Ernesto González Pradas

Descripción breve

Se implementan los distintos tipos de algoritmos de ordenación de formas variadas para ver los resultados en tiempo computaiconal.

Laboratorio

Algoritmos de ordenación

íNDICE

[Cuestiones previas 2](#_Toc71996201)

[Partes del Proyecto 2](#_Toc71996202)

[Descripción breve de las clases 2](#_Toc71996203)

[Ejecutamos el programa 3](#_Toc71996204)

[Ejecutamos el main 3](#_Toc71996205)

[Opción1: Ejecutamos los algoritmos por separado 3](#_Toc71996206)

[Opción 2: Ejecutamos los algoritmos a la vez y mostramos gráfica 5](#_Toc71996207)

[Opción 3: Ejecutamos todos los algoritmos a la vez con hilos 6](#_Toc71996208)

[Análisis de los algoritmos 7](#_Toc71996209)

[Ordenación de la burbuja 7](#_Toc71996210)

[Ordenación por inserción 8](#_Toc71996211)

[Ordenación por MergeSort 9](#_Toc71996212)

[Ordenación por QuickSort 10](#_Toc71996213)

[Ordenación por selección 10](#_Toc71996214)

[Conclusiones y reflexiones 12](#_Toc71996215)

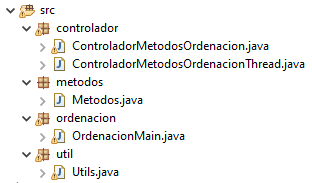
# Cuestiones previas

En el presente trabajo se van a realizar distintas implementaciones de algoritmos de ordenación y de distintas formas. El lenguaje utilizado para esta implementación es java versión 1.8 y se realizará de las siguientes formas:

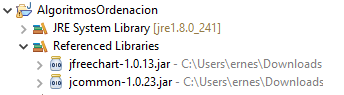
1. Se implementará una opción en la que los algoritmos se ejecutarán por separado y se mostrará al usuario los resultados del tiempo que tarda en ejecutarse cada uno de los algoritmos con el mismo array de enteros generado de forma aleatoria.
2. Se implementará una opción en la que se ejecutarán todos los algoritmos uno detrás de otro, y le mostrará al usuario los tiempos de cada uno de ellos a la vez para que pueda comparar, junto con una gráfica generada por el propio programa indicando dichos tiempos y facilitando la visualización de estos.
3. Se implementará una opción en la que los algoritmos se ejecutan a la vez utilizando hilos para ver si se aprovechan los núcleos del procesador y ver si los tiempos de dichos algoritmos son distintos o no.

## Partes del Proyecto

El proyecto consta de varias partes. Por un lado, tenemos los paquetes (controlador, métodos, ordenación y util) que a su vez tienen las clases necesarias para el desarrollo de este laboratorio:



Y, por otro lado, tenemos un par de librerías externas que se han introducido en el proyecto para poder pintar los gráficos del tiempo de cada uno de los algoritmos:



## Descripción breve de las clases

Dentro del paquete controlador, tenemos las clases *ControladorMetodosOrdenacion.java* y *ControladorMetodosOrdenacionThread.java*. En estos controladores gestionamos las implementaciones de los distintos métodos de ordenación y controlamos los tiempos que tardan cada uno de ellos.

Dentro del paquete métodos tenemos la clase *Metodos.java*. En esta clase, tenemos declarados los distintos métodos de ordenación (selección, burbuja, inserción, mergesort y quicksort).

En el paquete ordenación, tenemos la clase java *OrdenacionMain.jav*a en la que se encuentra el método main para ejecutar toda la aplicación.

Y finalmente, en el paquete útil, tenemos la clase java *Utils.java* en la que disponemos de distintos métodos o utilidades como sería la generación del array aleatorio de enteros o la generación de los menús necesarios para interactuar con la aplicación.

