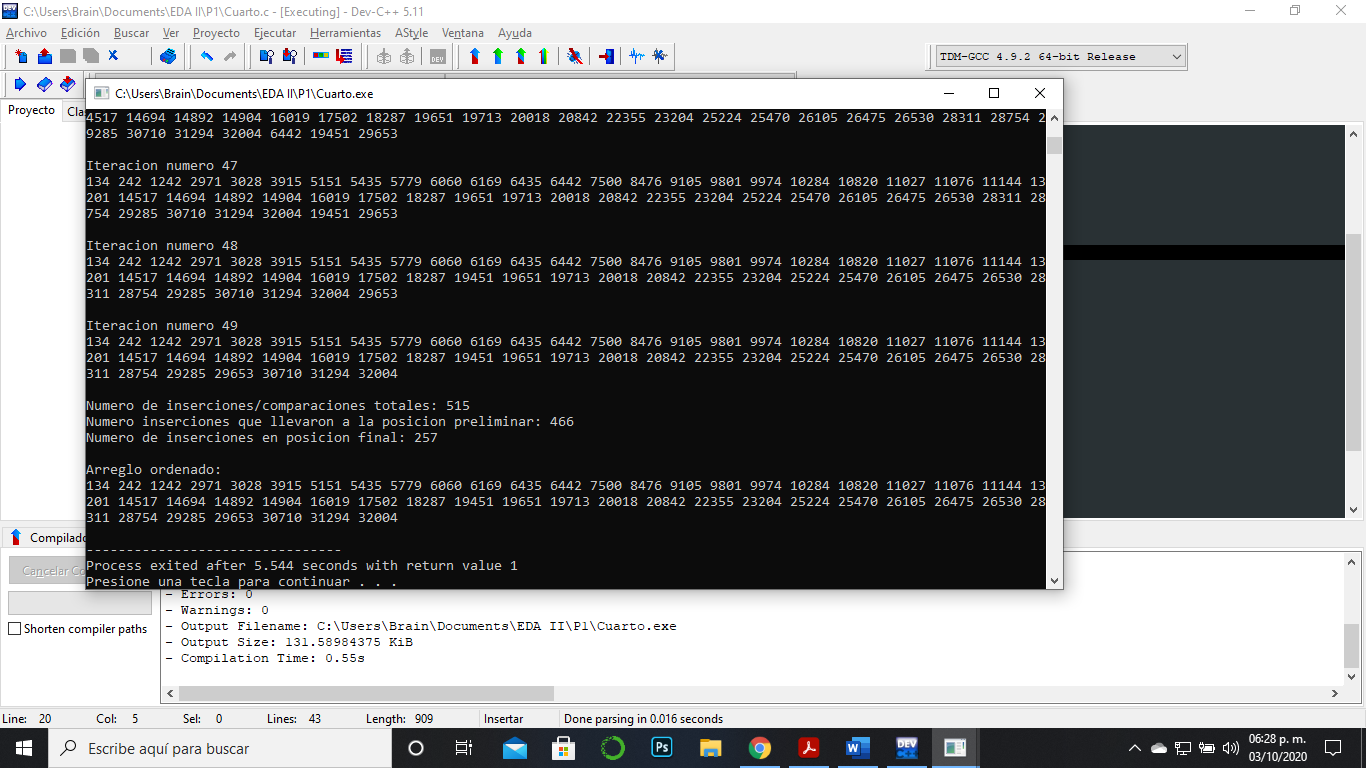
* Relación con la teoría:

En este momento esta es la parte mas relevante de los algoritmos y creo es un elemento importante de que sean utilizados o no, y esto es porque la complejidad de los algoritmos dicta que tan eficiente es utilizarlo o no, además de ser un tema del semestre anterior, toma su verdadero peso al estar dentro de los algoritmos. Además que nos permite darnos una idea de como se realizan los análisis superficiales de los algoritmos cuando se busca su complejidad.

