|  |  |  |
| --- | --- | --- |
|  | **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):* | 2 |
| *Integrante(s):* | Díaz Hernández Marcos Bryan |
| *No. de Equipo de cómputo empleado:* | Equipo personal |
| *No. de Lista o Brigada:* | 9 |
| *Semestre:* | 2021-1 |
| *Fecha de entrega:* | 11 de octubre de 2020 |
| *Observaciones:* |  |
|  |  |

CALIFICACIÓN: \_\_\_\_\_\_\_\_\_\_

**Objetivo de la practica**

El estudiante identificará la estructura de los algoritmos de ordenamiento BubbleSort, QuickSort y HeapSort.

**Introducción**

En el reporte siguiente se podrá encontrar una explicación concreta de las dificultades, del aprendizaje adquirido del breve análisis realizado a los algoritmos y de la obtención o cumplimiento del objetivo planteado al comienzo de este. Todo realizado con el fin de poder comprender a futuro como realice esta práctica y para tener mas claro cada concepto.

**Ejercicios de la practica:**

* Ejercicio 1:

El primero consistía en colocar los algoritmos de ordenamiento: BubbleSort, QuickSort, y HeapSort, dentro de la biblioteca de ordenamientos y una vez dentro se tenía que evidenciar si el QuickSort fue visto en clase, por lo que se hizo un análisis del algoritmo. El segundo punto para hacer era el detener el BubbleSort porque este se itera n veces sin importar si ya se ordenó la lista, por lo que es necesario detenerlo. Y por último se tenía que analizar la función Heapify que tiene el HeapSort y describir su funcionamiento.

* Dificultades en el código

El primer problema que tuve fue el QuickSort debido a que al ejecutarlo, eliminaba valores o agregaba negativos, lo que termino siendo un problema para poder analizar la forma en que este operaba y distinguir si ya lo habíamos visto en clase (Imagen 1).

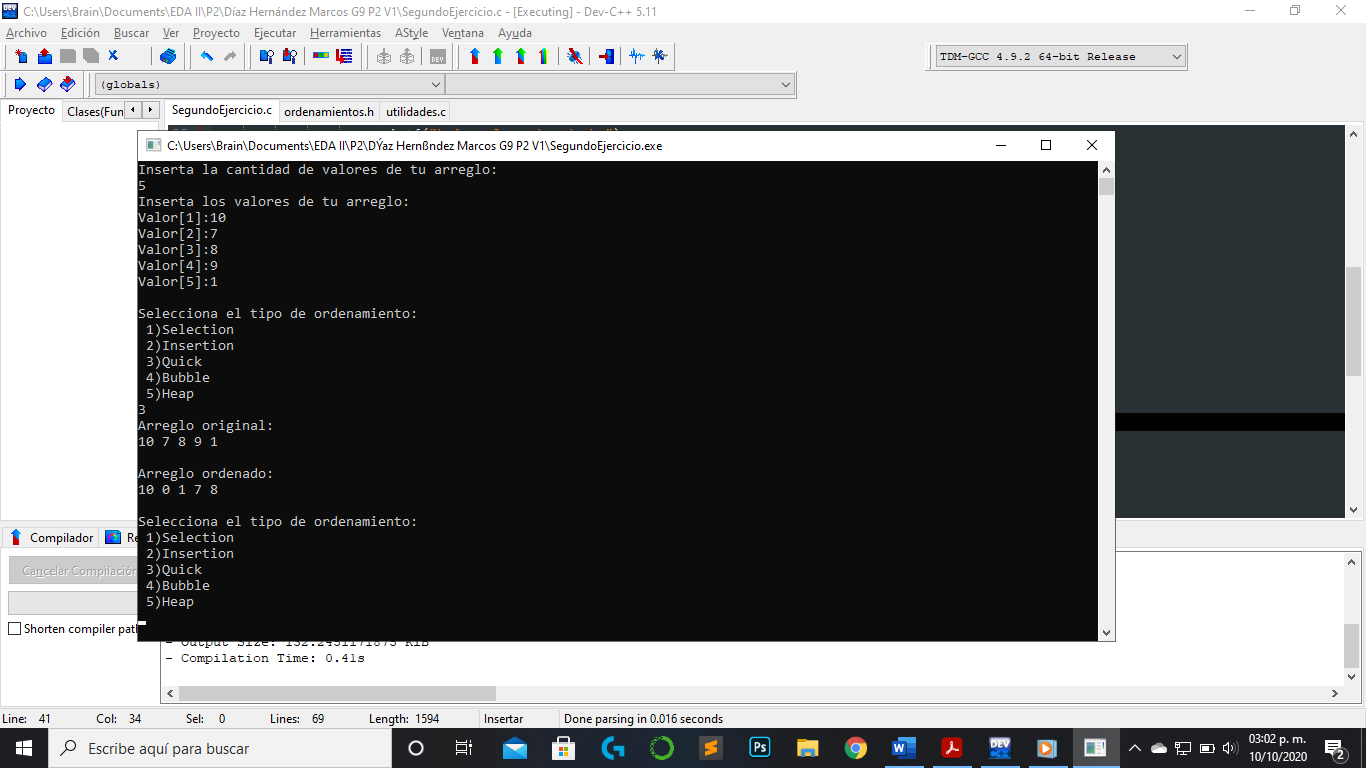
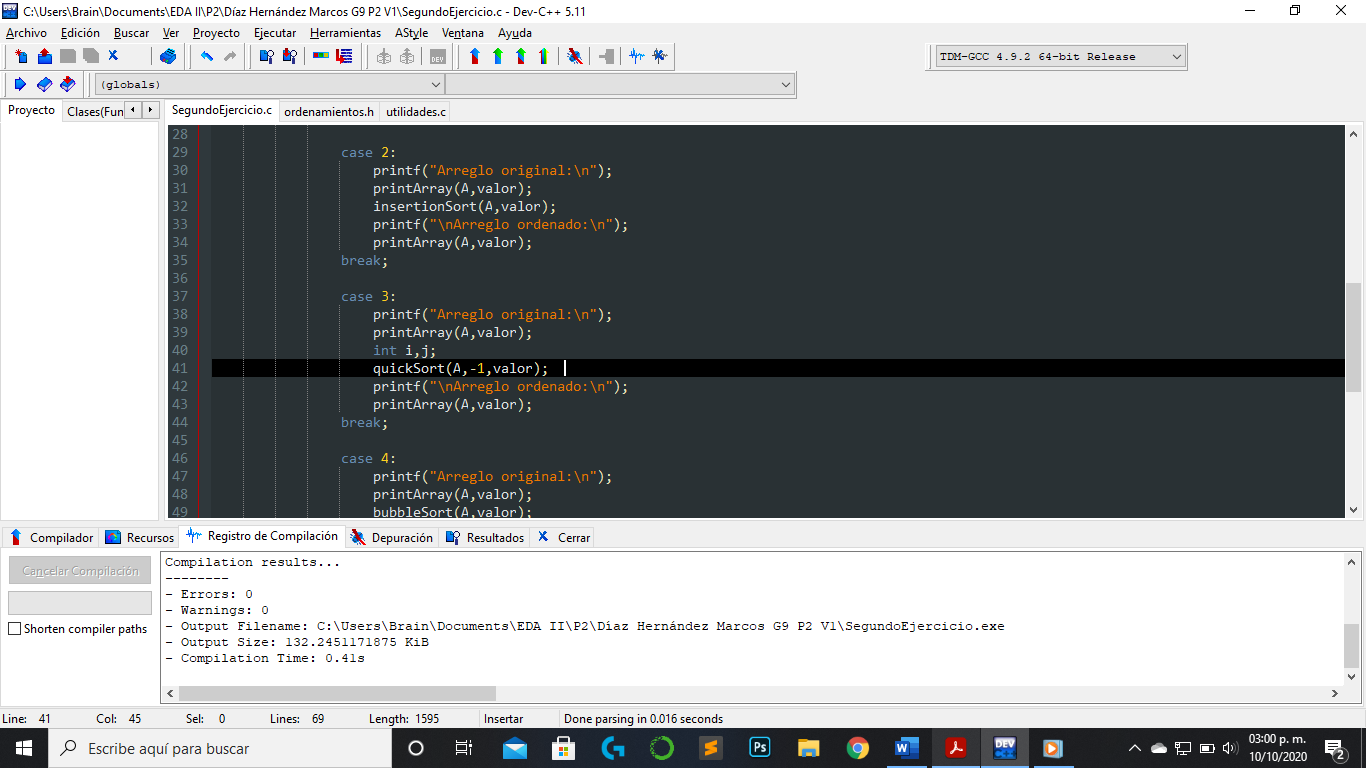
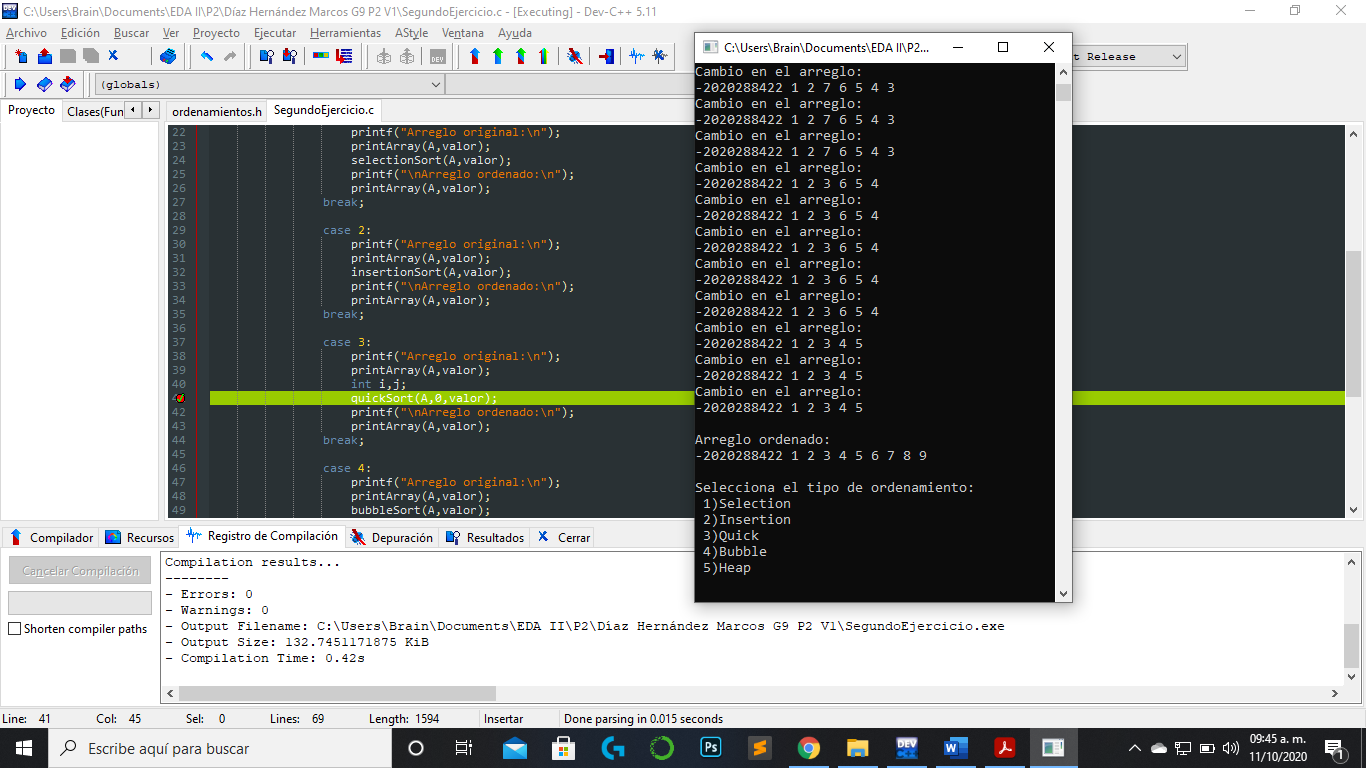


Imagen 1

Solución: Para poder resolverlo, tuve que checar el algoritmo a mano y eso me genero la evidencia incontrovertible de que no habíamos visto esa variable del algoritmo, además que pude observar que hace iteraciones innecesarias o que se vuelve poco eficiente una vez ordenada la lista, en el momento que devuelve el pivote, ya que pide la recursividad. Al final tuve que colocar en el valor de low, un -1 para que este funcionara, debido a que los valores en los for se manejan uno menos si se deja en el valor original.



Evidencia de Quick

El segundo problema que encontré fueron las bibliotecas, ya que al implementarlas y querer llevar a cabo buenas practicas que saltaba el undefined de las funciones que tenían dentro, así que incluso definiendo las bibliotecas esto no funcionaba, puede ser por la versión de mi Dev, pero igual lo solucione por medio de colocar los prototipos y las instrucciones en la misma biblioteca (Imagen 2).

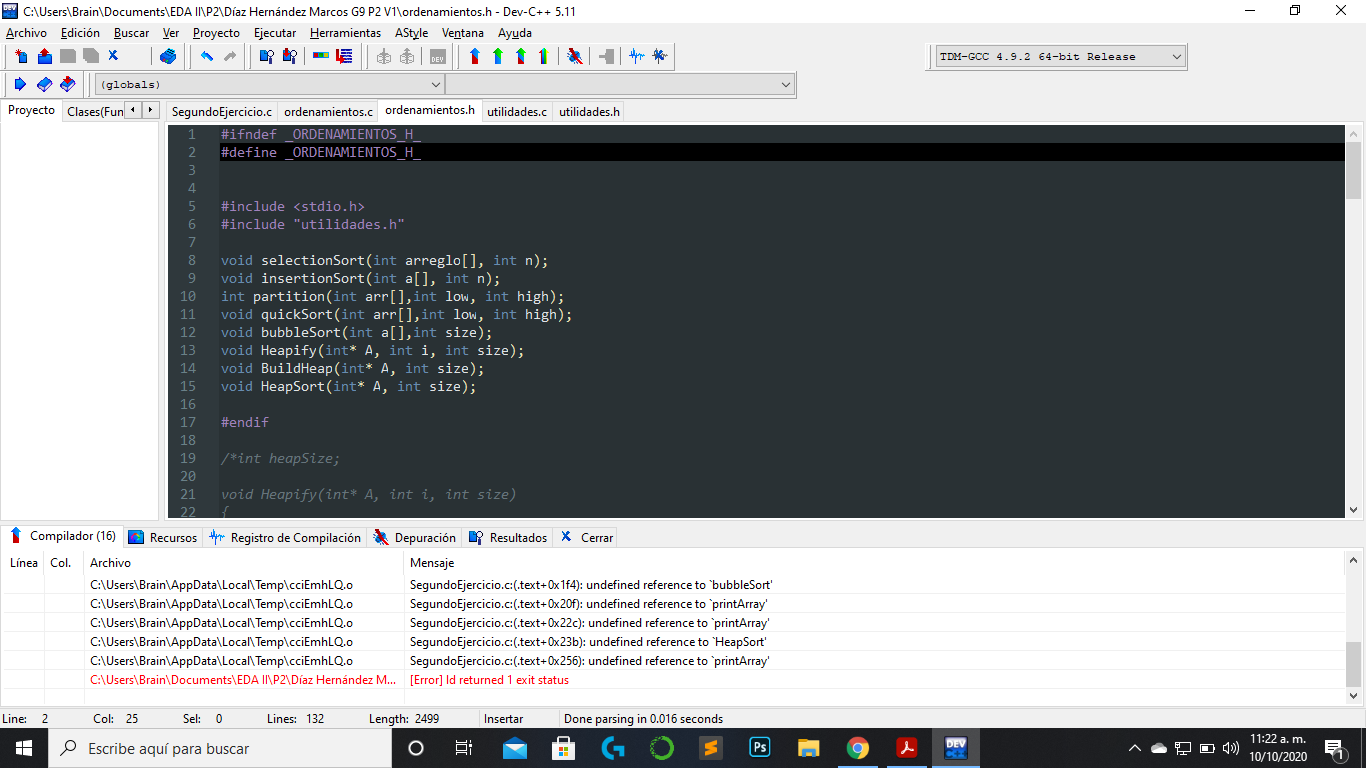


Imagen 2

* Análisis del QuickSort.

El QuickSort o la variable de este, no la habíamos visto en clase, por varias razones, ya que si bien comparte una estructura de recursividad con los demás vistos, eso es porque el algoritmo funciona de esa forma creando sublistas para ordenar y dividir en más.

Mi evidencia no es una ejecución porque al final de cuentas los algoritmos no se pondrían determinar en cuanto a las operaciones que hacen, a menos que se tengan todas las versiones y se ejecuten para el mismo arreglo, pero eso no es posible debido a que no se tiene el algoritmo de uno de los vistos, o por lo menos no se sabe si esta bien hecho, porque no pude comprobarlo, de cualquier forma al revisar apuntes y realizar el algoritmo a mano pude comprobar que no funciona igual que los demás.

* En el primero el ordenamiento se escogía como pivote al primero de la lista y después se comparaba con el ultimo para buscar menores que el pivote y una vez que los encontraba, se ponía a buscar mayores que el y eventualmente tras repetirse, quedaban los menores a la izquierda y los mayores a la derecha, para separar en dos listas y ordenar de nuevo.
* El segundo tomaba como pivote al elemento de en medio y después comparaba desde los extremos para encontrar los mayores y menores del pivote, pero este permanecía en su posición los que se intercambiaban eran los valores que resultaban ser mayores o menores, una vez ordenado se dividía, en otras sublistas y se repetía.
* Este algoritmo toma como pivote al elemento del final, y comienza a comprobar todos los valores que sean menores, en caso de encontrarlos pues solo intercambia entre los valores de i y j, lo que permite encontrar el valor mas grande y colocarlo a la derecha del valor y posteriormente se comienza a dividir en otras listas, y se repite el proceso, por lo que no son iguales las versiones.