## Ejecutamos el programa

Si abrimos el código enviado en un compilador como eclipse, por ejemplo, nos situaremos en el paquete ordenación clase *OrdenacionMain.java* y le daremos al botón de run:



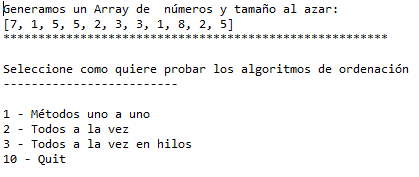
Si ejecutamos el jar adjuntado en el zip mediante la consola de comandos ejecutaríamos el siguiente comando:



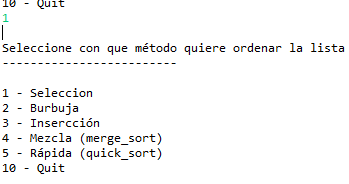
# Ejecutamos el main

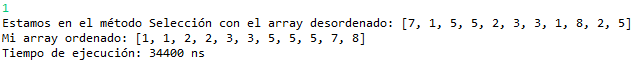
## Opción1: Ejecutamos los algoritmos por separado

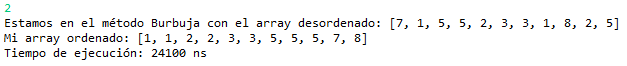
Al ejecutar el programa nos aparecerán varias cosas. La primera es el array generado aleatoriamente tanto en tamaño como en valores. Además, nos aparecerá un menú con tres opciones. En este caso seleccionaremos la primera de ellas:



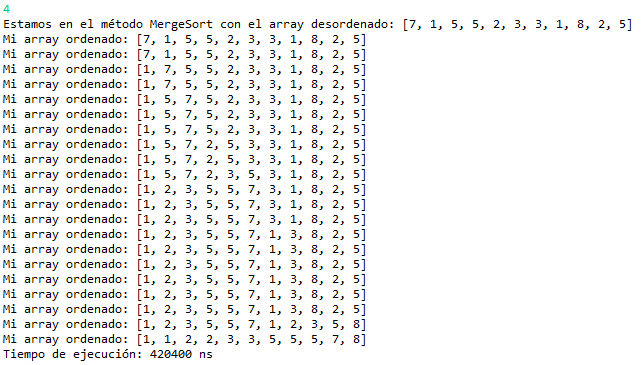
Después de seleccionar la opción “1”, nos aparecerá un segundo menú para elegir qué tipo algoritmo queremos seleccionar para realizar la ordenación y ver los resultados del tiempo que tarda. A continuación, se muestran capturas de todos los algoritmos ejecutados y sus tiempos:

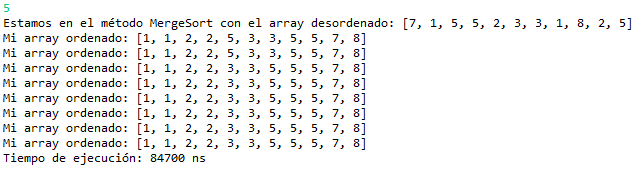








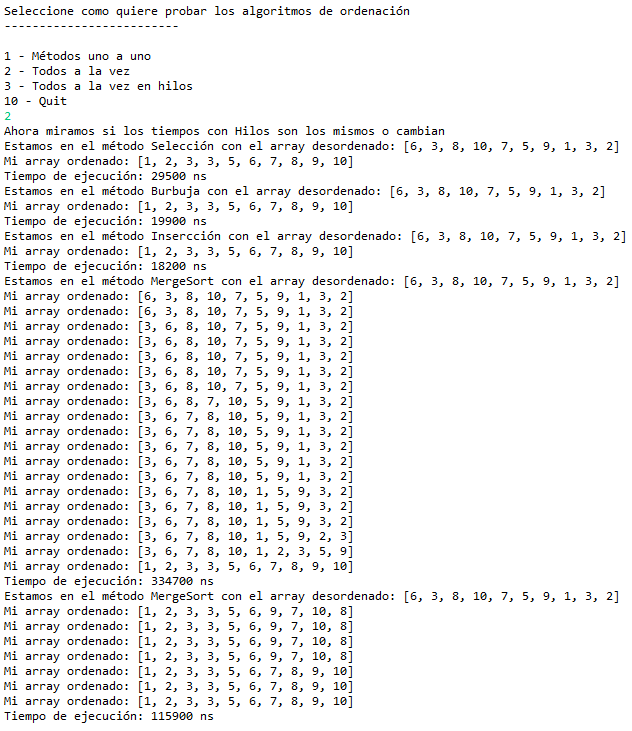




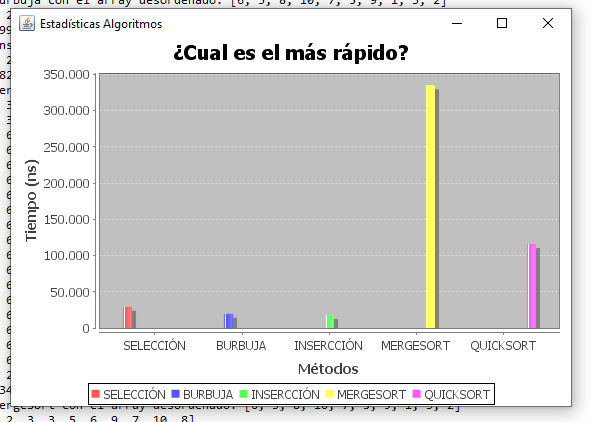
Como se puede apreciar en los algoritmos mergesort y quick sort se han puesto las salidas por consola para ver como en cada llamada recursiva se va ordenando el array en el lado izquierdo y en el derecho.

## Opción 2: Ejecutamos los algoritmos a la vez y mostramos gráfica

Realizamos los mismos pasos que en la opción anterior, pero esta vez en el primer menú seleccionamos la opción “2”.

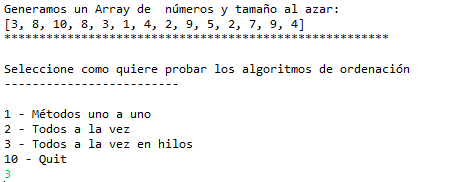


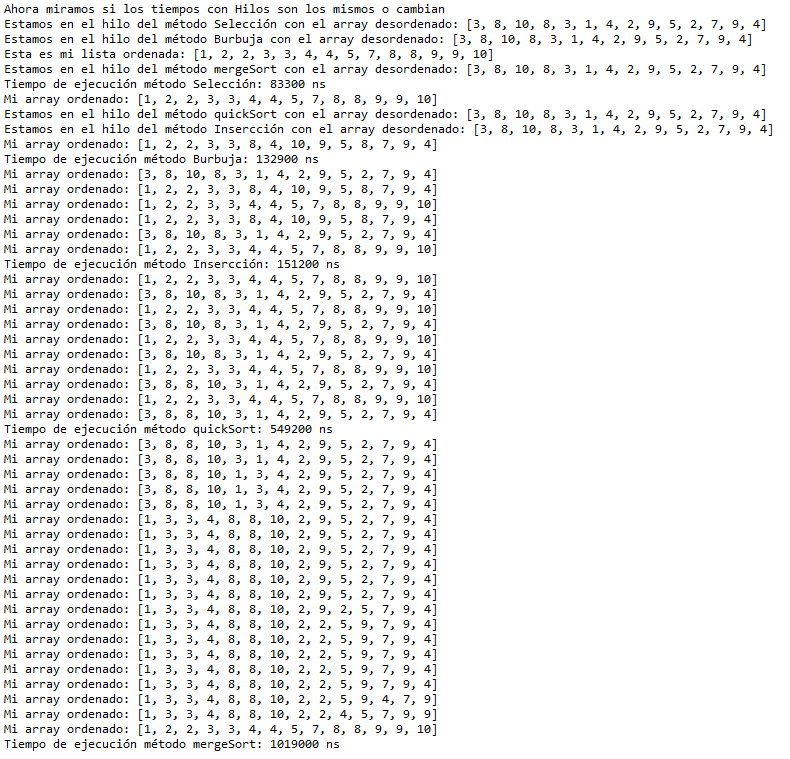
Y la gráfica que generamos a partir de estos datos sería la siguiente:



## Opción 3: Ejecutamos todos los algoritmos a la vez con hilos

Ejecutamos la tercera opción del menú visto en los apartados anteriores.:





# Análisis de los algoritmos

A continuación, siguiendo con la actividad 1 y viendo los resultados obtenidos en los tiempos de estos algoritmos, vamos a realizar un análisis de los algoritmos utilizados en la implementación del código. Vamos a ver que costes computacionales tienen cada uno de ellos.