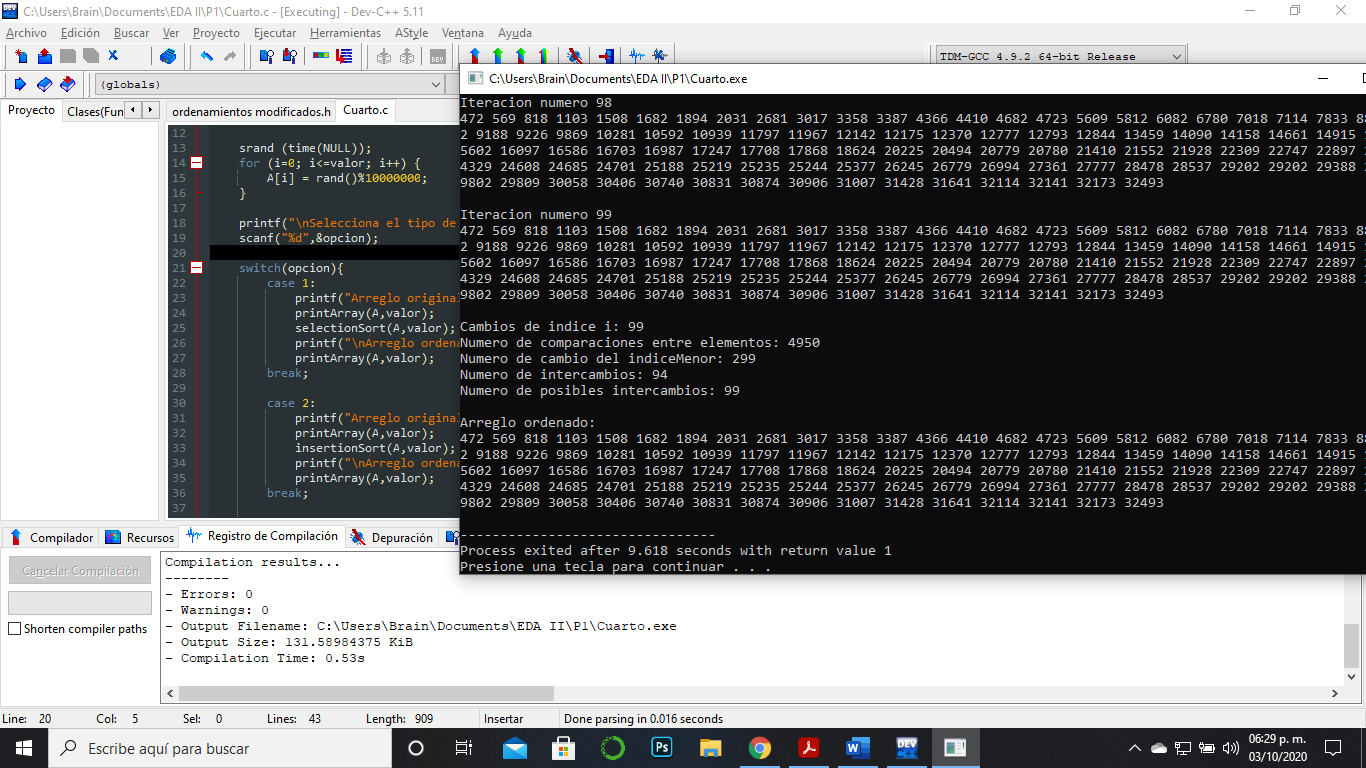
* Evidencias de implementación



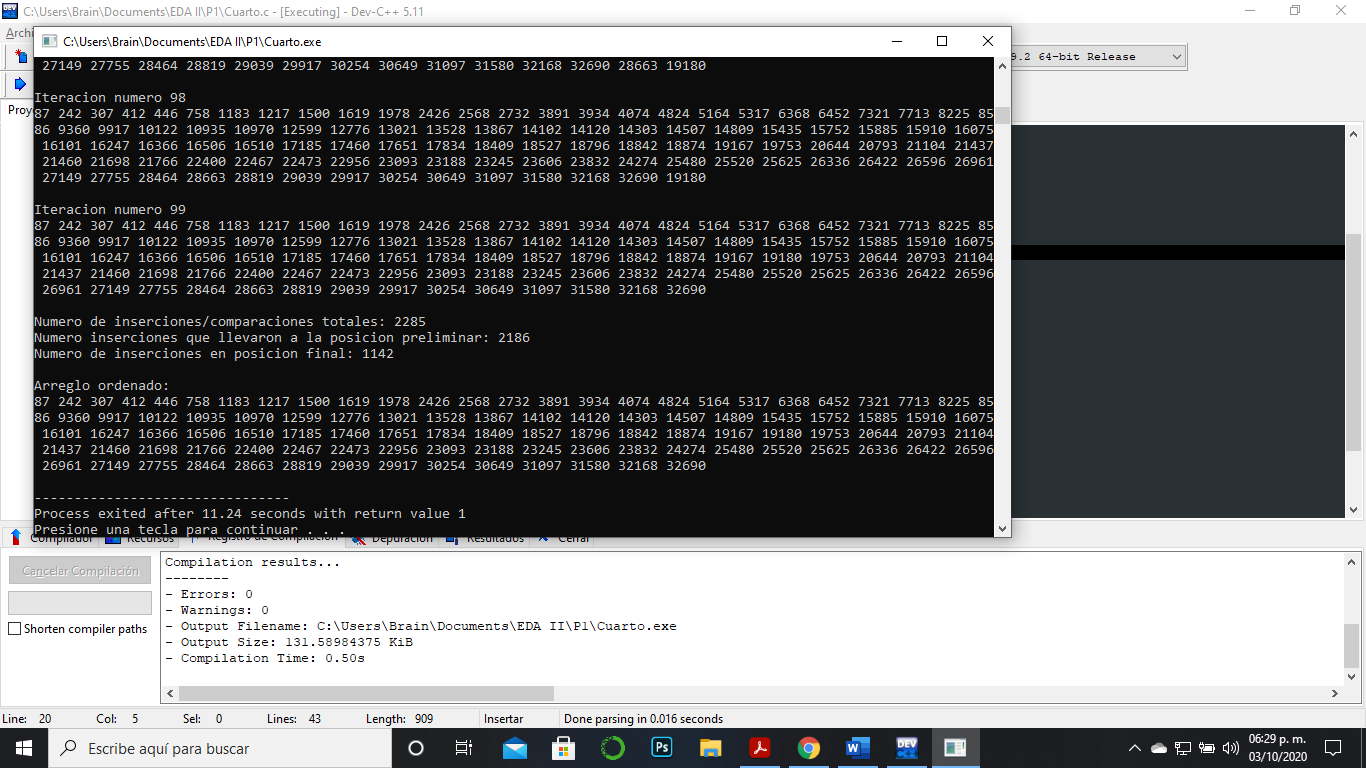
50 -SelectionSort



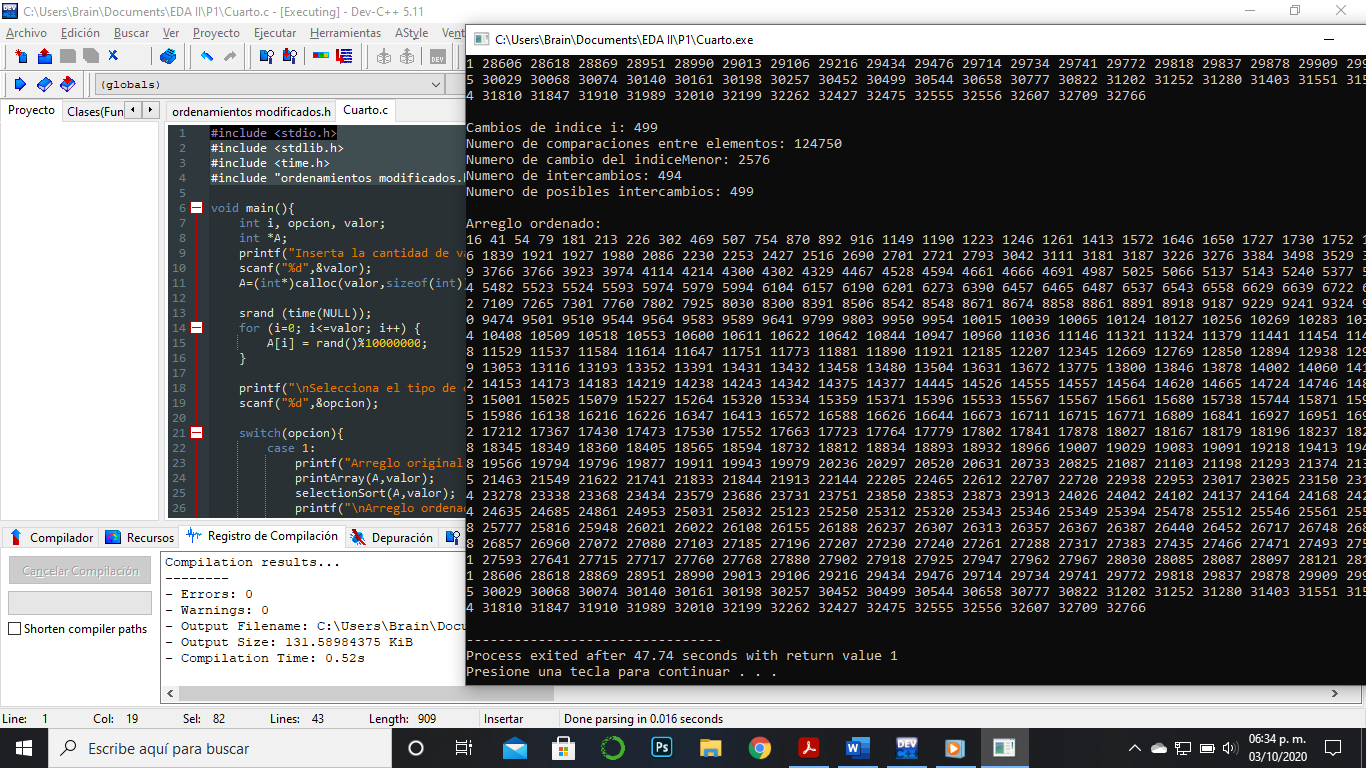
50 – InsertionSort