Ahora sabiendo como funciona cada uno pues, ya no hay duda de que es una nueva versión y que hace operaciones distintas para poder llevar a cabo el ordenamiento de la lista y que se cumplan las condiciones del algoritmo, como que los menores estén a la izquierda y los mayores a la derecha.

La evidencia igual muestra el problema de colocar el índice 1 o 0, ya que eso genera una reiteración innecesaria, lo que puede estar causando los problemas que se muestran en los problemas que tuve en la codificación (Imagen 1).

Recibo de tienda

Descripción generada automáticamente

Imagen 1

* Detener el BubbleSort cuando la lista esta ordenada.

Para hacer lo anterior únicamente tuve que ver como funciona y haciendo las pruebas a mano, me di cuenta de que pues comprueba todo pero el for de afuera es quien dicta las iteraciones innecesarias en caso de que ya se ordeno la lista y se sigan las iteraciones, por lo que simplemente coloque una comprobación dentro del segundo for, el cual recorre la lista y si se cumple que ya no hay intercambios en toda esas comprobaciones mi valor de comprobación no va a cambiar por lo que la lista estará ordenada y se tiene que romper el primer for para que no haya más comparaciones(Imagen 2).

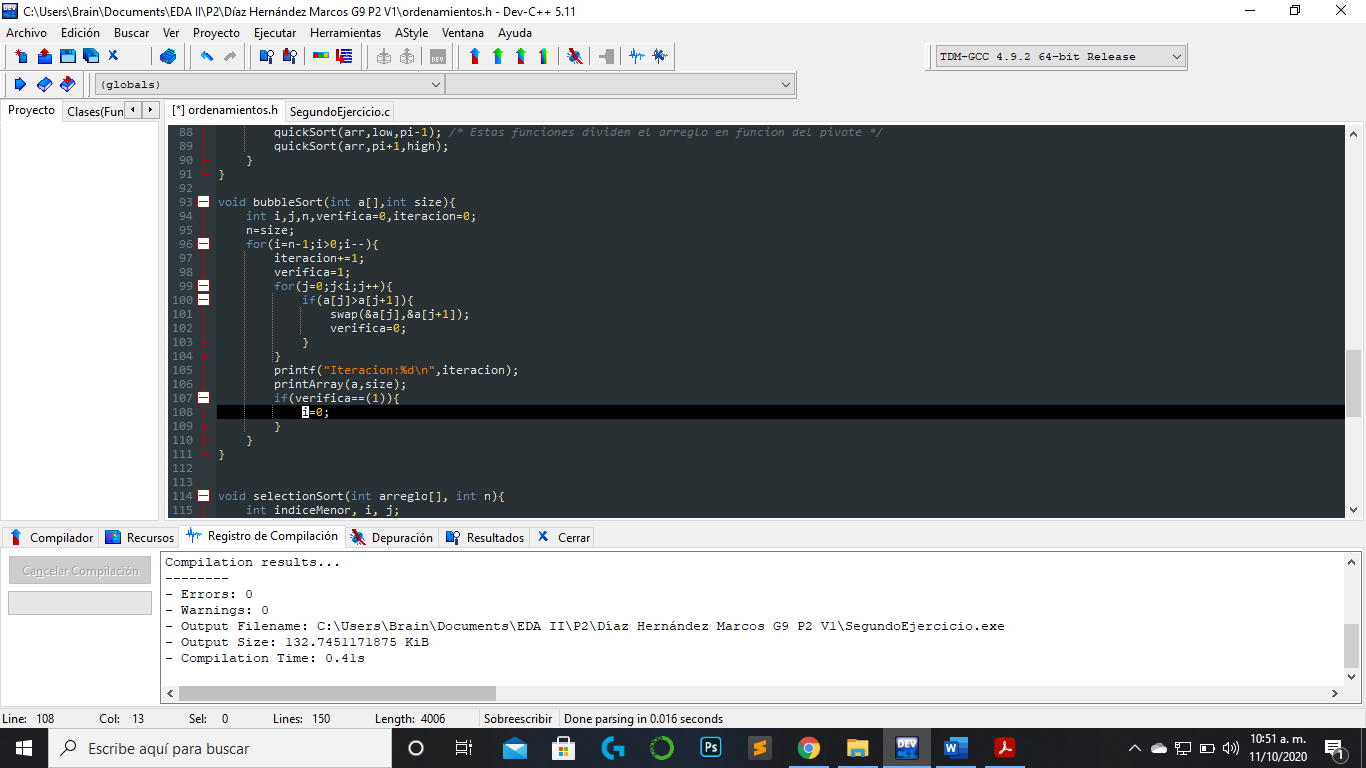


Imagen 2

* Describe la función Heapify.

Esta función lo que hace es utilizar el arreglo que le envían y poder verificar los valores de los padres, por medio de comparaciones simples, lo importante de la función es verificar los argumentos y valores que recibe porque con eso se puede determinar el tipo de organización que lleva a cabo si es Top o Buttom.

Lo que recibe es una lista, un índice, y el tamaño de la lista, por lo que hay que describirlos:

* La lista que recibe es un paso directo de la lista original, por lo que se trabaja sobre una lista ya hecha (Imagen 1).

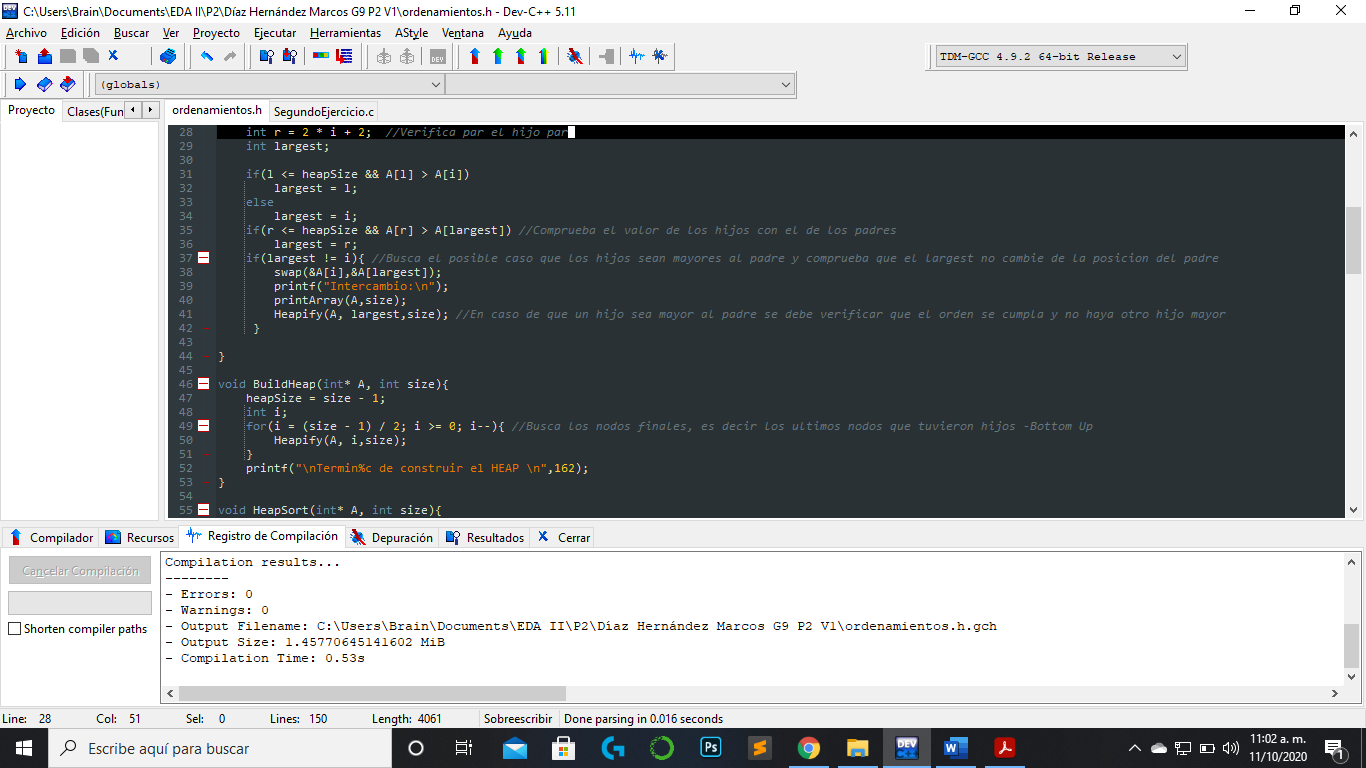


Imagen 1

* El índice busca una posición, es decir que se divide sobre dos y se le resta un uno, esto significa que se buscan el ultimo nodo que tuvo hijos y por medio de este se comienza a verifican los valores de los padres con los hijos (Imagen 2).

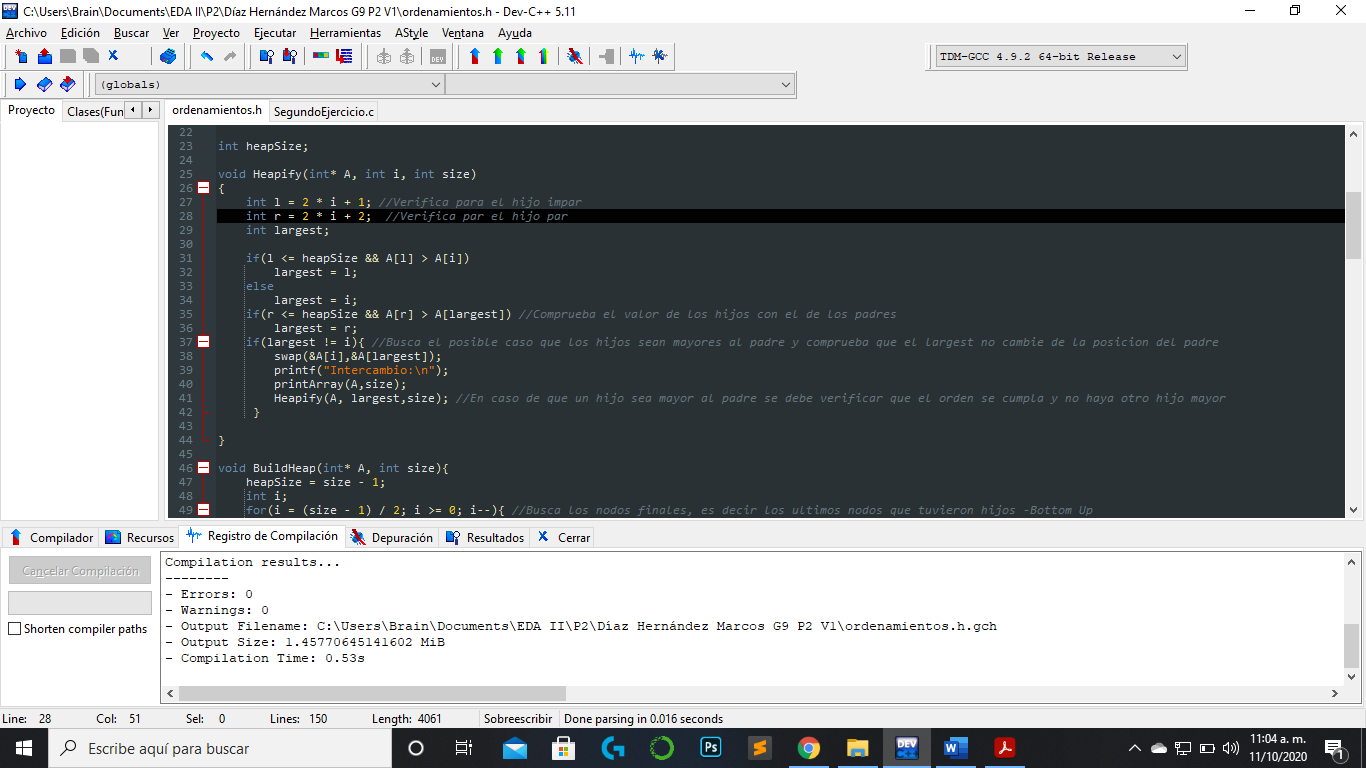
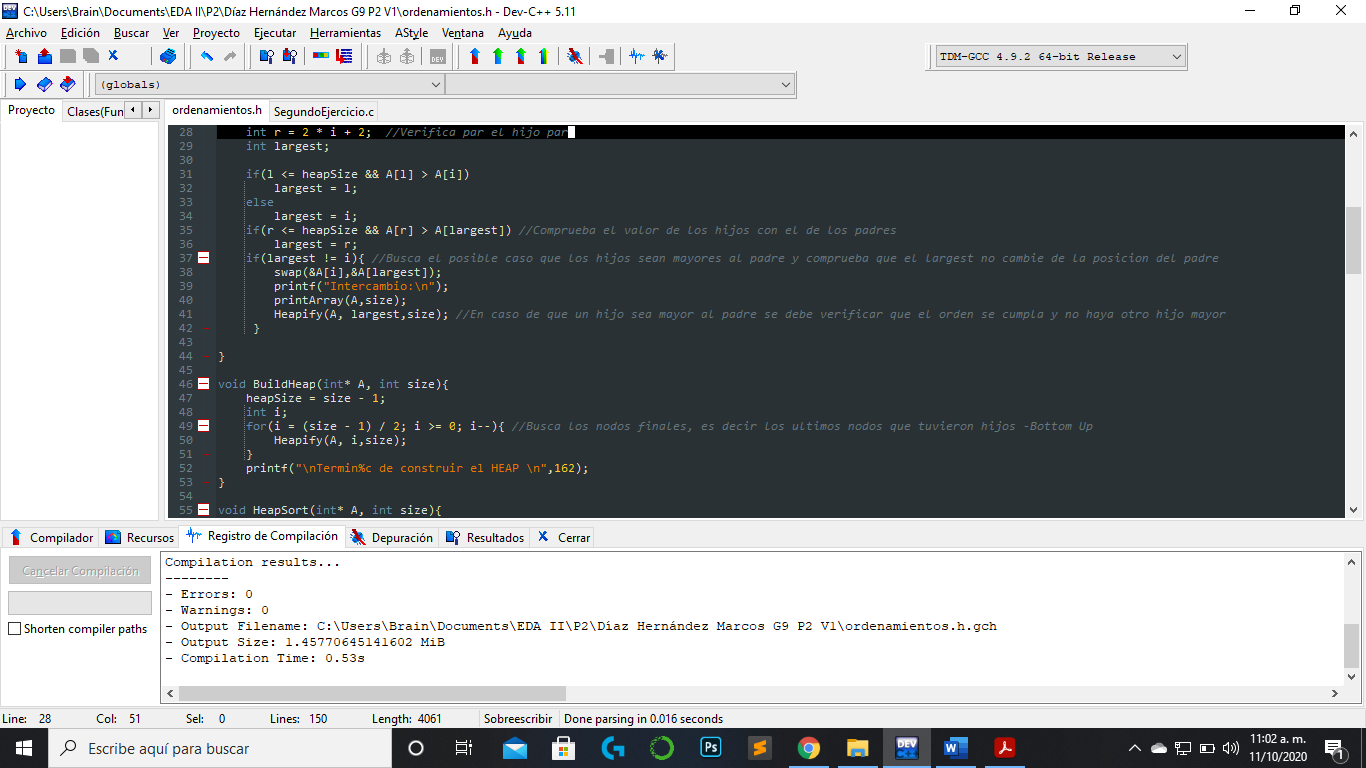


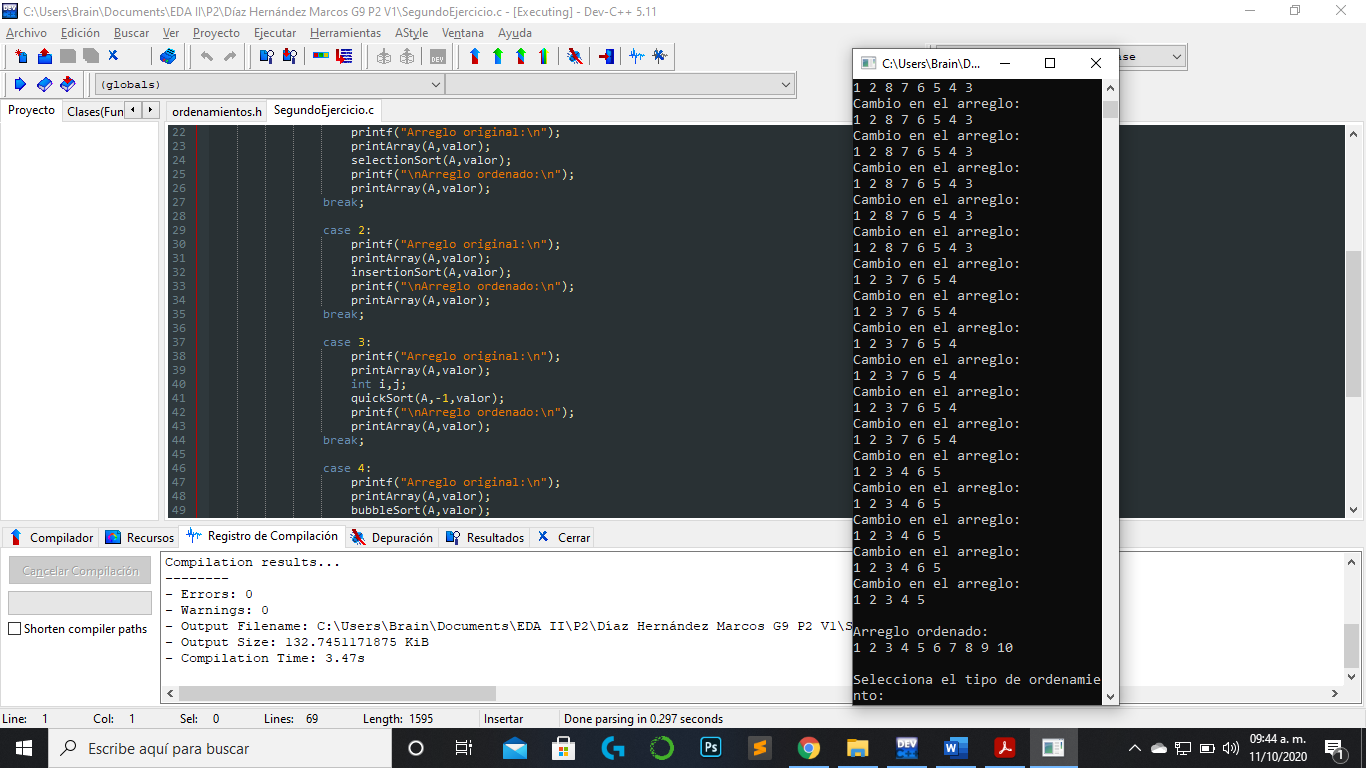
Imagen 2

Una vez revisado esto nos podemos dar cuenta que es Bottom Up, por las características mencionadas, y para concluir se verifica que los valores de cada hijo no sean mayores al del padre, y en caso de que suceda se intercambian posiciones y se vuelve a comprobar que el nuevo padre no sea menor a otro hijo y al final ya se comienza a eliminar las raíces para comenzar a ordenar los valores.