## Ordenación de la burbuja

Para empezar a analizar este algoritmo vamos a ver que implementación se ha utilizado:

**public** **void** burbuja(**int**[] A) {

**int** i, j, aux;

**for** (i = 0; i < A.length; i++) {

**for** (j = 0; j < A.length - i - 1; j++) {

**if** (A[j + 1] < A[j]) {

aux = A[j + 1];

A[j + 1] = A[j];

A[j] = aux;

}

}

}

System.***out***.println("Mi array ordenado: " + Arrays.*toString*(A));

}

Tenemos dos bucles anidados y un condicional anidado también que dependen de la entrada de forma directa, es decir, dependen del tamaño del array. Por ello podemos decir de forma intuitiva, que la cota superior será **O(n2)**. Vamos a analizarlo un poco mejor:

* Tenemos un primer bucle que depende totalmente de la longitud del array de entrada, por lo que tendrá un coste computacional de **C1(n) = n**:

**for** (i = 0; i < A.length; i++)

* Tenemos un segundo bucle que depende nuevamente del tamaño del array de entrada, por lo tanto, tendrá un coste computacional de **C2(n) = n**:

**for** (j = 0; j < A.length - i - 1; j++)

* Por ultimo, tenemos un if anidado que siempre va a realizar la misma comparación, por lo que tendrá un coste computacional constante **O(1)**.

Al ser dos bucles y un condicional anidados la teoría nos dice que para calcular el coste computacional total debemos aplicar **la regla del producto**, quedando un coste computacional total:

CT(n) = C1(n) \* C2(n) \* C3(n) = n \* n \* 1 = n2

Y por lo tanto su orden de complejidad será:

O1(n) = n

O2(n) = n O(CT(n)) = n \* n \* 1= n2 => CT(n) = n \* n \* 1 ϵ O(n2)

O3(n) = 1

## Ordenación por inserción

Para empezar a analizar este algoritmo vamos a ver que implementación se ha utilizado:

**public** **void** inserccion(**int**[] A) {

**int** n = A.length;

**for** (**int** j = 1; j < n; j++) {

**int** key = A[j];

**int** i = j-1;

**while** ( (i > -1) && ( A [i] > key ) ) {

A [i+1] = A [i];

i--;

}

A[i+1] = key;

}

System.***out***.println("Mi array ordenado: " + Arrays.*toString*(A));

}

Tenemos dos bucles anidados, uno de ellos es un while que para que se siga ejecutando tiene que cumplir una condición, y ambos dependen de la entrada de forma directa, es decir, dependen del tamaño del array. Por ello podemos decir de forma intuitiva, que la cota superior será **O(n2)**. Vamos a analizarlo un poco mejor:

* Tenemos un primer bucle que depende totalmente de la longitud del array de entrada, por lo que tendrá un coste computacional de **C1(n) = n**:

**for** (**int** j = 1; j < n; j++)

* Tenemos un segundo bucle que para que siga ejecutándose tiene que cumplir una condición, el cual realiza desplazamientos para dejar sitio a los elementos. Su coste computacional será **C2(n) = n**:

**while** ( (i > -1) && ( A [i] > key ) )

Al ser dos bucles anidados la teoría nos dice que para calcular el coste computacional total debemos aplicar **la regla del producto**, quedando un coste computacional total:

CT(n) = C1(n) \* C2(n) = n \* n = n2

Y por lo tanto su orden de complejidad será:

O1(n) = n

O(CT(n)) = n \* n = n2 => CT(n) = n \* n ϵ O(n2)

O2(n) = n

## Ordenación por MergeSort

Para empezar a analizar este algoritmo vamos a ver que implementación se ha utilizado:

**public** **void** mergesort(**int** A[],**int** izq, **int** der){

**if** (izq < der){

**int** m=(izq+der)/2;

mergesort(A,izq, m);

mergesort(A,m+1, der);

*merge*(A,izq, m, der);

}

System.***out***.println("Mi array ordenado: " + Arrays.*toString*(A));

}

**public** **static** **void** merge(**int** A[],**int** izq, **int** m, **int** der){

**int** i, j, k;

**int** [] B = **new** **int**[A.length]; //array auxiliar

**for** (i=izq; i<=der; i++) //copia ambas mitades en el array auxiliar

B[i]=A[i];

i=izq; j=m+1; k=izq;

**while** (i<=m && j<=der) //copia el siguiente elemento más grande

**if** (B[i]<=B[j])

A[k++]=B[i++];

**else**

A[k++]=B[j++];

**while** (i<=m) //copia los elementos que quedan de la

A[k++]=B[i++]; //primera mitad (si los hay)

}

La verdad que he intentado analizar este algoritmo, pero no he sabido como meterle mano ya que en clase hemos visto como analizar algoritmos recursivos utilizando la reducción por división y la reducción por sustracción, y sabría como aplicarlo aquí pero buscando en el temario el coste computacional de este método es **C(n)=nlog2n.**

## Ordenación por QuickSort

Para empezar a analizar este algoritmo vamos a ver que implementación se ha utilizado:

**public** **void** quicksort(**int** A[], **int** izq, **int** der) {

**int** pivote=A[izq];

**int** i=izq;

**int** j=der;

**int** aux;

**while**(i < j){

**while**(A[i] <= pivote && i < j) i++; **while**(A[j] > pivote) j--;

**if** (i < j) {

aux= A[i];

A[i]=A[j];

A[j]=aux;

}

}

A[izq]=A[j];

A[j]=pivote;

**if**(izq < j-1)

quicksort(A,izq,j-1);

**if**(j+1 < der)

quicksort(A,j+1,der);

System.***out***.println("Mi array ordenado: " + Arrays.*toString*(A));

}

Con este algoritmo me pasa un poco como con el quickSort, no tengo muy claro como abordarlo a la hora de analizarlo (soy capaz de distinguir los bloques y las llamadas recursivas). Siguiendo el temario el coste computacional de este algoritmo es **Cworst(n)=n2**.

## Ordenación por selección

Este método lo he dejado para el final ya que lo he modificado pensando en que sea más óptimo y en el apartado de Conclusiones y reflexiones pongo los dos códigos y las diferencias en cuanto a la mejora en el tiempo que supone esta optimización. Aunque vamos a analizar el código optimizado el algoritmo “sin optimizar” tendría el mismo coste computacional ya que tiene los mismos grandes bloques:

**public** **void** seleccion2(**int** A[]) {

**for** (**int** i = 0; i < A.length - 1; i++) {

**for** (**int** j = i + 1; j < A.length; j++) {

**if** (A[i] > A[j]) {

//mover el actual a la derecha y el de la derecha al actual

**int** temporal = A[i];

A[i] = A[j];

A[j] = temporal;

}

}

}

System.***out***.println("Mi array ordenado: " + Arrays.*toString*(A));

}

Tenemos dos bucles anidados y un condicional anidado también que dependen de la entrada de forma directa, es decir, dependen del tamaño del array. Por ello podemos decir de forma intuitiva, que la cota superior será **O(n2)**. Vamos a analizarlo un poco mejor:

* Tenemos un primer bucle que depende totalmente de la longitud del array de entrada, por lo que tendrá un coste computacional de **C1(n) = n**:

**for** (**int** i = 0; i < A.length - 1; i++)

* Tenemos un segundo bucle que depende nuevamente del tamaño del array de entrada, por lo tanto, tendrá un coste computacional de **C2(n) = n**:

**for** (**int** j = i + 1; j < A.length; j++)

* Por ultimo, tenemos un if anidado que siempre va a realizar la misma comparación, por lo que tendrá un coste computacional constante **O(1)**.

Al ser dos bucles y un condicional anidados la teoría nos dice que para calcular el coste computacional total debemos aplicar **la regla del producto**, quedando un coste computacional total:

CT(n) = C1(n) \* C2(n) \* C3(n) = n \* n \* 1 = n2

Y por lo tanto su orden de complejidad será:

O1(n) = n

O2(n) = n O(CT(n)) = n \* n \* 1= n2 => CT(n) = n \* n \* 1 ϵ O(n2)

O3(n) = 1

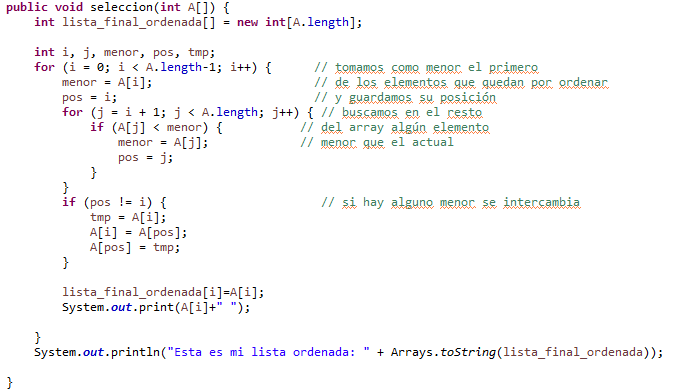
# Conclusiones y reflexiones

He querido realizar el estudio de los costes computacionales porque antes de realizar las pruebas pensaba que los algoritmos tanto MergeSort como QuickSort (que tienen un coste computacional menor que los otros algoritmos) iban a tardar menos tiempo que el resto de algoritmos, pero al realizar distintas pruebas siempre me dan tiempos más elevados, sobre todo en MergeSort. Podría ser, que la implementación en el código no esté bien realizada o que no esté midiendo los tiempos bien, lo he repasado varias veces y creo que está bien, pero me resulta realmente interesante.