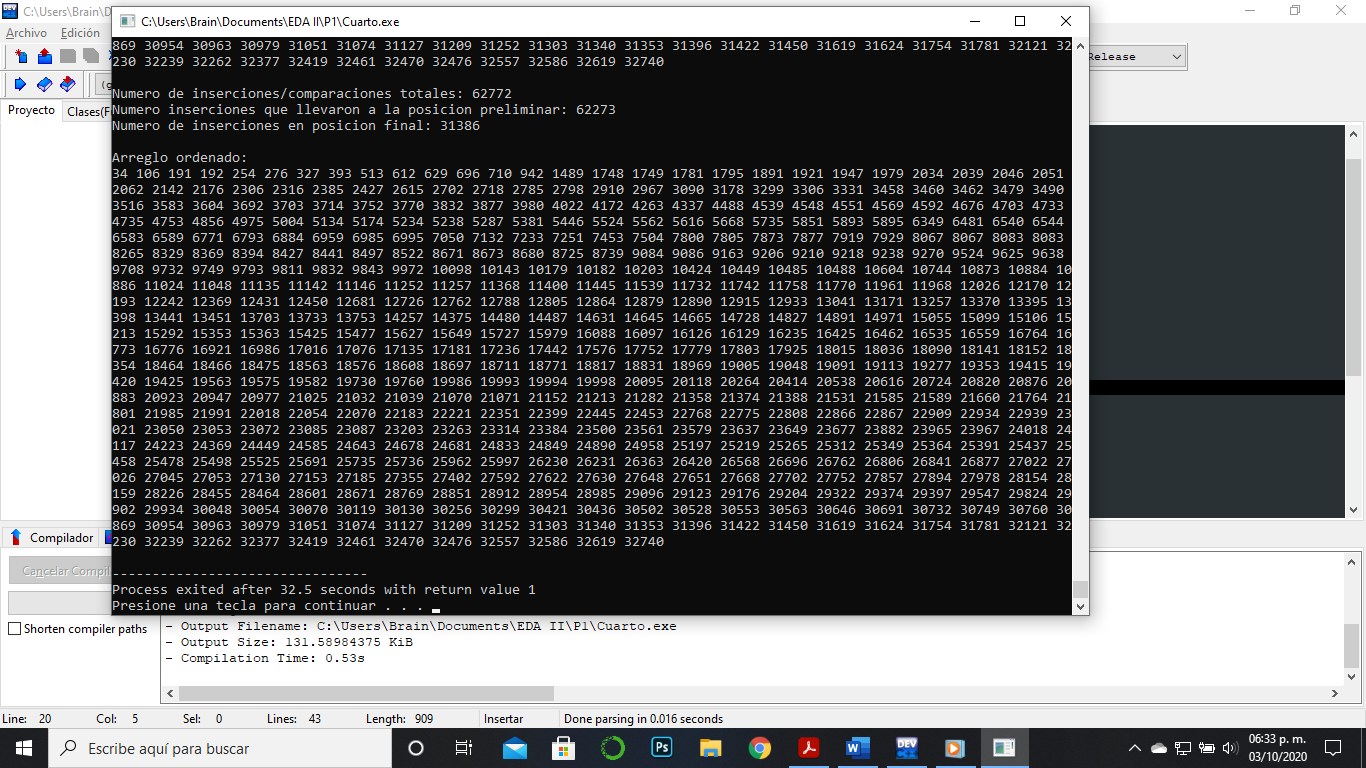
100 -SelectionSort



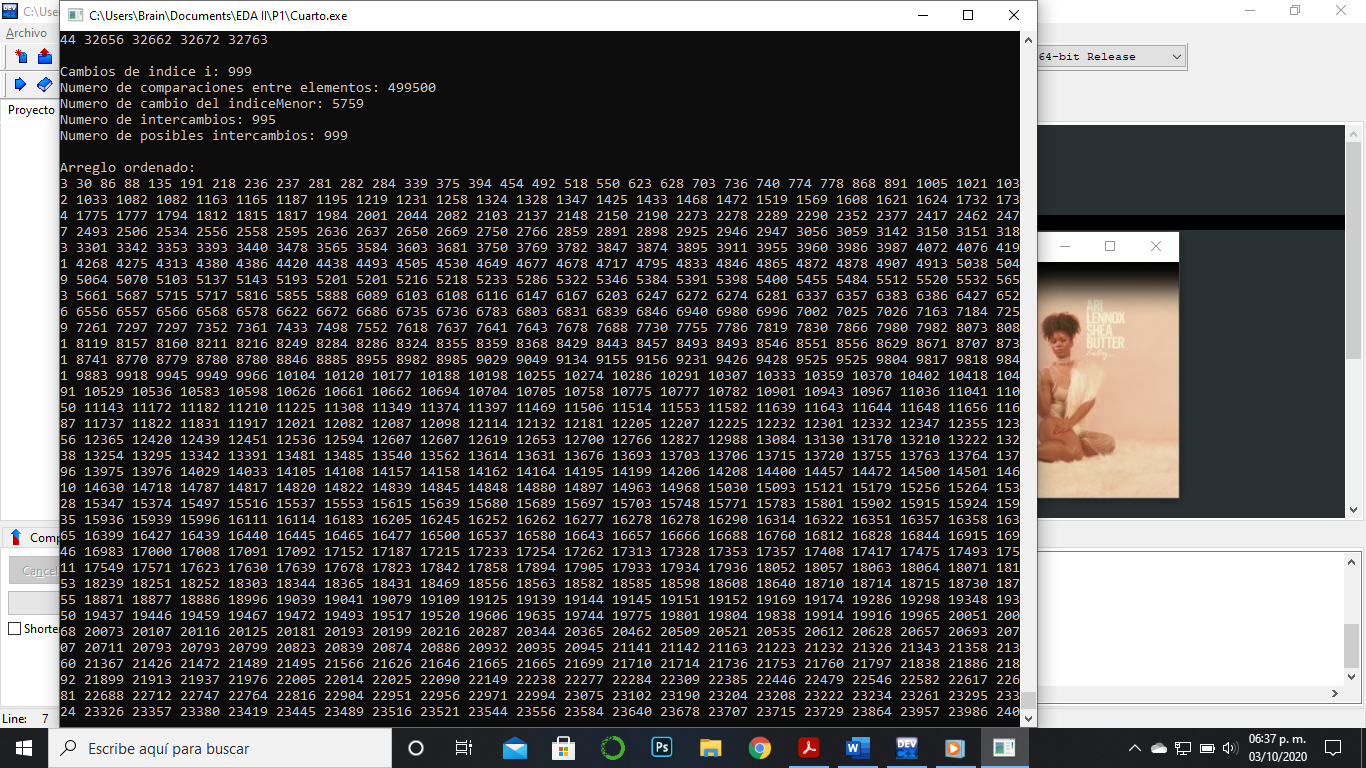
100 - InsertionSort



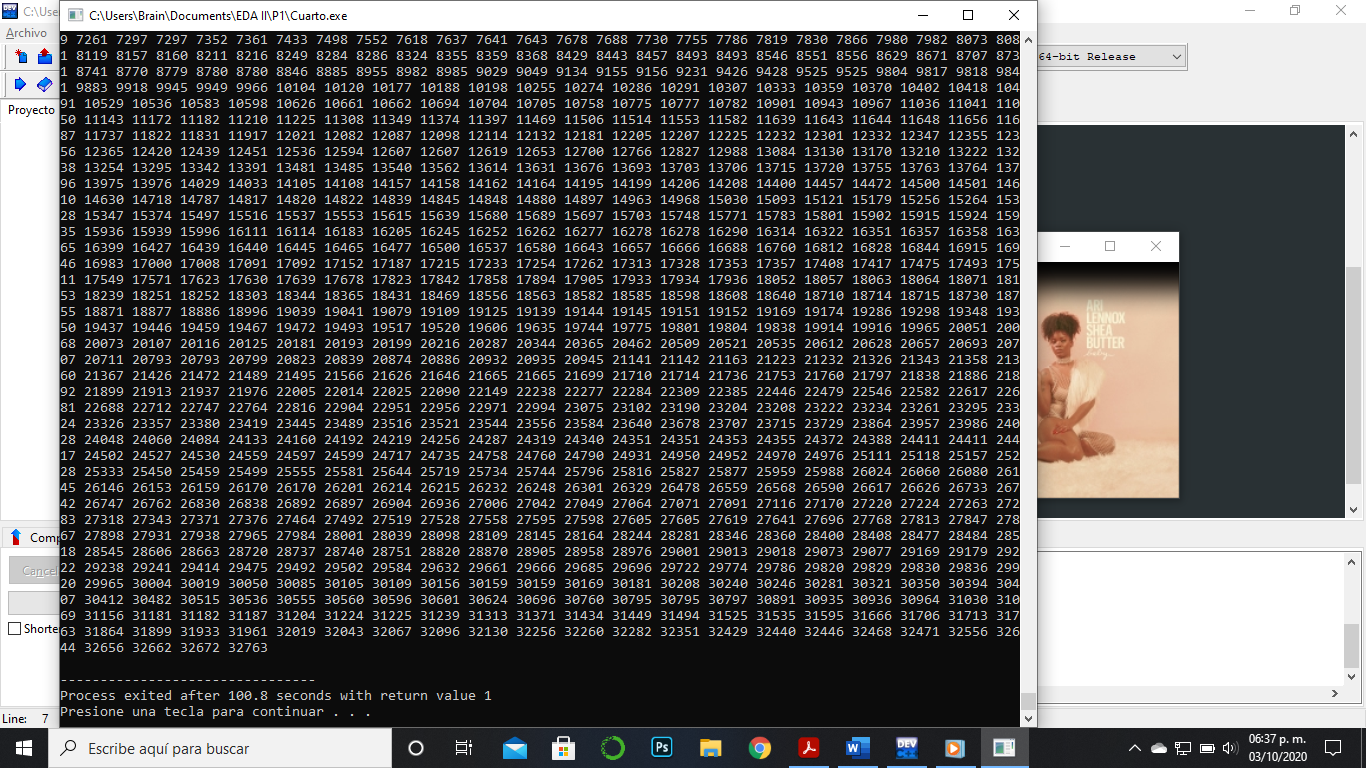
500 - SelectionSort



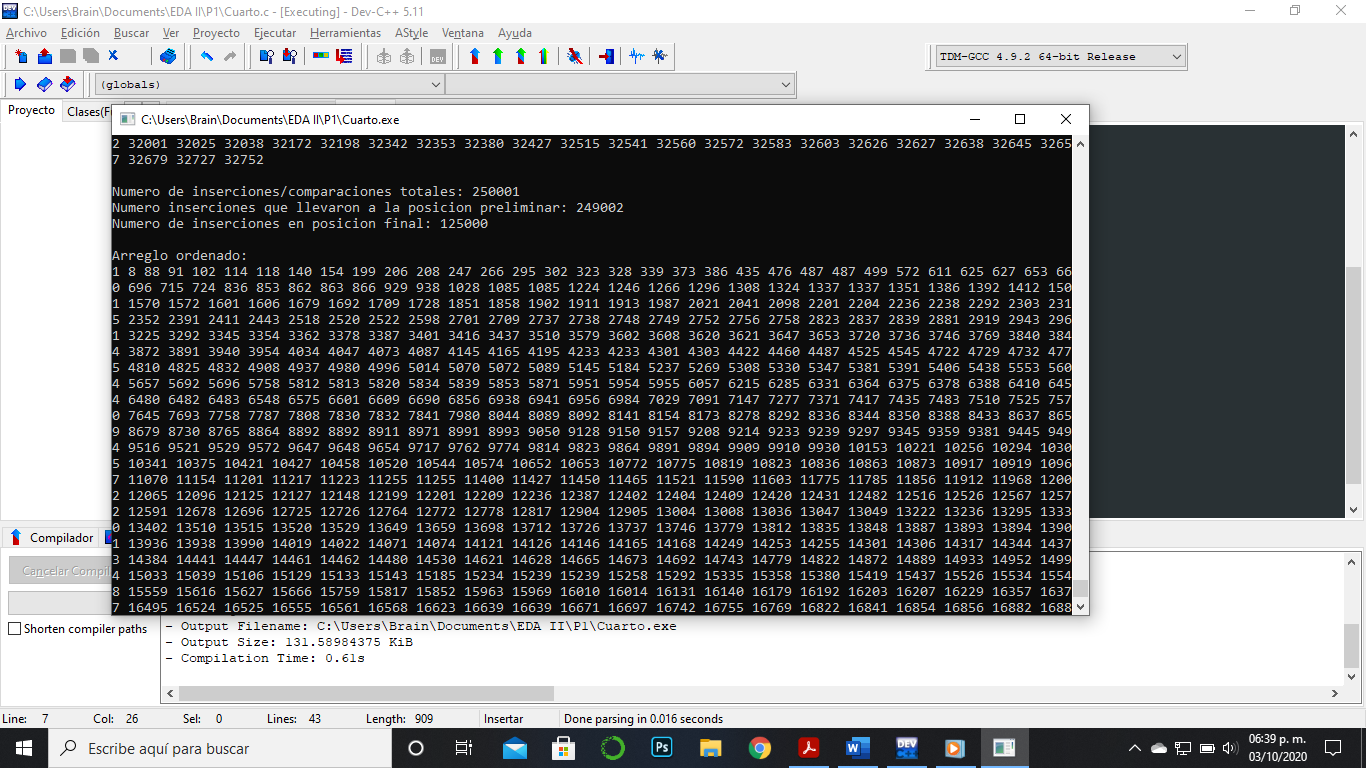
500 – InsertionSort



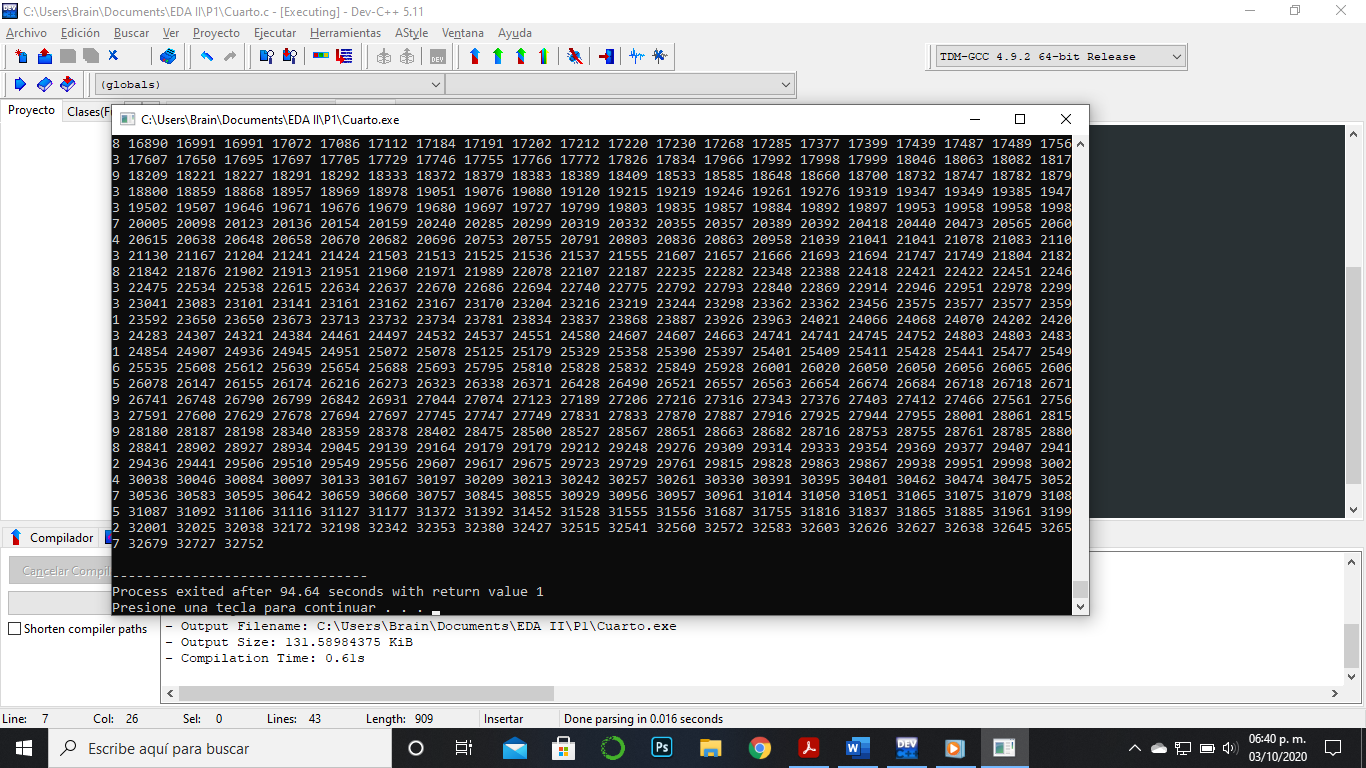
1000 – SelectionSort



1000 - SelectionSort



1000 – InsertionSort



1000 - InsertionSort

**Conclusiones.**

Todo lo bueno tiene un final por lo que este reporte igual, los objetivos se cumplieron, es decir, logre identificar las características de los ordenamientos de Selección y de Inserción, por medio de la realización de 4 ejercicios que consistían en análisis, codificación, y desarrollo de los algoritmos de ordenamiento, reforcé lo aprendido en las clases, además recordé bastantes cosas de la codificación.

Por último, comprendí la importancia de las implementaciones o pruebas de estrés de los algoritmos, para el completo entendimiento de las cosas, ya que la noción de la complejidad en los algoritmos era algo teórico y se volvió más práctico, eso me ayudo bastante a comprender mejor el funcionamiento de los algoritmos.