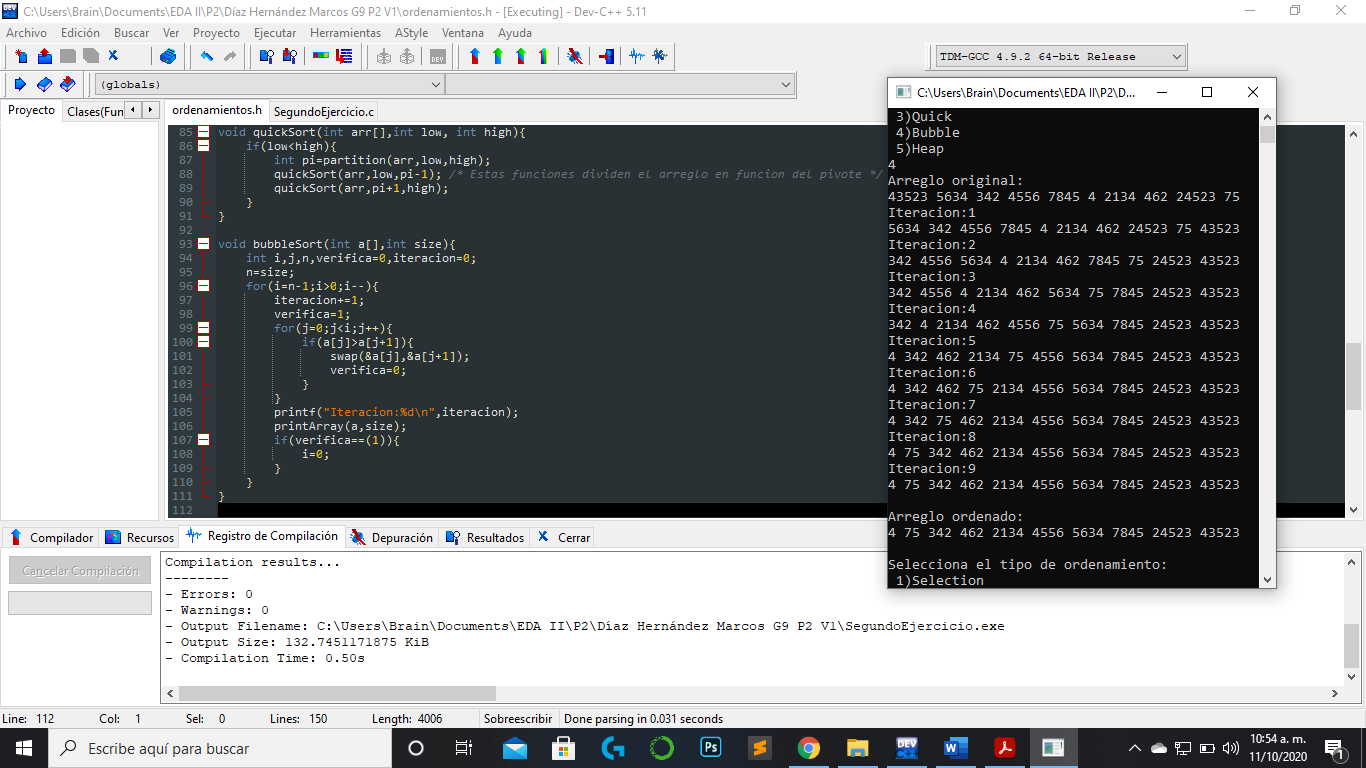
* Relación con la teoría:

En esta ocasión fue bastante interesante el analizar los algoritmos y fue más fácil intentar realizar el algoritmo a mano con una lista simple, de esta forma se puede distinguir el funcionamiento de cada algoritmo y al mismo tiempo saber donde implementar las cosas, du cualquier forma el peso de clase es bastante ya que es necesario el poder identificar las características de cada algoritmo para poder emitir un juicio acertado.

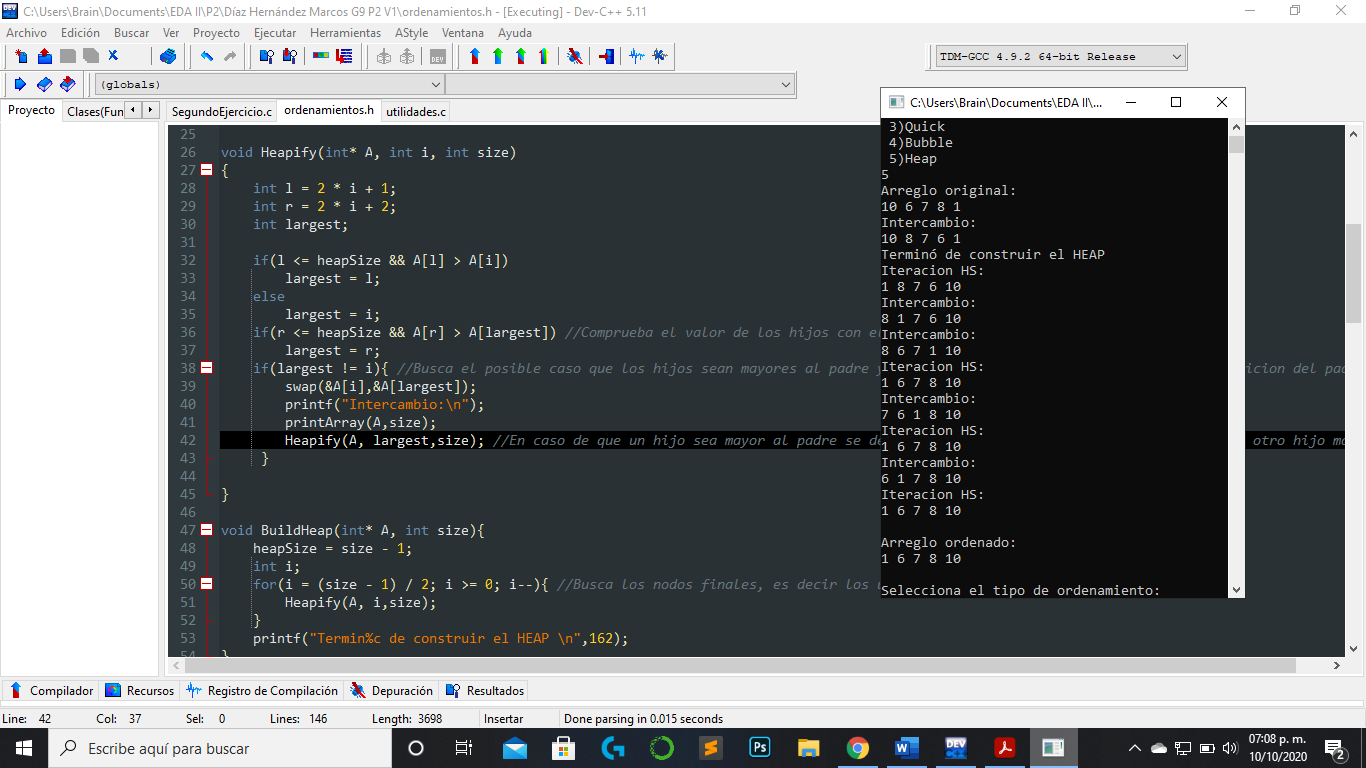
* Evidencia de implementación



Quicksort



BubbleSort



HeapSort

* Ejercicio 2:

El segundo ejercicio consistía en poder implementar cada uno de los algoritmos y el colocar las operaciones correspondientes para verificar los cambios de los valores en el arreglo, después de cada interacción.

* Dificultades en el código

En esta ocasión no tuve dificultades que resolver ya que las había resuelto en el anterior y así que esta esta parte de la practica la puedo llamar felicidad.

* Diagrama de funcionamiento:

Sin embargo es necesario el conocer la forma de funcionamiento de cada algoritmo, por lo que aplicare mis diagramas en forma de explicación de lo que sucede y de igual forma esto pues será la parte principal de este ejercicio que consiste en comprender las operaciones que realizan los algoritmos y como se repiten.

* + En cuanto a los algoritmos se puede decir que el único mas predecible es el BubbleSort porque si el termino menor se encuentra al final este tendrá que llevar a cabo cada iteración y en cada una solo se moverá una posición.

1. QuickSort

QuickSort(A,0,4)

Pivote=Partition(A,0,4)

1)

Arreglo: A

If(0>4)

Partition(A,low,high)

Partition(A,0,4)

Partition(A,0,3)

Partition(A,0,4)

|  |
| --- |
| 5 |
| 2 |
| 3 |
| 1 |

For(j=1;j<=2:i++)

For(j=0;j<=2:i++)

If(2<=1) No

If(5<=1) No

Pivote=1

i=-1

2)

For(j=2;j<=2:i++)

Busca el termino menor, y eventualmente al hacer swap, intercambiara de posición con el mayor al pivote.

If(3<=1) No

QuickSort(A,0,-1) No aplica

QuickSort(A,1,4) Cierra las opciones de dónde escoger

Pivote=0

Swap(5,1)

Return 0

Arreglo: A

Este algoritmo se parece al segundo de clase pero solo que escoge los pivotes como los últimos miembros y comienza a organizar en base a esa premisa.

En este caso solo se hizo un intercambio y al volver a llamar la función se verifica la primera condición.

|  |
| --- |
| 1 |
| 2 |
| 3 |
| 5 |

Algo importante que se tiene que considerar con el QuickSort, son los índices ya que si no se conocen, no es posible implementar el algoritmo, y en ese caso el algoritmo no es conveniente para su uso, de otro lado es bastante eficiente.

1. BubbleSort

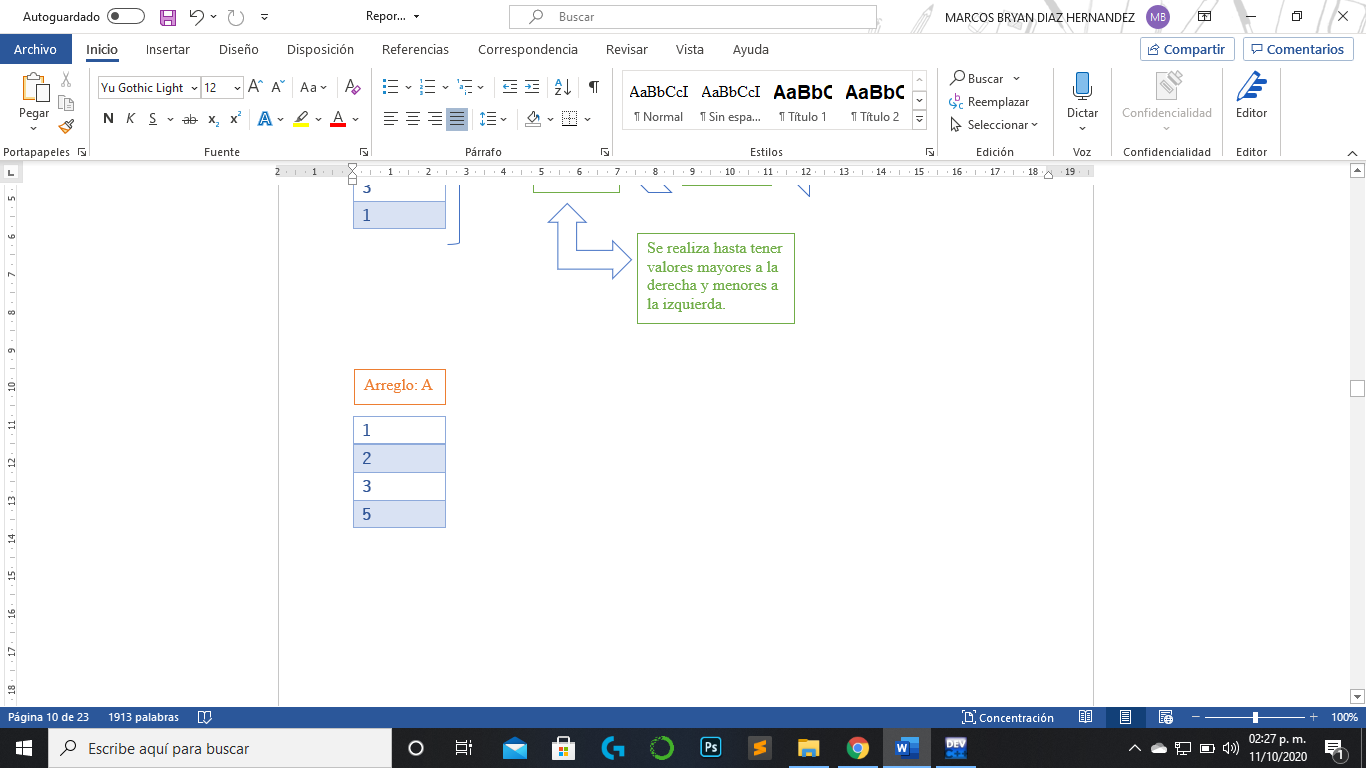
For(i=3;i>0;i++)

N=4

BubbleSort(A,4)

1)

Arreglo: A



swap(5,2)

If(5>2) Si

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

|  |
| --- |
| 5 |
| 2 |
| 3 |
| 1 |

Se realiza hasta tener valores mayores a la derecha y menores a la izquierda.

Como es posible ver el arreglo quedo listo en las iteraciones del segundo for, pero todavía faltan que se realicen las demás iteraciones hasta que acabe el for principal

En el caso de que se ordene al primer ciclo del segundo

For, se tiene que esperar a que el algoritmo termines todos

los for, de modo que es bastante innecesario el tener

que esperar cuando ya está listo el arreglo.

1. HeapSort

BuildHeap(A,3)

BubbleSort(A,3)

Arreglo: A

For(i=1;i>=0; i--)

HeapSize=2

2)

BuildHeap(A,3)

1)

|  |
| --- |
| 5 |
| 2 |
| 3 |
| 1 |

Heapify(A,1,3)

If(3<=2 && 1>2) No

Largest=1

If(4<=2 && algo>2) No

If(Largest!=i) No

L=3

R=4

Int Largest

Heapify(A,1,3)

3)

If(1<=2 && 2>5) No

Largest=0

If(2<=2 && 3>5) No

If(Largest!=i) No

L=1

R=2

Int Largest

Heapify(A,0,3)

Heapify(A,0,3)

Swap(5,3)

HeapSize=1

For(i=2;i>0; i--)

1)

Heap listo

2)

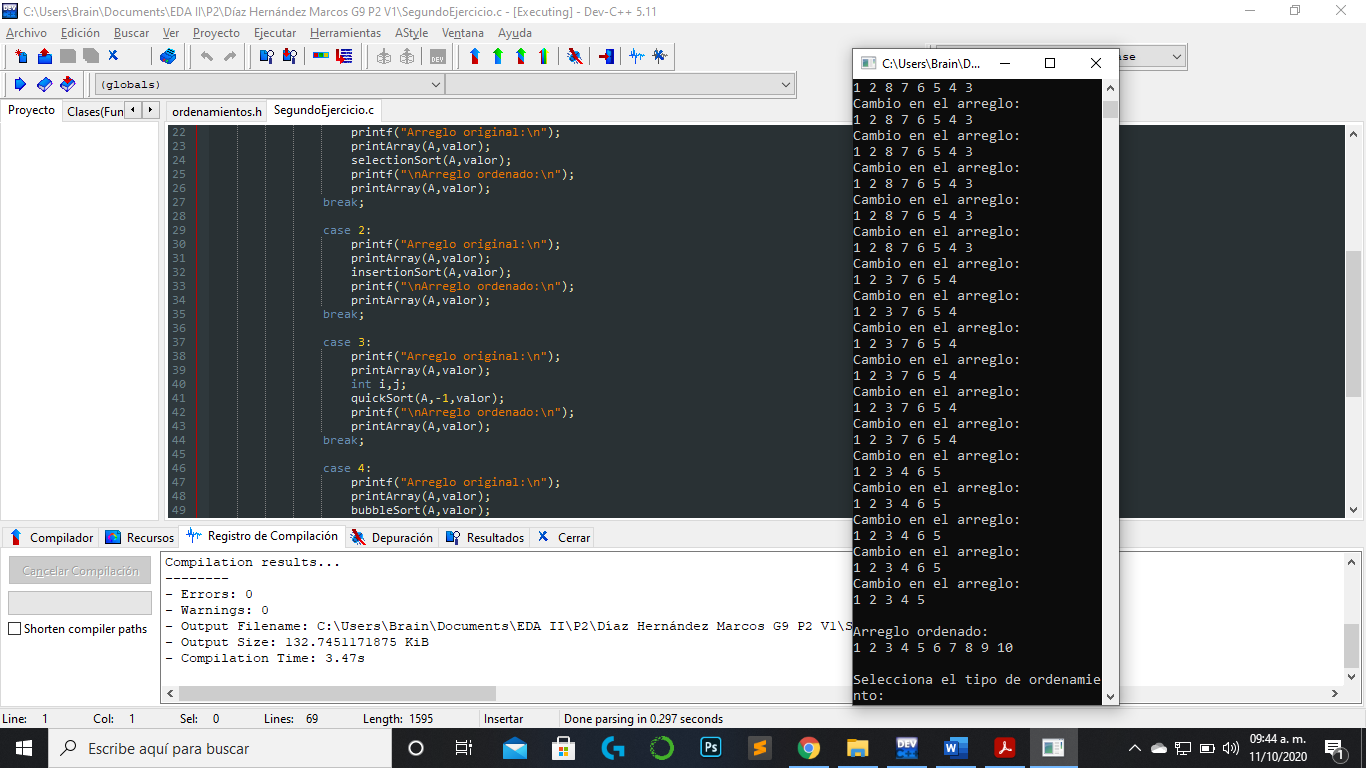
A groso modo se puede ver el modo de funcionamiento de los algoritmos, por lo que son bastantes recursivos por la verificación de las condiciones

* Relación con la teoría:

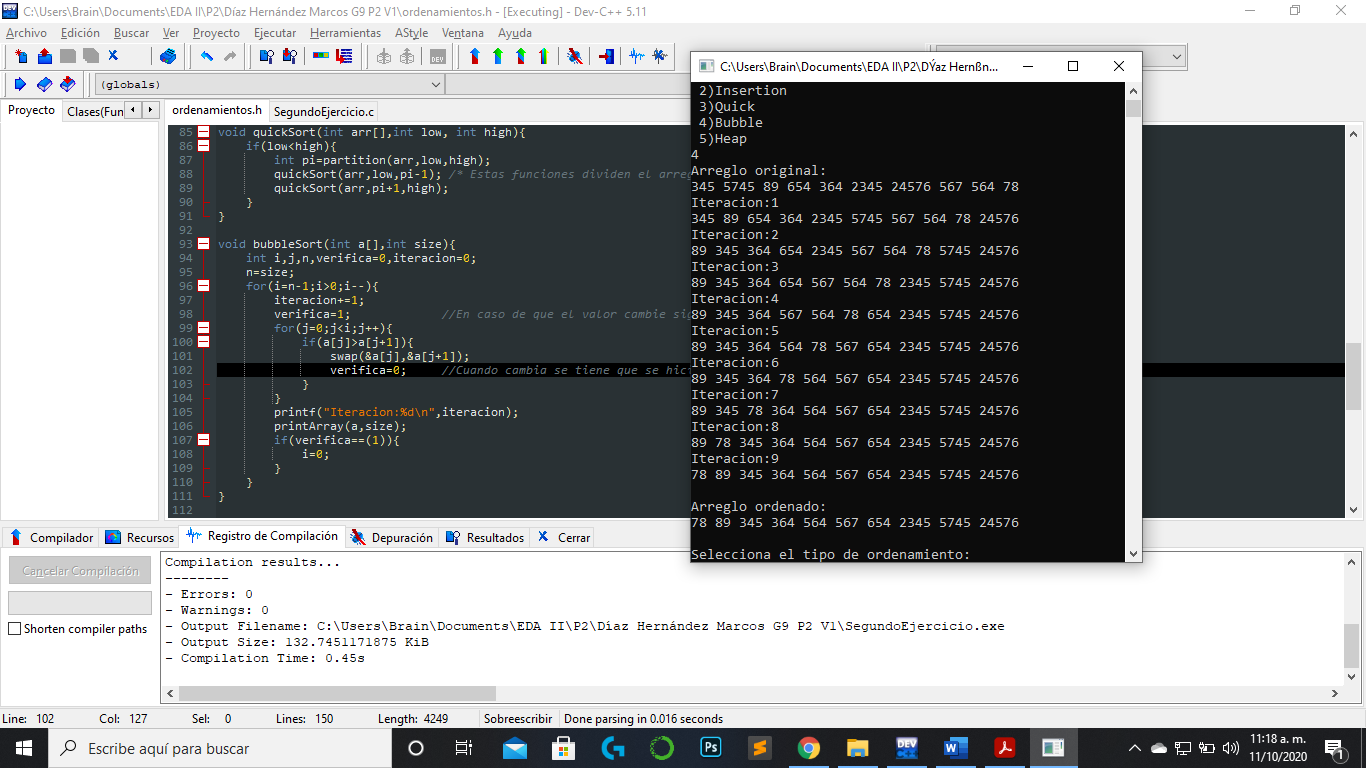
Estos algoritmos si bien los vimos en clase, la verdad es que en código la implementación siempre tiene que cumplir con las características de cada algoritmo y eso se cumple en esta ocasión donde el Quicksort, cumple con las características básicas pero la implementación es diferente.

En el caso de HeapSort, se puede verificar como se implementa el Heap y como se hace la verificación de los valores, cuando se hacen intercambios. Y bueno el BubbleSort es Bubble pero igual se puede ver cómo funciona claramente al ver los cambios en cada iteración.

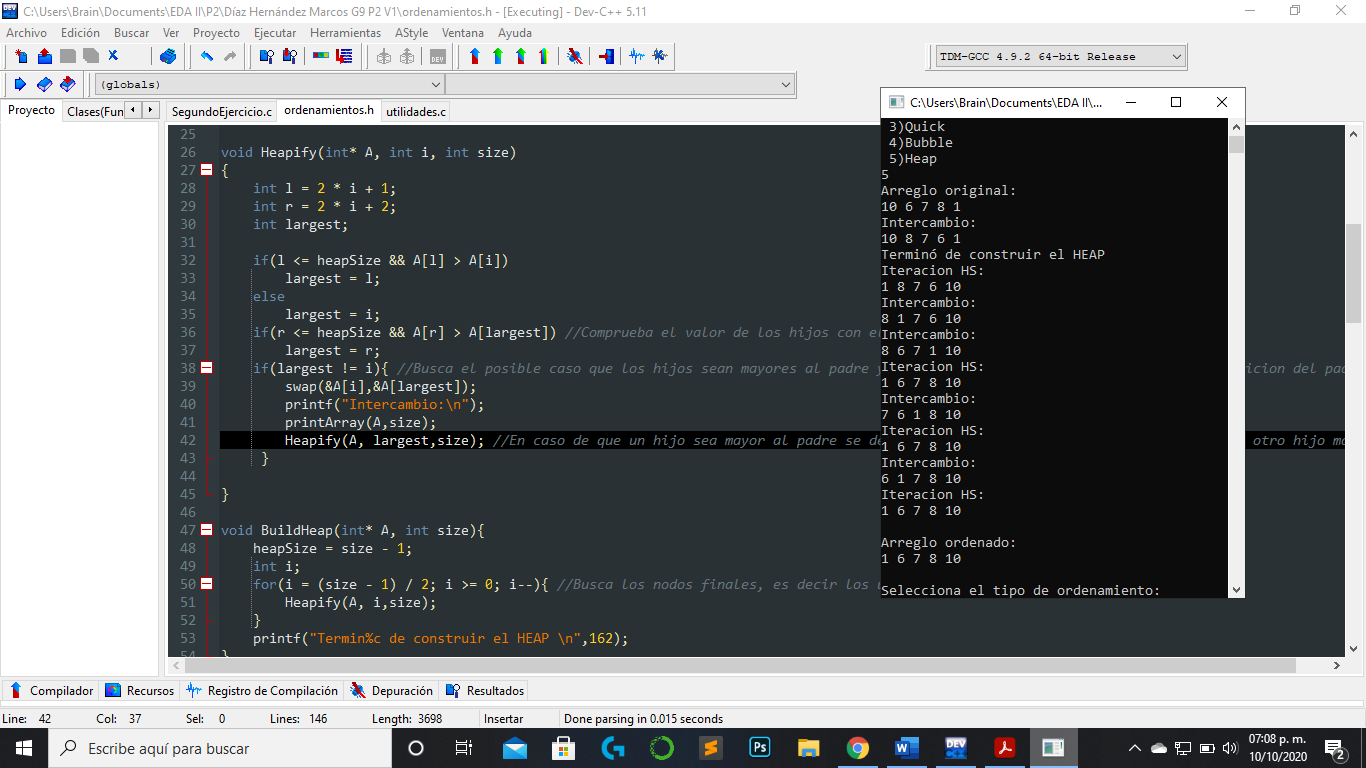
* Evidencia de implementación



Quicksort



BubbleSort



HeapSort

* Ejercicio 3:

Este ejercicio busca el análisis de la complejidad de cada algoritmo, por lo que es necesario el evaluar y realizar pruebas de estrés en cada algoritmo, y a partir de graficas el poder evaluar el tipo de complejidad que tienen.

* Análisis de complejidad.
  + QuickSort

|  |  |
| --- | --- |
| Entrada | Tiempo |
| 50 | 6.214 |
| 100 | 4.231 |
| 500 | 7.617 |
| 1000 | 4.96 |

|  |  |
| --- | --- |
| Entrada | Operaciones |
| 50 | 1030 |
| 100 | 1865 |
| 500 | 13362 |
| 1000 | 30751 |

En el caso de Quicksort para la complejidad se puede ver que el tiempo de ejecución no fue proporcional a la cantidad de la entrada, esto significa que intervinieron varios factores, como las ventanas abiertas, el sistema operativo y el procesador, sin embargo se tiene una subida en base a las tres primeras entradas, lo que podría afectar la grafica en general que tiene un tendencia lineal que no corresponde con la gráfica, y los primeros tres puntos nos muestran que es una gráfica de nlog(n), es decir que tiene la tendencia de ser de complejidad O(nlogn), checar comparación de la graficas.

Como segunda muestra se utiliza la grafica de entrada y las operaciones correspondientes, esta segunda grafica muestra la tendencia que existe entre el tiempo y las operaciones, con esta segunda grafica podemos concretar la hipótesis del párrafo anterior.

* + BubbleSort

|  |  |
| --- | --- |
| Entrada | Tiempo |
| 50 | 3.385 |
| 100 | 3.466 |
| 500 | 3.732 |
| 1000 | 3.49 |

|  |  |
| --- | --- |
| Entrada | Operaciones |
| 50 | 3129 |
| 100 | 12104 |
| 500 | 314344 |
| 1000 | 1252261 |

En este algoritmo vuelve a suceder algo similar con el tiempo de ejecución y el tamaño de la entrada lo que significa que de nuevo el existen factores que pueden modificar la velocidad del procesamiento de ciertas actividades y de los algoritmos a la par, entonces analizando la grafica podemos ver que existe una tendencia de crecimiento en el tiempo, pero de pronto baja.

Para comprobar esto tenemos la segunda grafica que claramente nos muestra la forma de esta, siendo cuadrática, lo que significa que los tres primeros tiempos de la primer grafica tenían una tendencia de crecimiento y que a cada entrada las operaciones que se realizan son cada vez mayores al grado de ser el cuadrado del valor.

Por lo tanto la complejidad es de 0(n2).

* + HeapSort

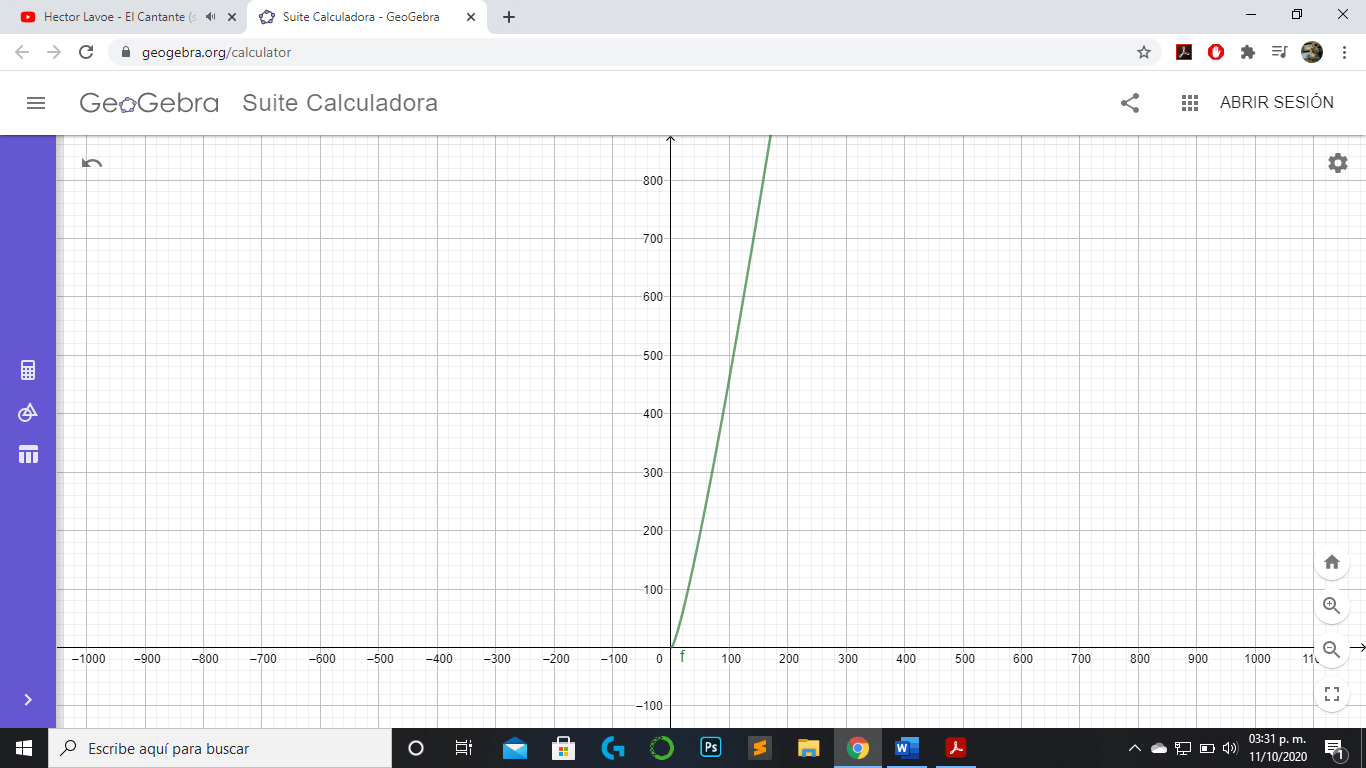
|  |  |
| --- | --- |
| Entrada | Tiempo |
| 50 | 3.575 |
| 100 | 9.698 |
| 500 | 196 |
| 1000 | 675.1 |

|  |  |
| --- | --- |
| Entrada | Operaciones |
| 50 | 1101 |
| 100 | 2663 |
| 500 | 17743 |
| 1000 | 39323 |

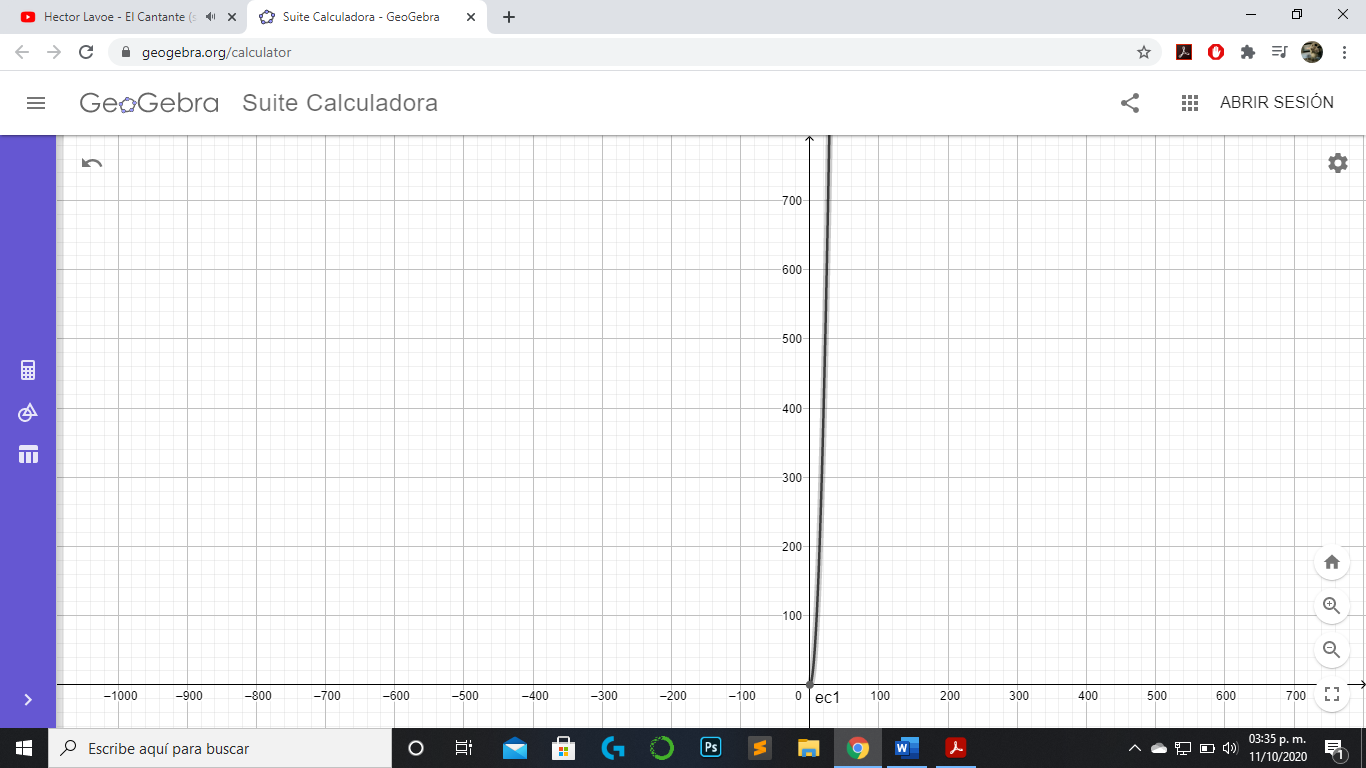
El caso de HeapSort me pareció bastante curioso ya que el tiempo que toma para realizar las operaciones en base a la entrada es mucho, y eso toma sentido al ver el código y al analizarlo, ya que tiene que evaluar valores e intercambiarlos, eliminar, por lo que se vuelve bastante amplia la operación, sin embargo están no crecen demasiado, se mantienen pocas si las comparamos con las del Bubble, esto da como resultado que al igual que el QuickSort, se tenga un crecimiento en tiempo y en operaciones realizadas de acuerdo a la entrada, pero el crecimiento de las operaciones no es mucho es proporcional y similar al del Quick.

Incluso la primer grafica muestra la tendencia semi-cuadratica pero en la segunda se ve la linealidad que compone al algoritmo, por lo que se tiene una complejidad de 0(nlog(n)).

* Comparación de graficas.



Al ver las graficas podemos ver que existe la tendencia de cumplir con la gráfica nlogn, igual es necesario el tener un comparado de graficas y la semejanza pero de este modo podemos darnos cuenta de la forma inicial de la grafica y como coinciden con las de tiempo y entrada.

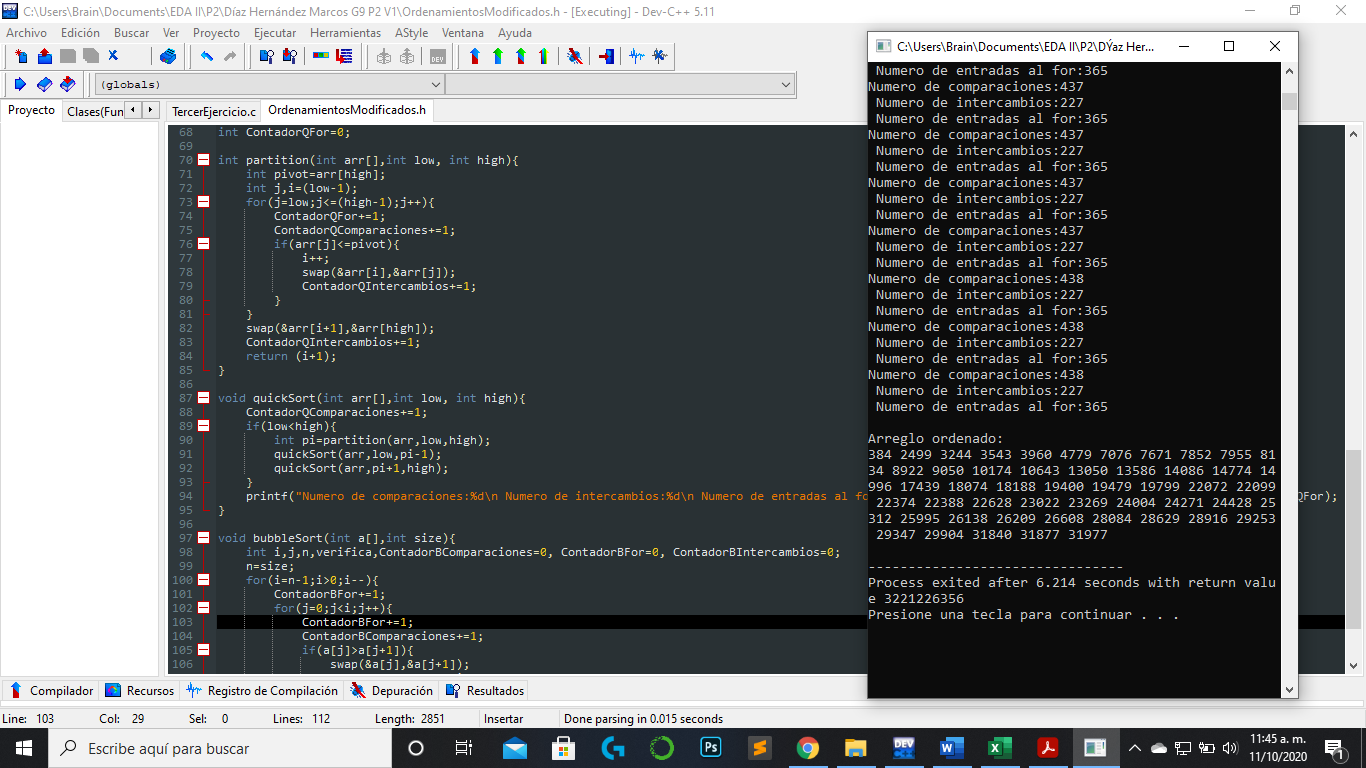


De este sencillo modo es posible comparar las graficar de complejidad y ver que gran parte de dictamen es dado por la apertura de nuestras graficas porque la n2, esta mas pegada al eje y, y la grafica de nlogn, está en un punto medio entre la y=x, y entre la n2, me parece un modo bastante bueno y sencillo.

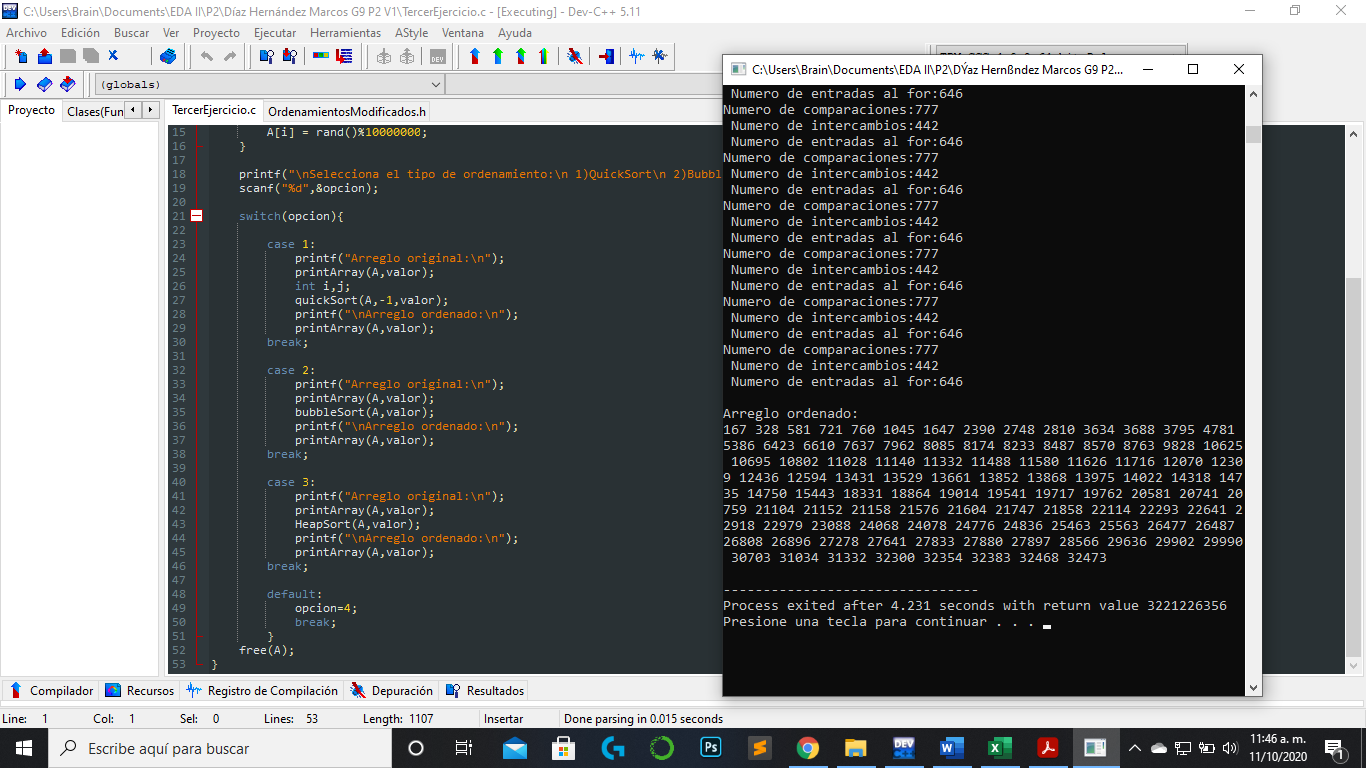
* Relación con la teoría:

Este ejercicio esta bastante cargado de análisis, y mas que nada de buen ojo, porque si bien es posible tener factores que determinen la complejidad dentro de la formalidad, la mayoría de estos métodos son matemáticas específicas para la solución de los problemas, tanto inducción como identidades que en este caso no se utilizaron pero no obstante, es posible llegar a una decisión de la complejidad.

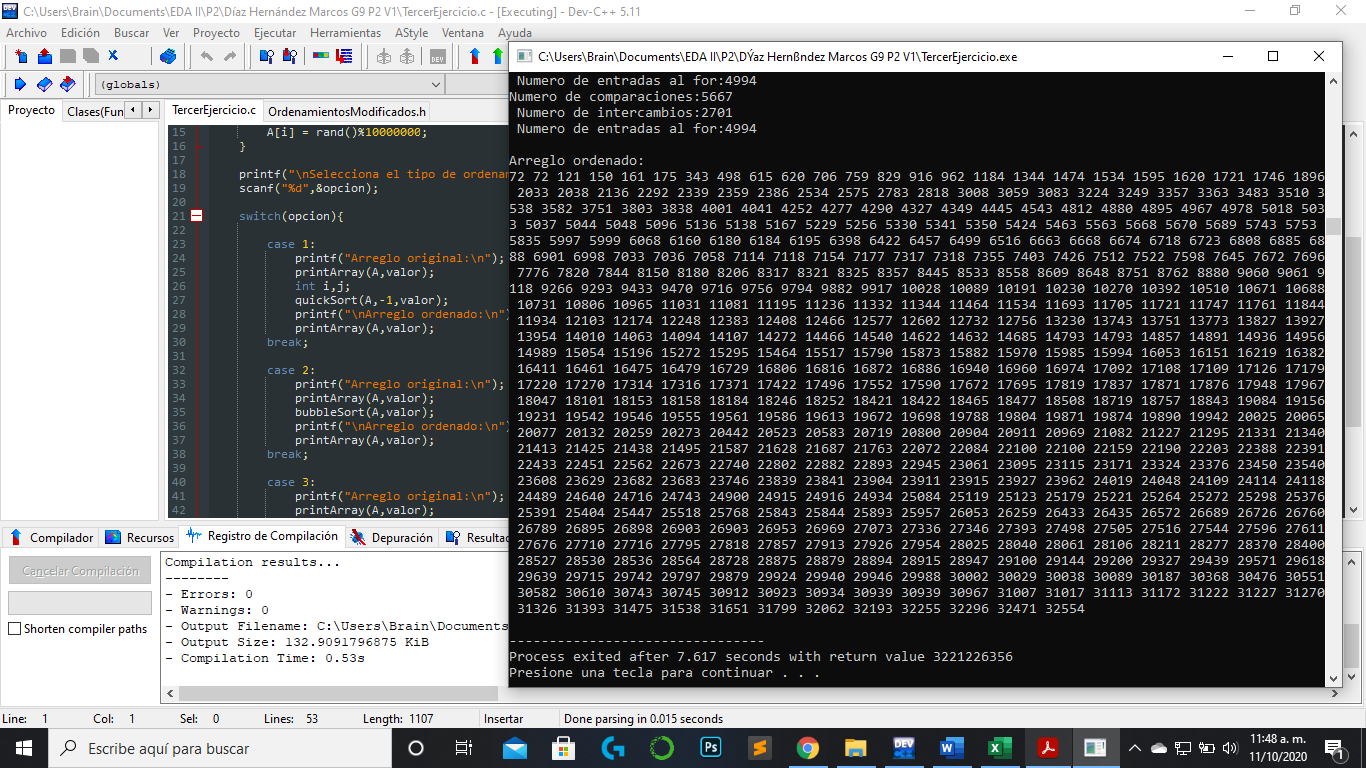
* Evidencia de implementación



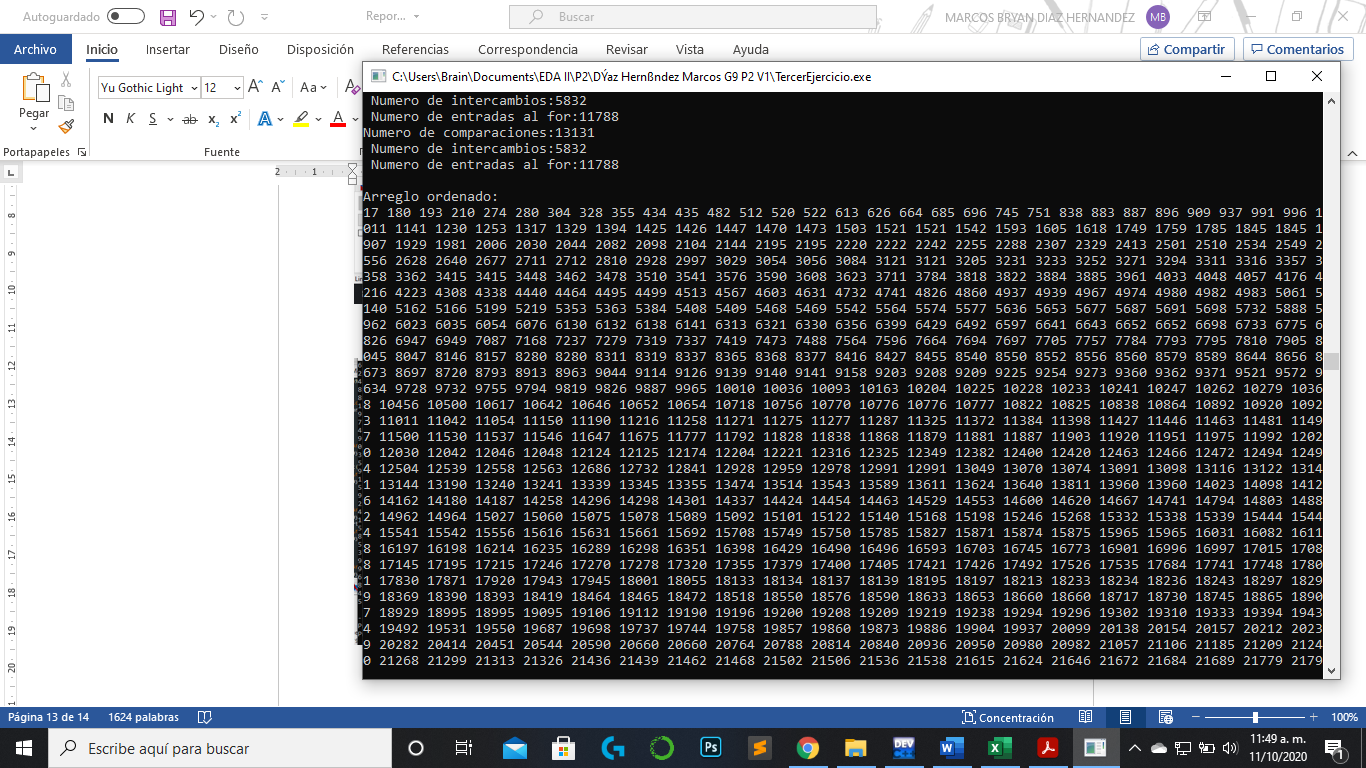
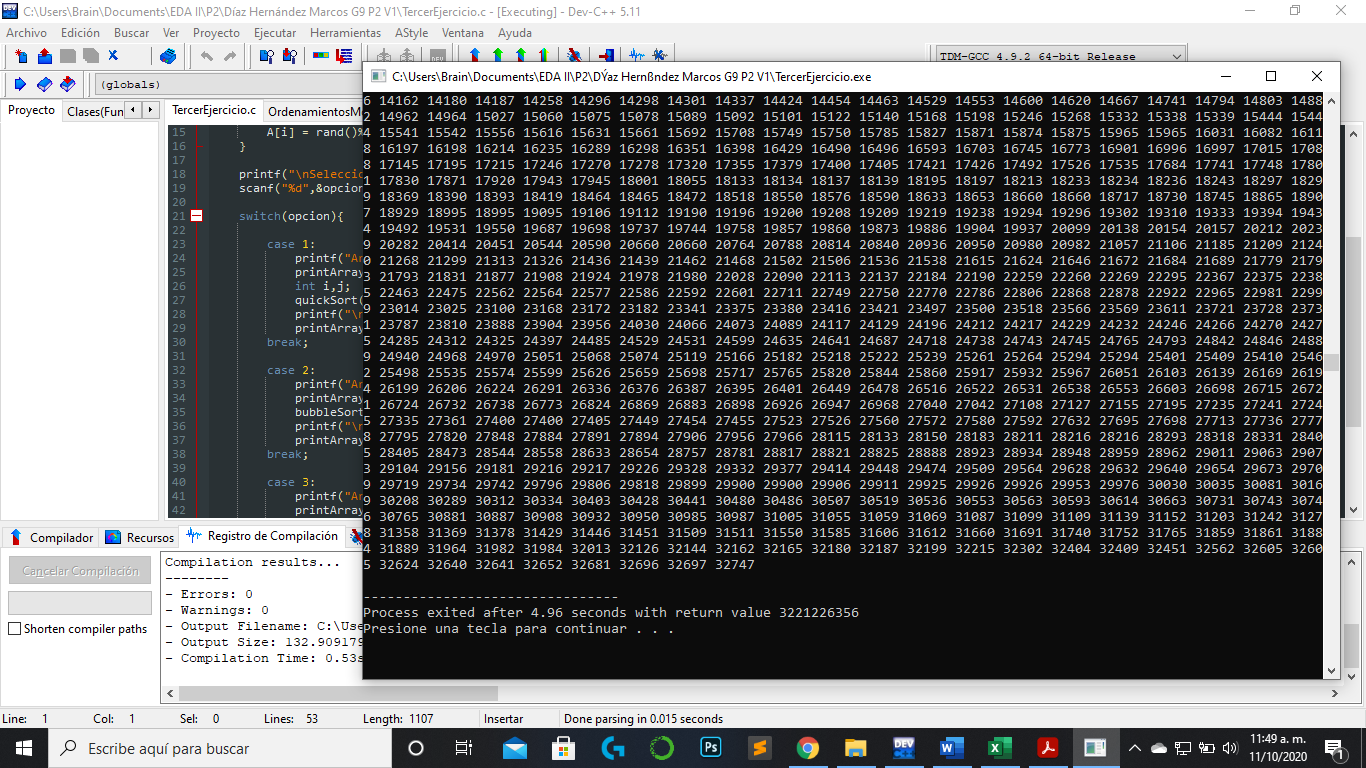
QuickSort – 50



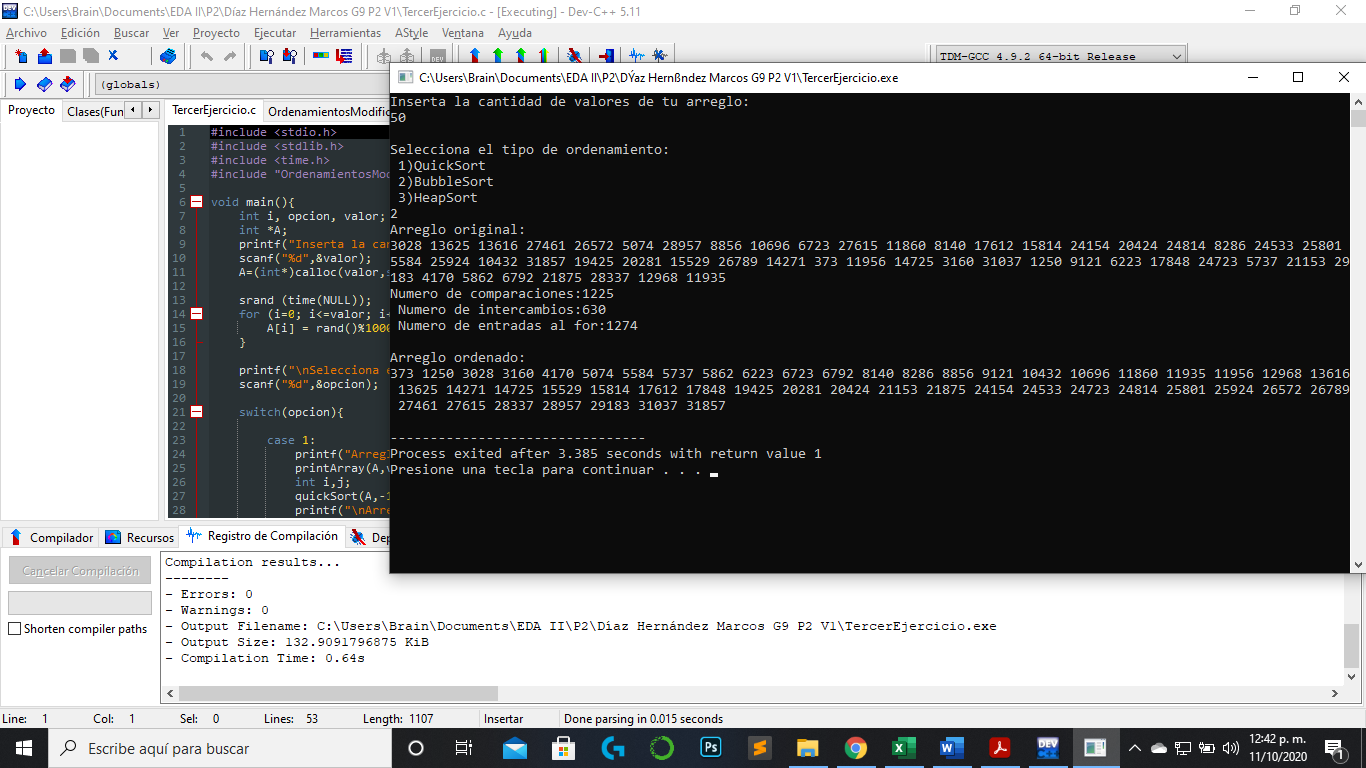
QuickSort - 100



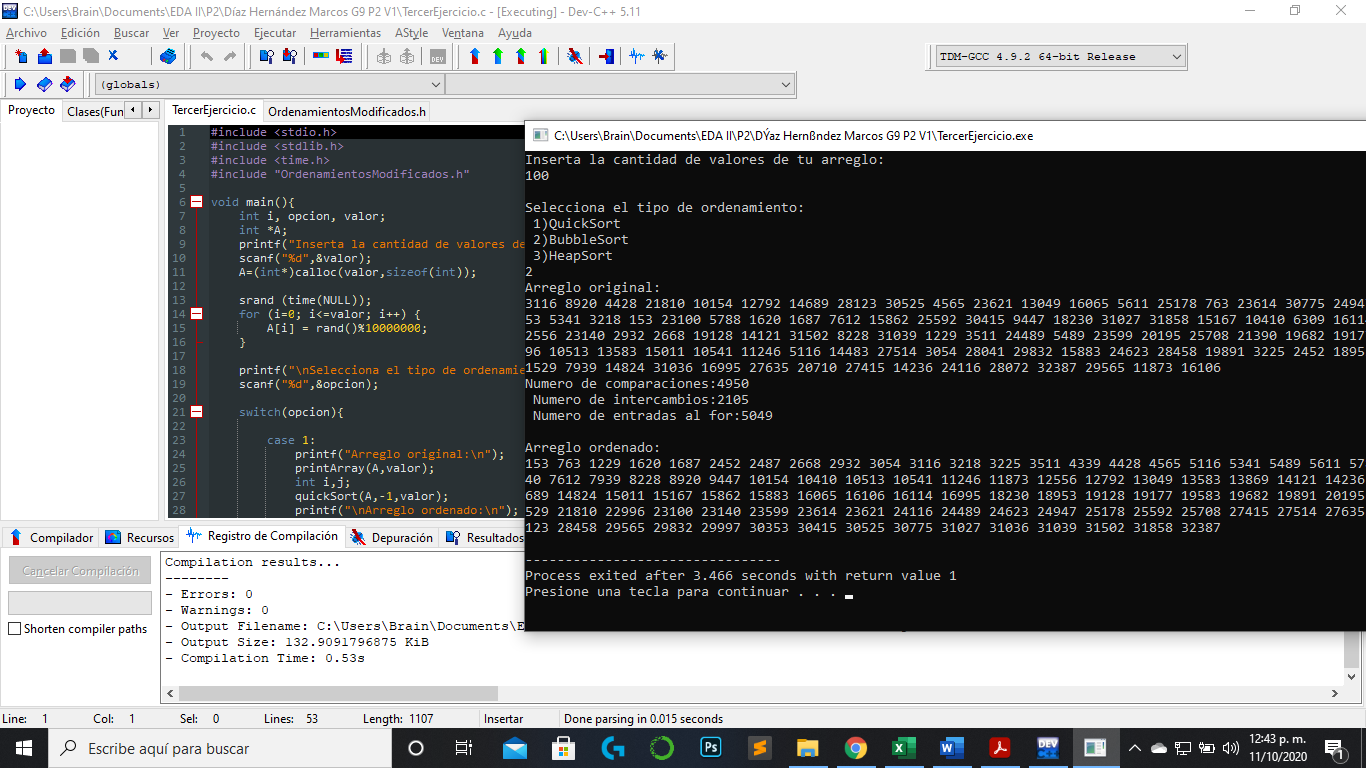
QuickSort – 500



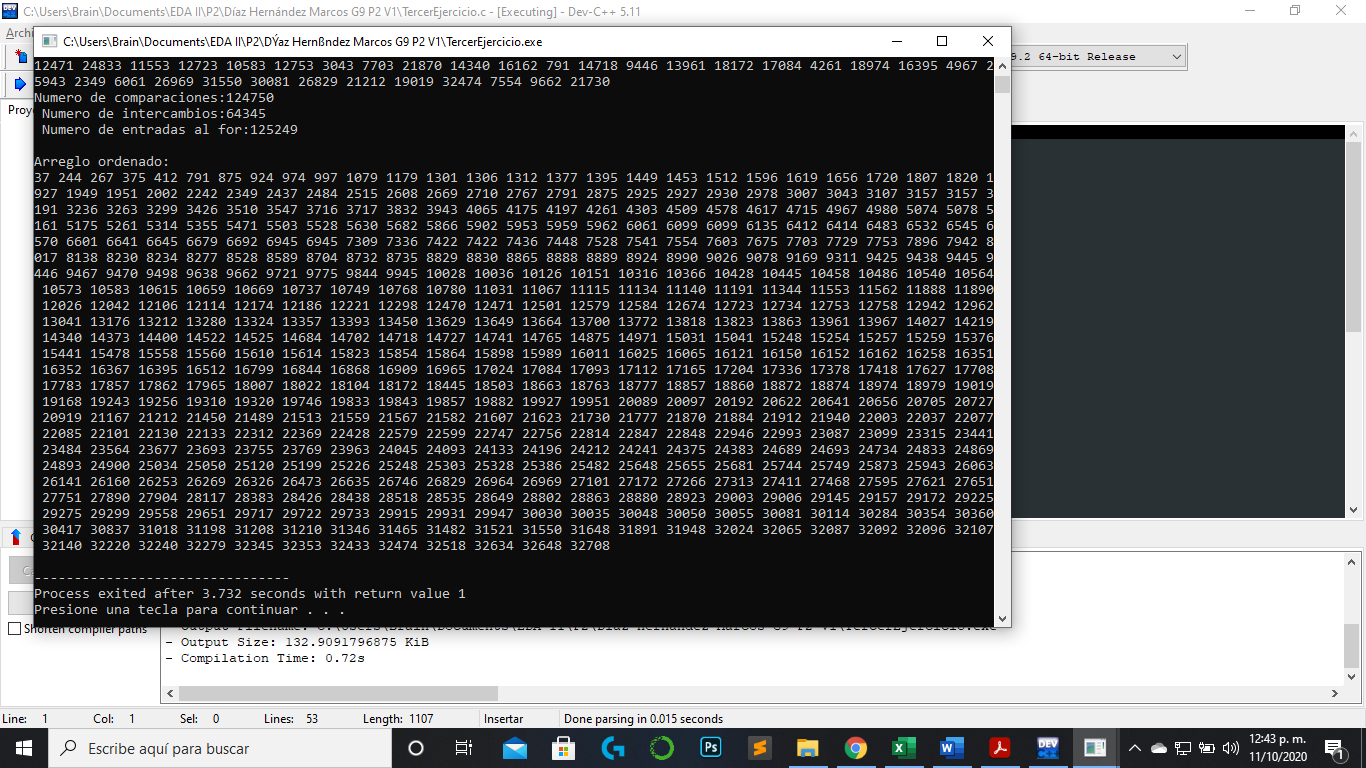
QuickSort - 1000



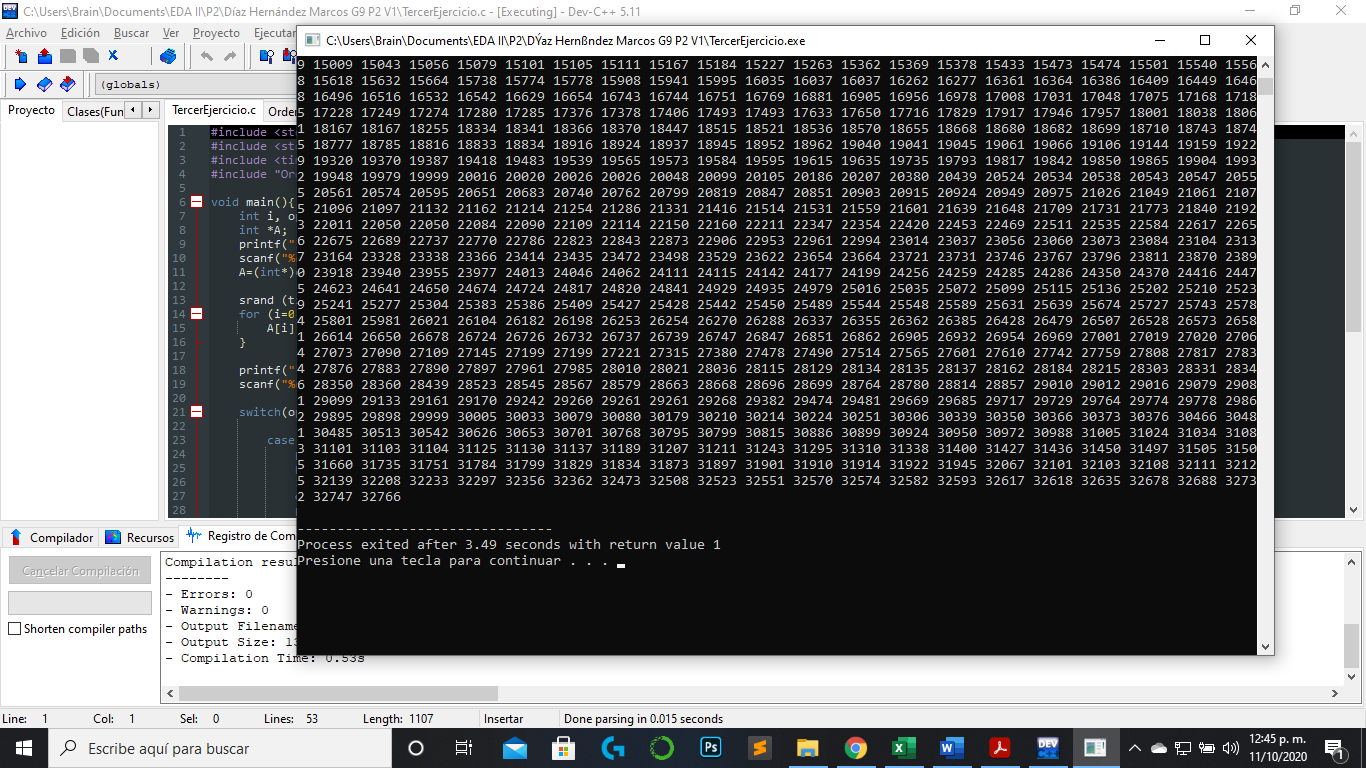
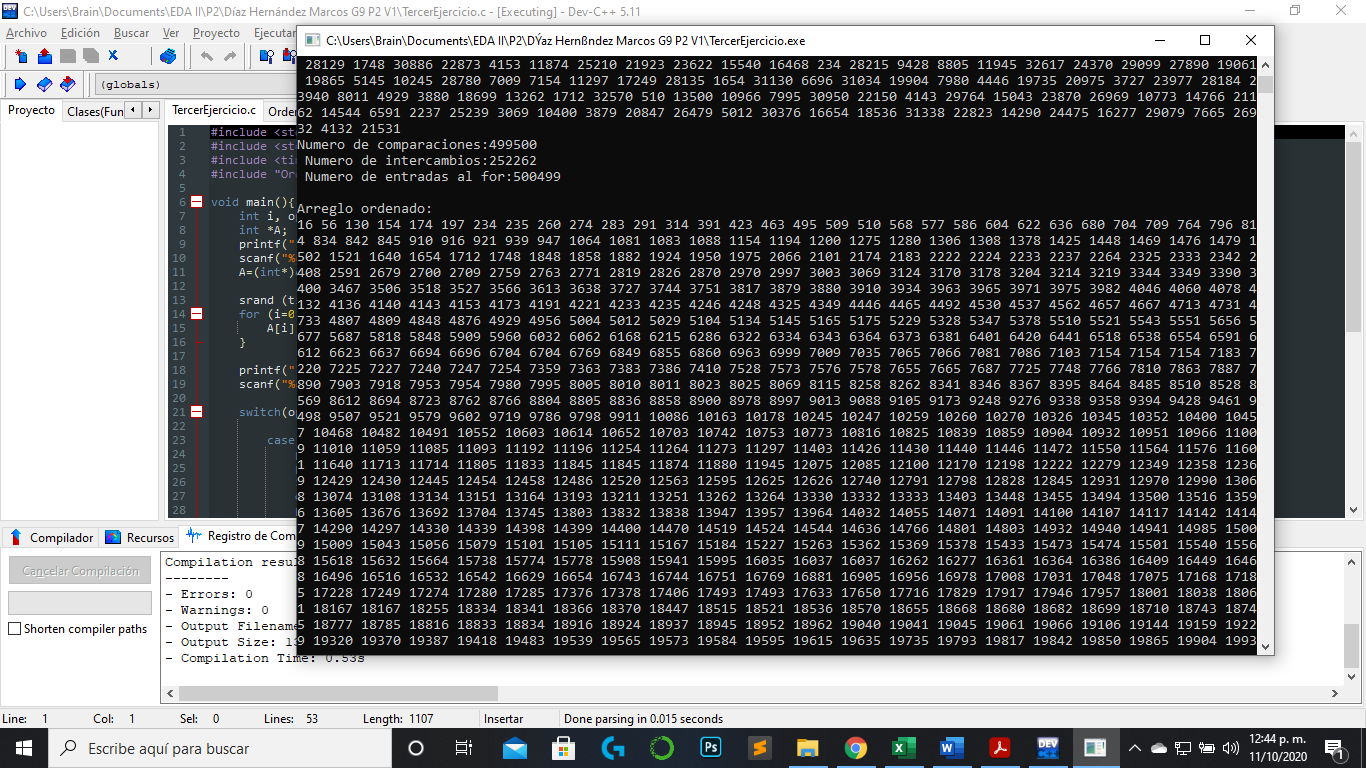
BubbleSort - 50



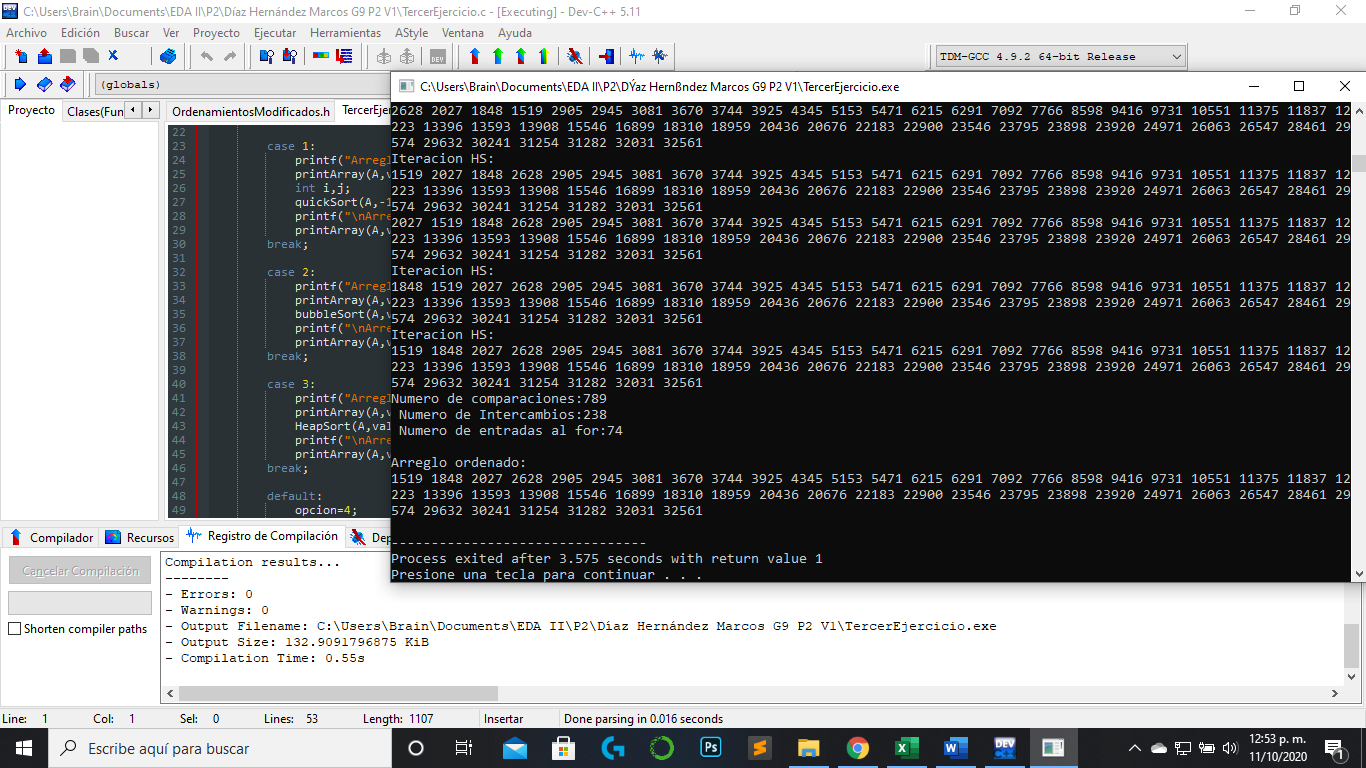
BubbleSort - 100



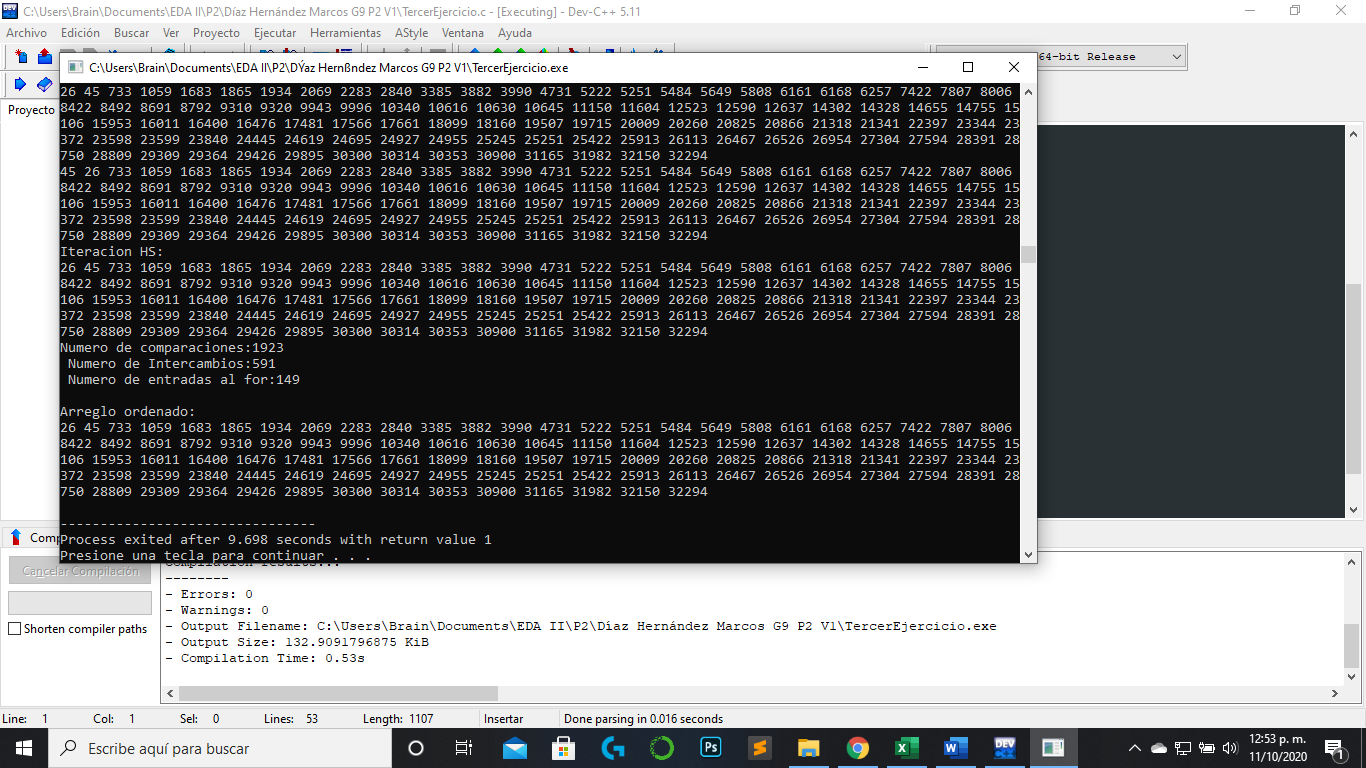
BubbleSort – 500



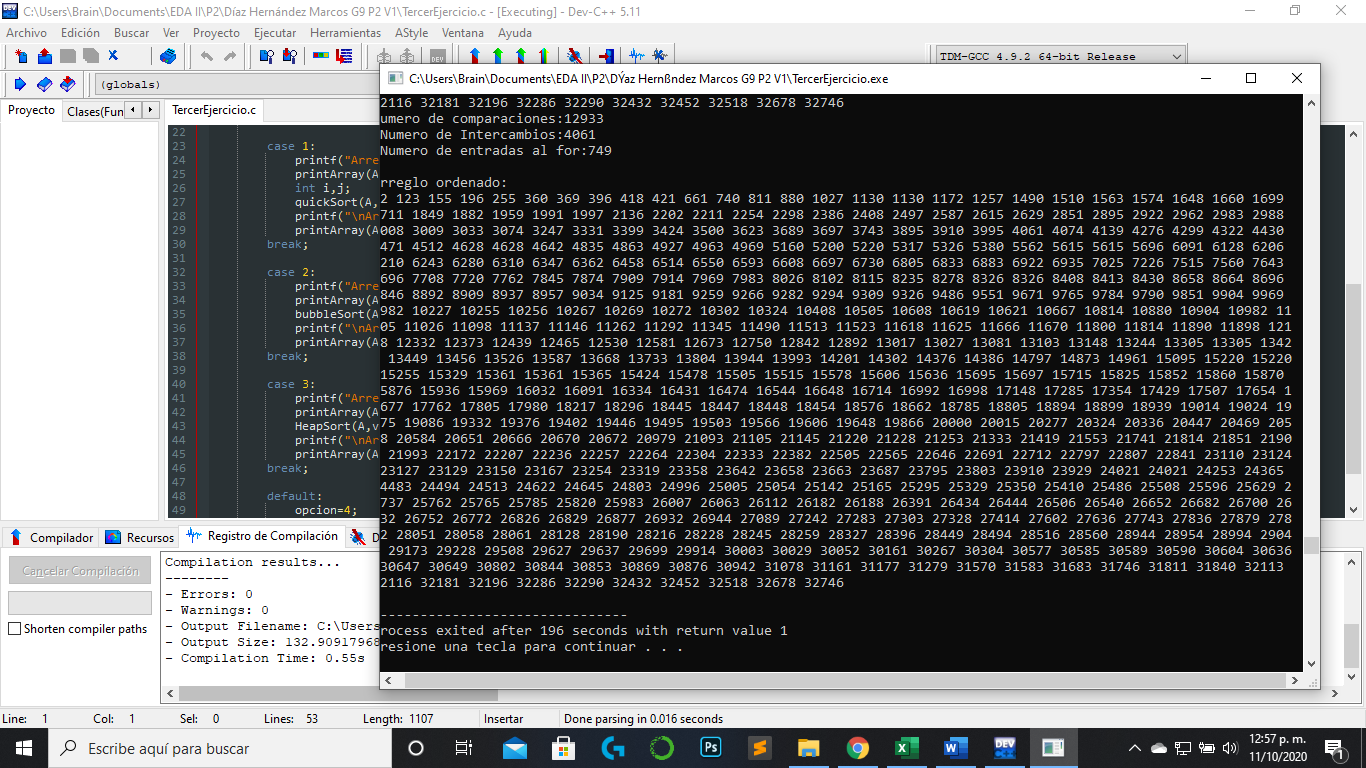
BubbleSort – 1000



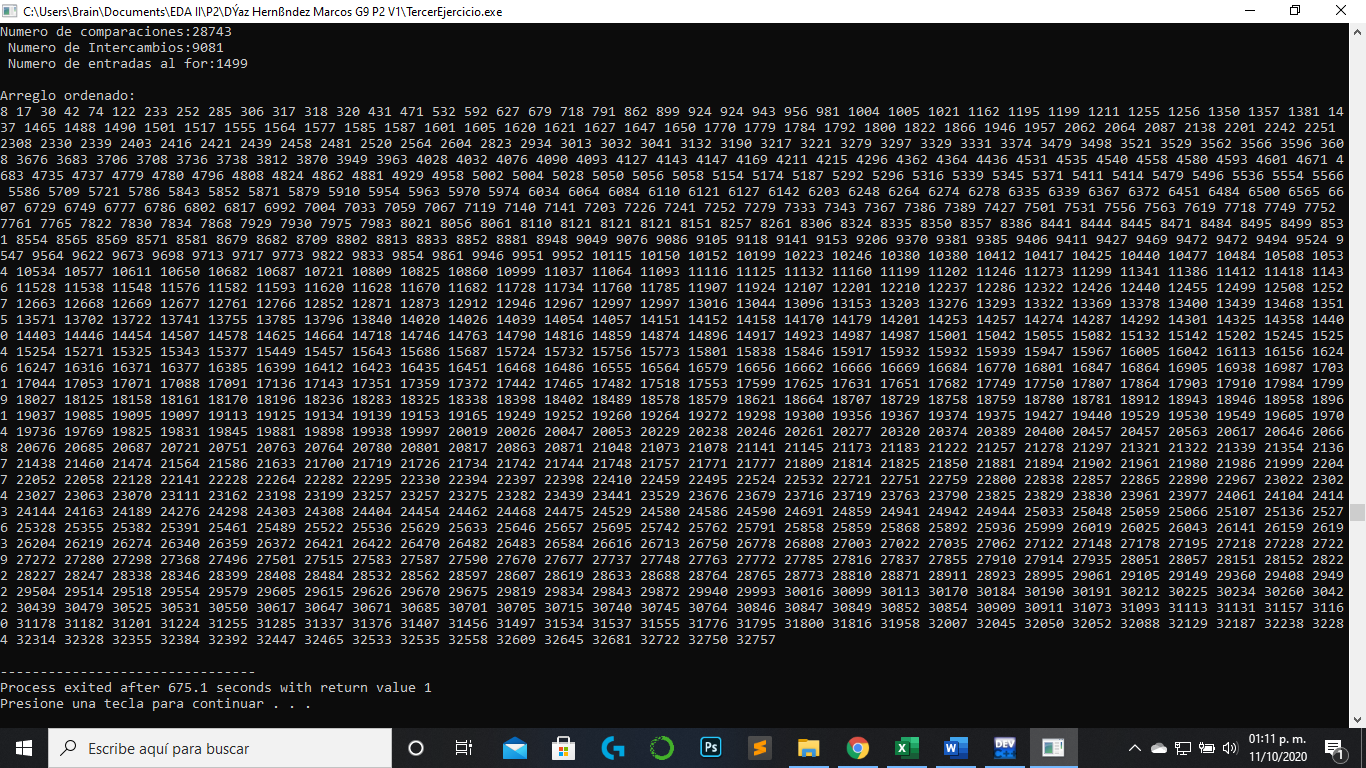
HeapSort - 50



HeapSort - 100



HeapSort – 500



HeapSort - 1000

* Ejercicio 4:

El último ejercicio consistió en la implementación del QuickSort en Java, y pues hacer todo lo que conlleva el poder crear un algoritmo en Java.

* Dificultades en el código

La parte que mas se me dificulto fue la clase QuickSort, debido a que si bien la parte de Partition no tenia problemas, en la parte de que tenia que igualarse al pivote no sabía como, tal vez suene burdo pero pensé que tendría que ser de otra forma, y cuando lo intente poner de la forma común en c resulto que si funciona así (Imagen 1).

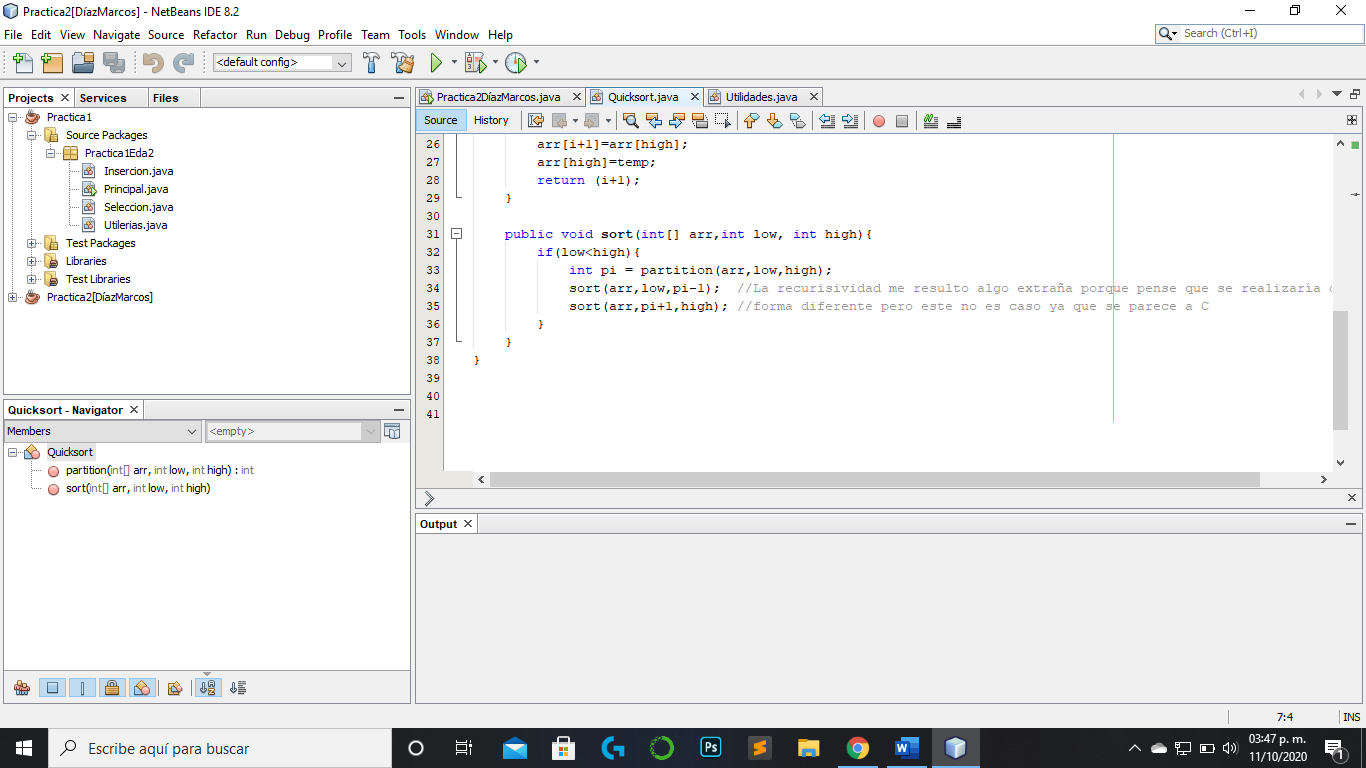


Imagen 1

Dentro del programa hice los respectivos comentarios pero algo que me causa duda o curiosidad es que se tenga que poner un el acceso y también que las clases con el acceso public, tengan que ser iguales al nombre del programa, porque sino causa problema en la compilación, eso no lo vie en este caso pero en al hacerlo con comandos, si pasa eso.

Otra cosa que se me hizo bastante curiosa fue que al tener una clase tienes métodos de esta y puede definir objetos que heredan los atributos y los métodos, y de la misma forma existen métodos que son pertenecientes a la clase, que son estáticos, entonces por esta condición decidí colocar los métodos de QuickSort como de instancia en caso de que se quisieran tener mas arreglos y arreglarlos, para que no se creara un problema por la modificación de los métodos.

En cuanto a la creación de los objetos y el uso de los métodos pues creía que para tener un objeto era necesario definir atributos pero no, con tener métodos basta, y para utilizar los métodos es necesario acceder a ellos por medio del objeto así que es bastante interesante la forma de trabajo en Java (Imagen 2), y por ultimo algo importante que vi es que se necesita tener los métodos en publico de lo contrario otras clases no pueden acceder.

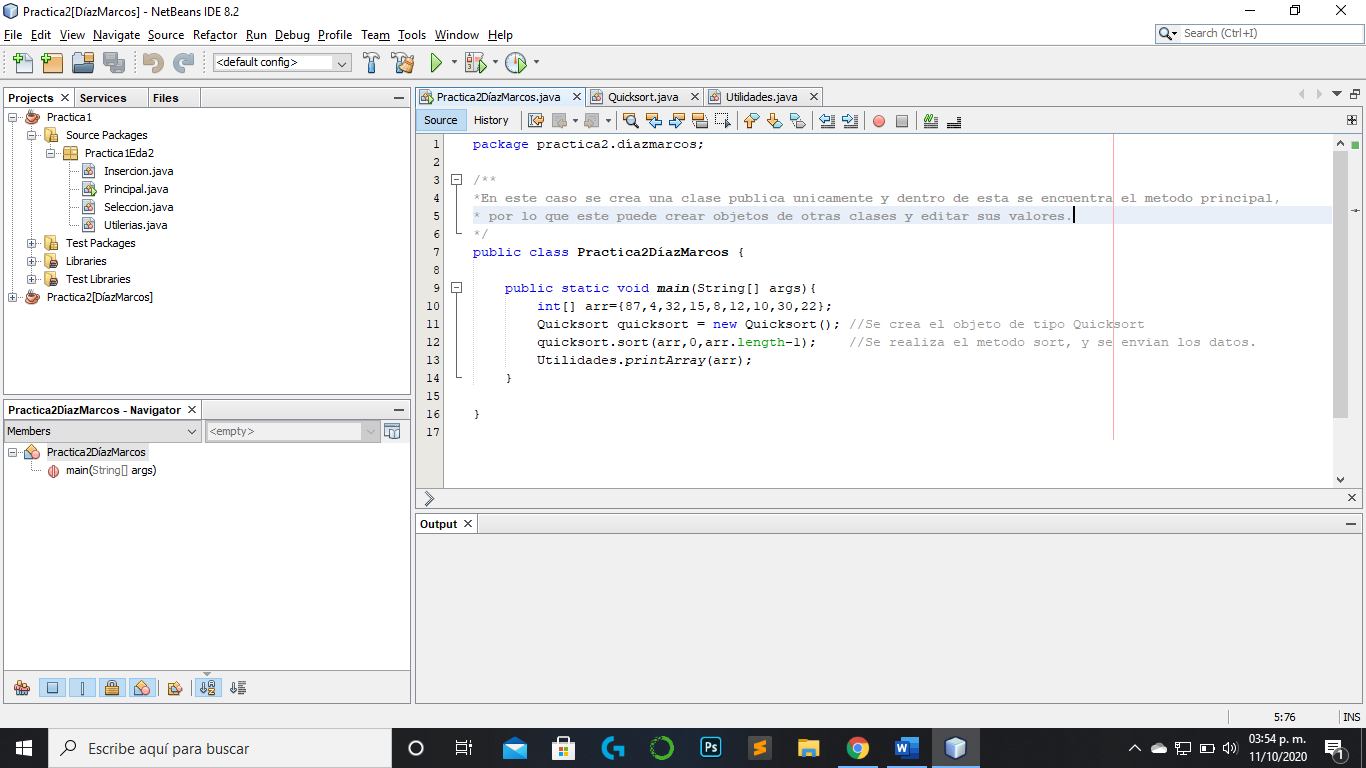
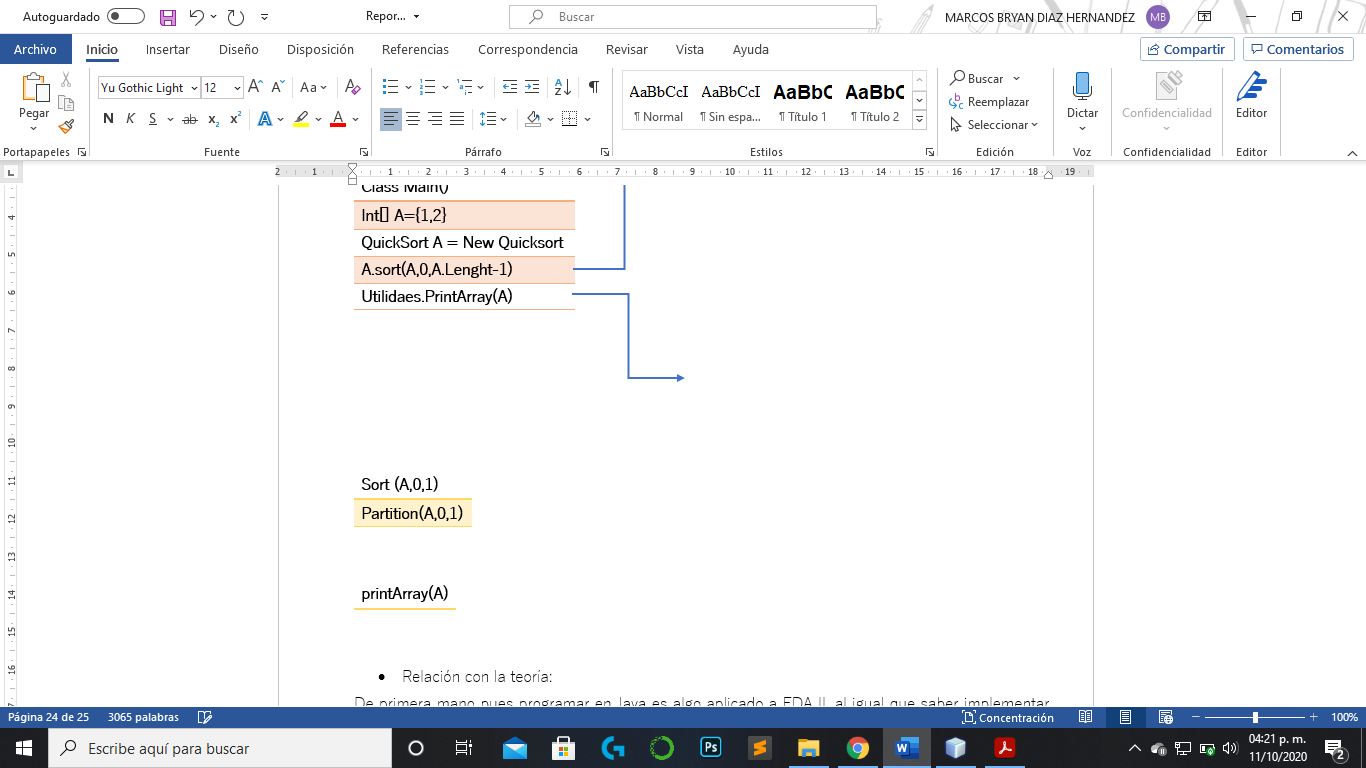


Imagen 2

* Diagrama de funcionamiento:

Public



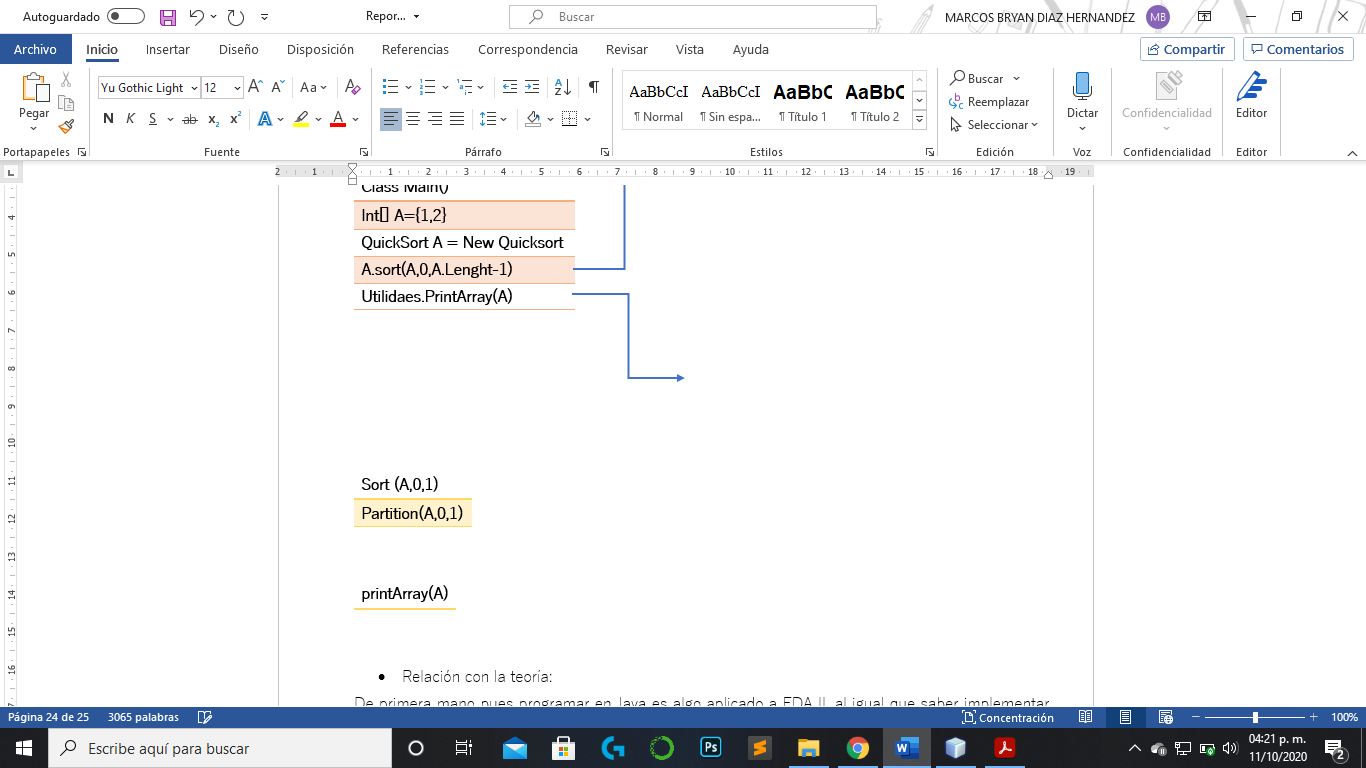
De esta forma es visible la estructura de Java y como llama a las clases que tienen acceso público.

n

Public

Public

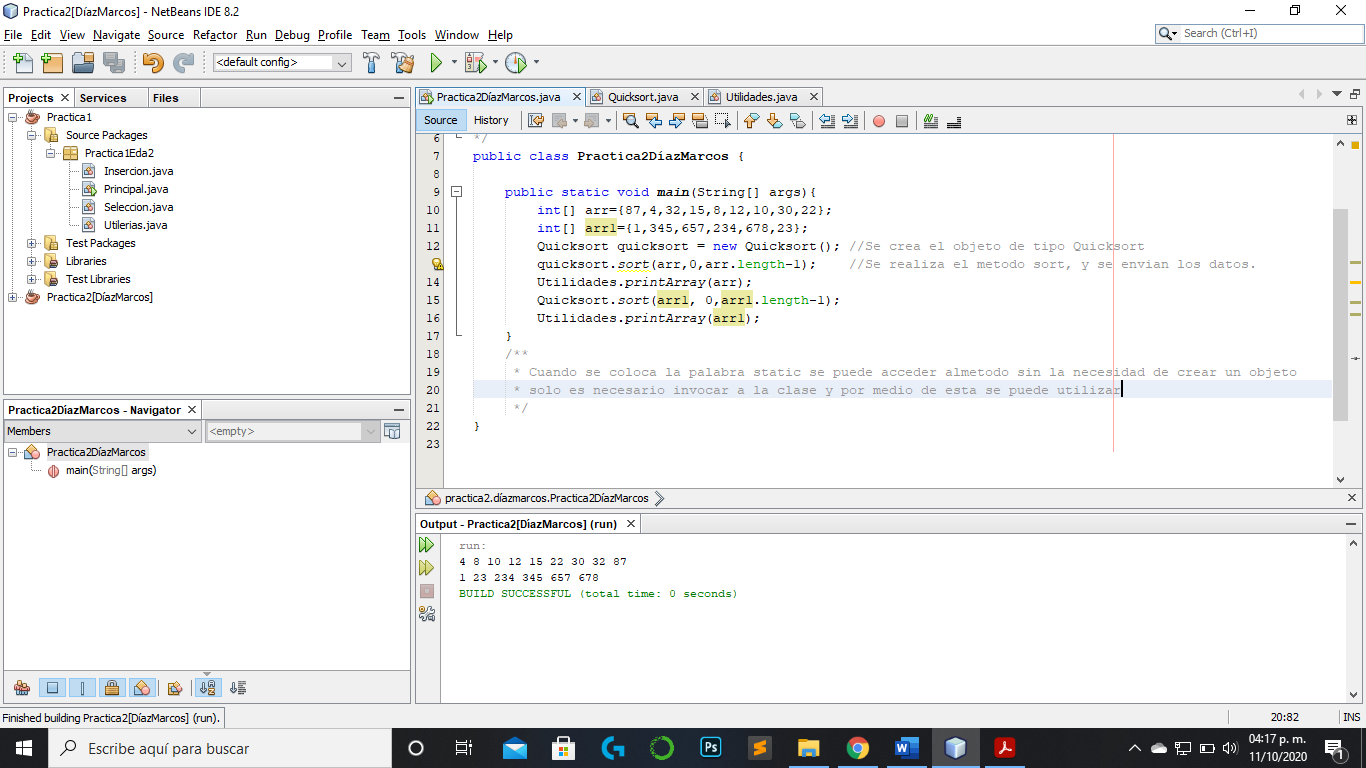
|  |
| --- |
| Class Main() |
| Int[] A={1,2} |
| QuickSort A = New Quicksort |
| A.sort(A,0,A.Lenght-1) |
| Utilidaes.PrintArray(A) |



* Relación con la teoría:

De primera mano pues programar en Java es algo aplicado a EDA II, al igual que saber implementar los algoritmos en cualquier lenguaje, ya que eso permite ver la diferencia entre los lenguajes y los paradigmas que se manejan en cada lenguaje, en POO no llevo muchos programas por lo que se vuelve algo confuso a veces pero poco a poco voy entendiendo la forma de trabajo y mucho mejor si tengo que programar algo.

* Evidencia de implementación



**Conclusiones.**

En esta práctica se llevo a cabo el análisis de los algoritmos de ordenamiento: QuickSort, HeapSort, BubbleSort, los cuales se ejecutaron, se analizo su complejidad, se implementaron en otros lenguajes de programación por medio de otro paradigma, además de ser modificados con la intención de poder comprender las diferencias entre las versiones que existen de algunos de ellos, hacerlos mas eficientes, o simplemente comprender lo que hacían.

Por último el objetivo de la práctica se cumplió, y siento que a la vez pasada mejore el análisis de los algoritmos, a la vez que me metí más al análisis de algunos de ellos ya que antes de realizar los cambios es mucho mejor saber lo que hace en ese momento el algoritmo y en base a eso buscar los puntos que se pueden cambiar o arreglar. Sin mas que decir me llevo conocimiento que sirve para concretar lo visto en las clases.