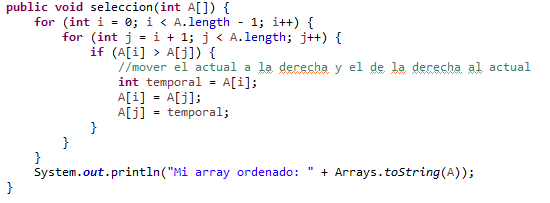
Por otro lado, y siguiendo un poco con la idea del párrafo anterior, quise probar a realizar la ejecución de todos los algoritmos a la vez para ver si conseguía reducir los tiempos, aprovechando que mi procesador tiene 8 núcleos y por lo tanto pensaba que los tiempos iban a ser menores. Pero la realidad es que, con diversas pruebas, generando arrays más largos y cortos la relación de los tiempos entre los algoritmos era muy similar y no había mucha diferencia entre hacer la ejecución de dichos algoritmos con hilos o sin ellos. Esto me puede llevar a pensar que a lo mejor tampoco estaba realizando bien la implementación de los hilos, aunque también creo que es correcta su implementación.

Y por último, buscando información en internet vi varias páginas y foros en los que la gente compartía estos algoritmos optimizados y me decidí a probar varios de ellos. En cuanto a tiempo de ejecución la mayoría salían tiempos prácticamente idénticos, pero encontré uno de selección que si reducía el tiempo algo más. A continuación paso a mostrar el código y las gráficas del tiempo en comparación:

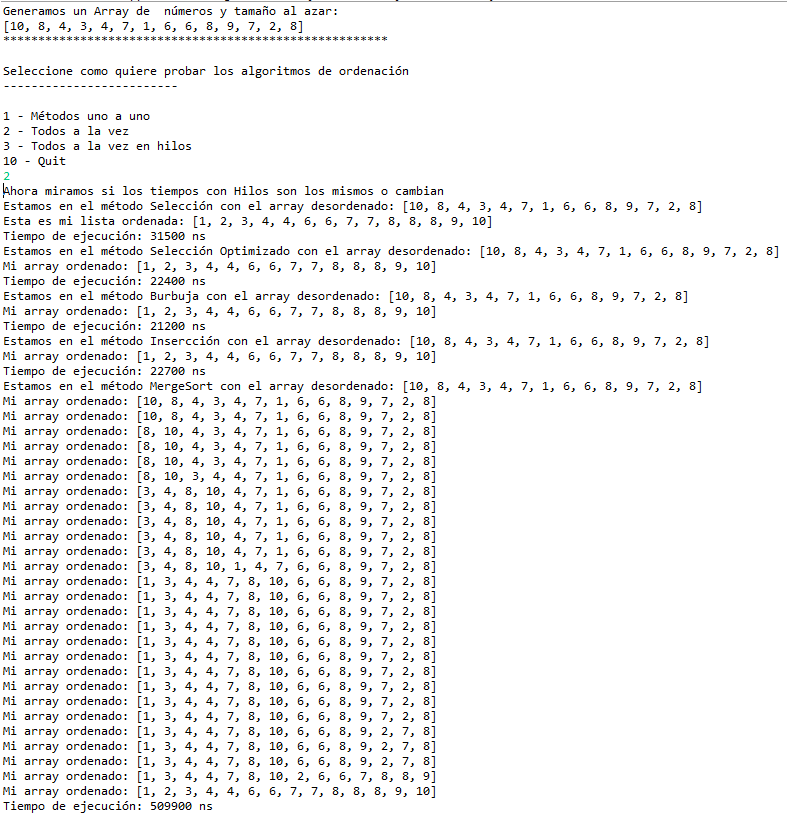
**Código visto en clase**

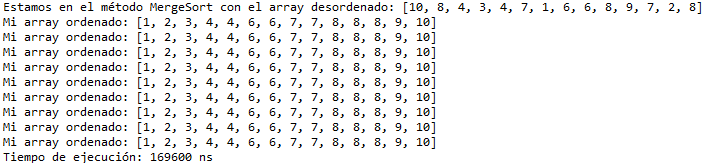


**Código optimizado**



Ejecutamos de nuevo el programa, esta vez comparando los dos códigos de selección:





Y su correspondiente gráfica:

