[I. Resumen II](#_Toc398113758)

**Tabla de contenido**

[II. Agradecimiento III](#_Toc398113759)

[1. Introducción 1](#_Toc398113760)

[2. Marco teórico 3](#_Toc398113761)

[2.1 Algoritmo 3](#_Toc398113762)

[2.2 Ordenamiento 4](#_Toc398113763)

[2.3 Efectividad 5](#_Toc398113764)

[2.4 Uso de recursos (hardware) 7](#_Toc398113765)

[2.5 Explicación breve de los algoritmos a usar 8](#_Toc398113766)

[2.5.1 Algoritmo Bubble sort 8](#_Toc398113767)

[2.5.2 Algoritmo Selection sort 10](#_Toc398113768)

[2.5.3 Algoritmo Insertion sort 12](#_Toc398113769)

[2.5.4 Algoritmo Quick sort 14](#_Toc398113770)

[2.5.5 Algoritmo Merge sort 16](#_Toc398113771)

[3 Metodología 18](#_Toc398113772)

[3.1 Diseño del experimento 18](#_Toc398113773)

[3.2 Aspectos a tomar en cuenta 19](#_Toc398113774)

[4 Pruebas 21](#_Toc398113775)

[4.1 Bubble sort 21](#_Toc398113776)

[4.2 Selection sort 22](#_Toc398113777)

[4.3 Insertion sort 23](#_Toc398113778)

[4.4 Quick sort 23](#_Toc398113779)

[4.5 Merge sort 24](#_Toc398113780)

[5 Análisis 26](#_Toc398113781)

[5.1 Bubble sort 26](#_Toc398113782)

[5.2 Selection sort 27](#_Toc398113783)

[5.3 Insertion sort 28](#_Toc398113784)

[5.4 Quick sort 29](#_Toc398113785)

[5.5 Merge sort 30](#_Toc398113786)

[6 Conclusión y evaluación 32](#_Toc398113787)

[7 Bibliografía 34](#_Toc398113788)

[8 Anexos 35](#_Toc398113789)

# Resumen

En este trabajo se investigaron cinco distintos algoritmos para compararlos, con el objetivo de determinar cuál era el algoritmo más rápido y cuál era el más lento al momento de ordenar datos numéricos basándose primordialmente en el tiempo que tomaban en ordenar distintas cantidades de datos numéricos dispuestos en distintos órdenes, además de comprobar si existía una relación entre el tiempo de ejecución con la cantidad de intercambios y comparaciones entre datos.

Para llevar a cabo este análisis, se desarrolló una aplicación en Java utilizando NetBeans, la cual consistía en que el usuario ingresaba la cantidad de datos a ordenar, luego este mismo especificaba en qué orden quería que esté esta cantidad, después escogía qué algoritmo quería usar para ordenar, y al final era mostrado en pantalla el tiempo que tomó el algoritmo especificado (en milisegundos), además de la cantidad de intercambios y comparaciones que se llevaron a cabo.

Utilizando este programa, se llevó a cabo las pruebas necesarias, para posteriormente analizar los datos obtenidos. Por medio de estos datos se comparó el rendimiento de los distintos algoritmos en las diferentes opciones dadas para así determinar puntos fuertes y débiles de cada uno para llegar a una conclusión al final.

Se concluyó que entre todos los algoritmos, el Quicksort tuvo un rendimiento general mucho mejor que los demás, es decir que su tiempo de ejecución mucho menor que todos, y que el algoritmo Bubblesort fue el que peor rendimiento tuvo en todos los casos, es decir mayor tiempo de ejecución. Además se comprobó que sí existía una fuerte relación entre el número de comparaciones e intercambios con el tiempo de ejecución, en donde mientras menos comparaciones e intercambios existían menor sería el tiempo tomado.

Conteo de palabras: 284 palabras

# 

# Agradecimiento

Agradezco a mis padres, ya que fue gracias a ellos que logré cumplir con este trabajo. A todos mis amigos que me estuvieron impulsando e incentivando y en especial a Valeria Rivera, quien fue la persona que me incentivó mas.

También agradezco a mi tutor, Enrique Guevara, quien fue el que me guió durante este trabajo.

# Introducción

Hoy en día, vivimos en un mundo en donde las personas, casi por naturaleza y por la exigencia del avance de nuestra sociedad, tratamos de simplificar nuestras tareas en varios aspectos, uno de estos es el tiempo, siendo un factor muy importante. Siempre buscamos la forma de reducir los intervalos e incrementar el número de acciones, hacer más en menos. Este elemento es decisivo tanto como para las grandes compañías como para la vida cotidiana de cualquier persona. Nuestro entorno cada vez acelera más, y nosotros debemos acelerar también, y esto es algo que aplica en todas las áreas, como en la producción de materiales, cantidades de ventas, ordenamiento, viajes, etc. Básicamente tratamos de sacarle provecho a toda acción que realizamos, siempre queremos completarla en el menos tiempo posible.

Los seres humanos hemos creado varias herramientas que nos ayudan al cumplimiento de este objetivo. Una de estas es la computación, ha sido la herramienta, que con el paso del tiempo, se ha vuelto la más predominante de todas. La computación o informática, comenzó de forma muy rústica, ya que solo facilitaban trabajos repetitivos, pero aun así fue de gran ayuda y mejoró varios aspectos en cuanto a productividad de aquellos tiempos. Desde sus inicios hasta el día de hoy, esta ciencia ha ido cambiando, mejorado e incrementando su eficiencia, es decir que ha ido evolucionando a base de las necesidades humanas. Para lograr esto es necesario desechar los métodos o procesos obsoletos.

En la computación estos procesos se conocen como algoritmos. Se componen de una serie de instrucciones lógicas con el fin de completar una acción. Existen distintos tipos de algoritmos para diferentes tareas a realizar, como los algoritmos de búsqueda que sirven para determinar la forma en la que se localizan los datos deseados; o los algoritmos de encriptación, que sirven para proteger datos mediantes claves, por ejemplo. Pero, en esta investigación, me enfocaré en los algoritmos de ordenamiento, los cuales serán definidos más adelante.

Los usos de estos algoritmos van más allá de simplemente ordenar datos en una lista, sus usos son variados y pueden estar dentro de procesos más complejos, en donde el ordenamiento es solo una pequeña parte del trabajo, pero que puede ser muy importante. Dentro del área de los algoritmos de ordenamiento, tenemos varios tipos de algoritmos, cada uno diferente entre sí, con sus características que definen su rapidez al momento de ordenar datos.

Los algoritmos a tratar son:

Bubblesort

Selection sort

Insertion sort

Quick sort

Merge sort

El objetivo de esta investigación y experimentación, será determinar cuál de estos algoritmos mencionados es el más rápido y cuál es el más lento para el ordenamiento de distintas listas de datos numéricos que están dispuestos de forma descendente, ascendente y aleatoria, para así determinar su eficacia en el cumplimiento del trabajo asignado, basándome esencialmente en el tiempo de ejecución, pero también relacionando con la cantidad de intercambios y comparaciones realizadas durante el proceso de cada algoritmo.

# Marco teórico

En esta sección procederé a explicar términos importantes para el desarrollo de este trabajo de investigación

## Algoritmo

Cormen, Leiserson, Rivest, y Clifford (2000)[[1]](#footnote-1) se refieren a los algoritmos de una manera informal como un procedimiento computacional bien definido que toma un valor, o conjunto de valores, como una entrada (*input)* y produce un valor, o conjunto de valores, como una salida (*output*). Por lo tanto, un algoritmo es una secuencia de pasos computacionales que transforma la entra en la salida.

También podemos ver un algoritmo como una herramienta para resolver un problema computacional bien especificado. La declaración del problema especifica en términos generales la relación deseada entre entrada y salida. Pero no todo problema resuelto por algoritmos tiene identificado fácilmente un grupo de posibles soluciones.

Según Joyannes Aguilar (2003)[[2]](#footnote-2) un algoritmo debe especificar tres partes: Entrada, Proceso y salida. Además las características fundamentales con las que debe cumplir un algoritmo son:

* Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.
* Un algoritmo debe estar definido. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
* Un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

## Ordenamiento

La ordenación o clasificación de datos, es un proceso que se encarga de arreglar un conjunto de datos en base a un criterio de ordenamiento, por lo general de forma ascendente o descendente, con respecto a uno de los campos de elementos del conjunto de acuerdo a Joyannes Aguilar (2003)[[3]](#footnote-3).

Existen distintos tipos de algoritmos de ordenamiento, cada uno con sus fortalezas y debilidades particulares. La mayoría de los algoritmos de ordenamiento trabajan comparando los datos que están siendo ordenados. En algunos casos, es preferible ordenar una cantidad de datos enorme basado solamente en una porción de esos datos. La pieza de datos realmente utilizados para determinar el orden de clasificación se llama la llave. (Alex Allain, 2011)[[4]](#footnote-4)

Hay dos tipos de ordenamiento:

* El ordenamiento interno es cuando el número de registros a ser ordenado es lo suficientemente pequeño que todo el proceso puede ser realizado en la memoria principal del computador. (Knuth, D. 1997) [[5]](#footnote-5)
* El ordenamiento externo es cuando la información que se tiene a ordenar no cabe en la memoria principal del computador por lo que se utiliza un tipo de memoria más lento durante el proceso. (Knuth, D. 1997)

En esta monografía todos los algoritmos a utilizar, serán algoritmos comparativos (Bubblesort, Selectionsort, Insertionsort, Mergesort y Quicksort), es decir que utilizan un operador comparativo para determinar cuál de los elementos que están siendo comparados deber ir antes o después en la lista final ordenada. De los cuales dos de ellos son recursivos (Quicksort y Mergesort). Para analizar sus diferencias en desempeño.

## Efectividad

Según Joyannes Aguilar (2003), para saber cuáles son los mejores métodos de ordenación, entre numerosas opciones existentes, nos basamos en su eficiencia. Este es el factor que mida la calidad y rendimiento de un algoritmo. En este caso de ordenación, existen dos criterios que son los más comunes para seguir al momento de determinar la eficiencia de algoritmos de ordenamiento: el tiempo menor de ejecución en computadora y el menor número de instrucciones, o comparaciones en el aspecto de ordenamiento. Es decir que si el algoritmo de ordenación A requiere de menos tiempo de ejecución y menor número de comparaciones que el algoritmo B, entonces A es más eficiente.

En computación, para clasificar algoritmos se utiliza la notación big-O. Esta notación describe el comportamiento limitante de una función cuando el argumento tiende a un valor particular o al infinito, por lo general se representan en funciones más simples [[6]](#footnote-6), como por ejemplo:

* O(1) = constante
* O(log(n)) = logarítmica
* O(n) = linear
* O(n2) = cuadrática

Según Jesús Gonzales (2012), El tiempo que requiere un algoritmo para dar una respuesta, se divide generalmente en 3 casos:

* Peor Caso: caso más extremo, donde se considera el tiempo máximo para solucionar un problema
* Caso promedio: caso en el cual, bajo ciertas restricciones, se realiza un análisis del algoritmo
* Mejor caso: caso ideal en el cual el algoritmo tomará el menor tiempo para dar una respuesta [[7]](#footnote-7)

La mayoría de los algoritmos utilizan una eficiencia algorítmica de O(n2) o O(n\*log(n)). Muchos algoritmos que tienen la misma eficiencia no tienen la misma velocidad de ejecución en una igual entrada. Primero, los algoritmos deben ser juzgados basándose en la eficiencia de su caso promedio, peor caso, y mejor caso. Algunos algoritmos tienen un mejor desempeño para una entrada, mientras que otro se desempeña pésimo para la misma entrada. [[8]](#footnote-8)

## Uso de recursos (hardware)

Según Orcola (2004), una forma de analizar la eficiencia de un algoritmo es determinar la cantidad de recursos informáticos que el algoritmo consume. Los recursos que generalmente se van a tomar en cuenta son la cantidad de memoria y la cantidad de tiempo de cómputo requeridos por el algoritmo. Y Orcola define a la memoria consumida por un algoritmo como “la suma de dos componentes: una parte fija que es independiente de sus datos y resultados, y una parte variable que depende del tamaño del problema considerado”[[9]](#footnote-9)

El espacio requerido por un algoritmo varía dependiendo de la estructura de dicho algoritmo. Dependiendo de la cantidad de variables a usar influirá en cuanta memoria se utilizará, algunos algoritmos necesitaran mayor memoria además de el de las variables, e incluso existen algoritmos en donde nunca se utilizara espacio extra.

Entonces basándose en estos datos, se tiene que el tipo de hardware en el cual correrá el algoritmo influye mucho en el tiempo de ejecución de estos procesos. Pero esta diferencia en el tiempo de ejecución variará en la comparación de distintos computadores, mientras que la diferencia entre los algoritmos se mantendrá ya que esta diferencia viene dada por la estructura del algoritmo y sus características particulares que se de diferencian de otros algoritmos.

## Explicación breve de los algoritmos a usar

En esta sección explicaré de forma breve los algoritmos que utilizaré en las pruebas y análisis para así tener una mejor comprensión de estos

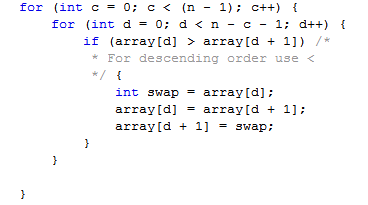
### Algoritmo Bubble sort

El primer algoritmo que muchas personas aprenden es el Bubble sort, debido a su simpleza y fácil entendimiento. Es sencillo de implementar, pero al ser una de las primeras formas de ordenar datos, es considerado de los más lentos al momento de ordenar, es por eso que es recomendable utilizar este algoritmo para cantidades de datos relativamente pequeños. (Robert Sedgewick, 2002)[[10]](#footnote-10)

La idea de este algoritmo consiste en comparar elementos por pares que sean adyacentes e intercambiarlos entre sí, en el caso de ser necesarios, hasta que todos se encuentren ordenados.

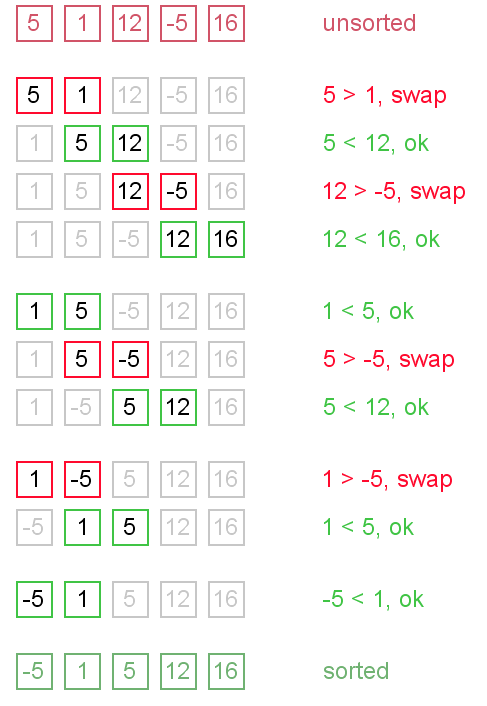
Su complejidad en su mejor caso, cuando los datos ya están ordenados, es O(n), mientras que tanto en su peor caso como en su caso promedio su complejidad viene dada por O(n2)

En la Figura 1 vemos el código del Bubble sort en Java.



**Figura 1** Bubble sort código en Java.

En la Figura 2 veremos un ejemplo paso a paso de este algoritmo



**Figura 2** ejemplo de Bubble sort

Fuente: http://www.algolist.net/Algorithms/Sorting/Bubble\_sort

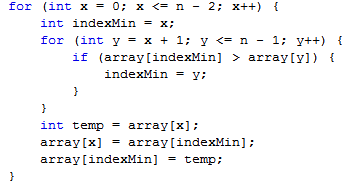
### Algoritmo Selection sort

Considerado uno de los algoritmos de ordenamiento más simple. Este método es llamado “selection sort” porque trabaja seleccionando repetidamente el elemento restante más pequeño. Funciona de la siguiente forma:

Primero, encontrar el elemento más pequeño en el arreglo, e intercambiarlo con el elemento en primera posición. Luego, encontrar el segundo elemento más pequeño e intercambiarlo con el elemento de la segunda posición. Continuar de esta forma hasta que el arreglo completo este ordenada. (Robert Sedgewick, 2002)[[11]](#footnote-11)

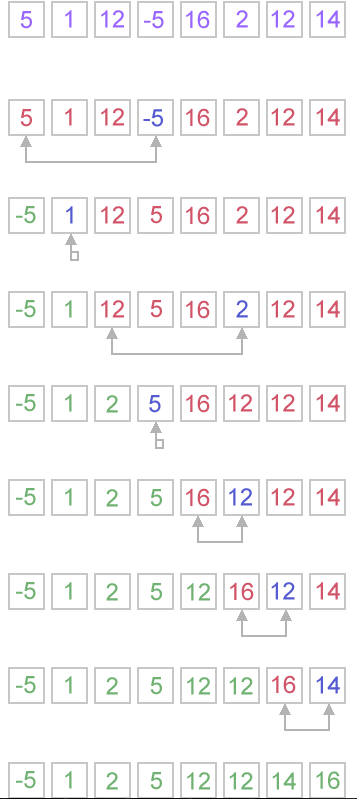
Su orden de complejidad viene dado por O(n2) tanto para su mejor caso, pero caso y caso promedio. Por lo que es recomendable usarlo ante cantidades relativamente pequeñas

En la Figura 3 vemos el código del Selection sort en Java.



**Figura 3** Selection sort código en Java

En la Figura 4 veremos un ejemplo paso a paso de este algoritmo



**Figura 4** ejemplo de Selection sort

Fuente: http://www.algolist.net/Algorithms/Sorting/Selection\_sort

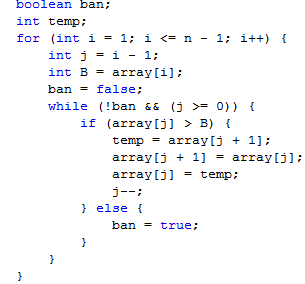
### Algoritmo Insertion sort

Este método consiste en: hacer espacio para el elemento que será insertado moviendo elementos más grandes una posición a la derecha, y luego se inserta el elemento en la posición que queda libre.

Al igual que en el selection sort, los elementos a la izquierda del índice actual están ordenados durante el proceso, pero no están en su posición final, debido a que es posible que tengan que moverse para hacer espacio a elementos más pequeños encontrados después. (Robert Sedgewick, 2002)[[12]](#footnote-12)

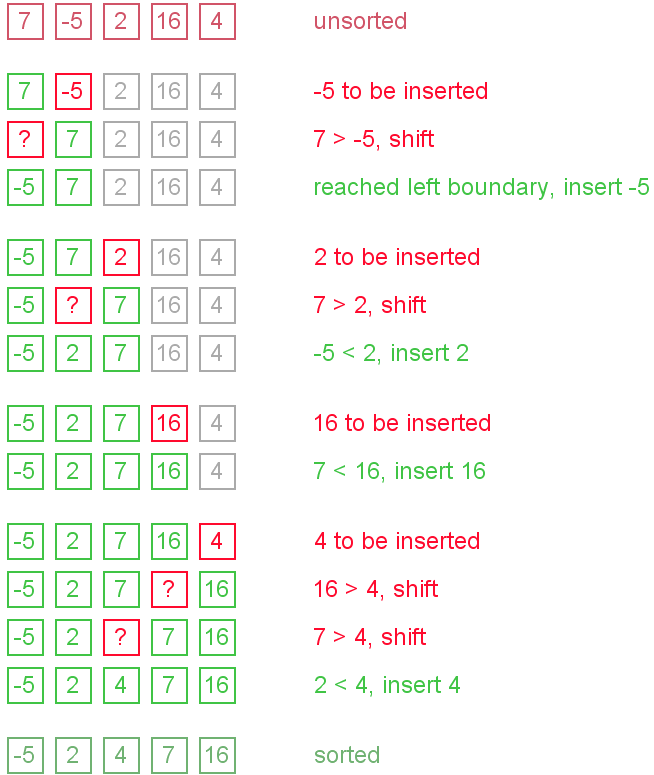
En el mejor caso de este algoritmo, que es cuando los datos están ya ordenados, su complejidad viene dada por O(n). En su peor caso más simple, que es cuando los están en orden invertidos, su complejidad es de O(n2), el cual también aplica para su caso promedio.

En la Figura 5 vemos el código del Insertion sort en Java.



**Figura 5** Insertion sort código en Java

En la Figura 6 veremos un ejemplo paso a paso de este algoritmo



**Figura 6** ejemplo de Insertion sort

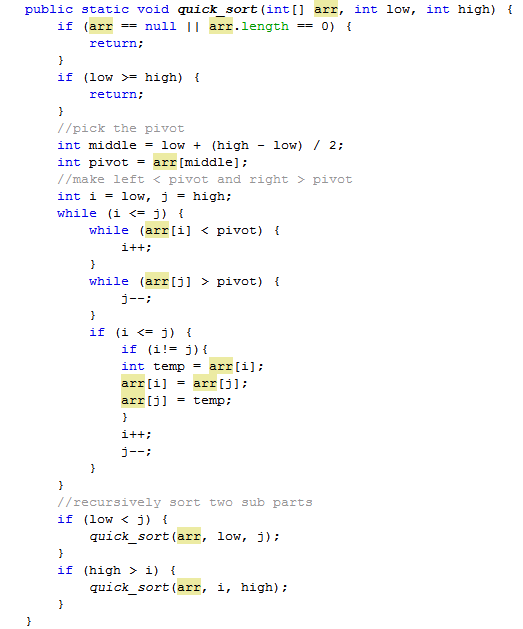
Fuente: http://www.algolist.net/Algorithms/Sorting/Insertion\_sort

### Algoritmo Quick sort

Quick sort es un método de que se basa en el paradigma de “dividir y conquistar” para ordenar. Funciona particionado un arreglo en dos partes, luego ordenando esas partes independientemente. La posición precisa de la partición depende del orden inicial de los elementos en la entrada. La esencia de este método es su proceso de partición y el escoger un pivote. (Robert Sedgewick, 2002)[[13]](#footnote-13)

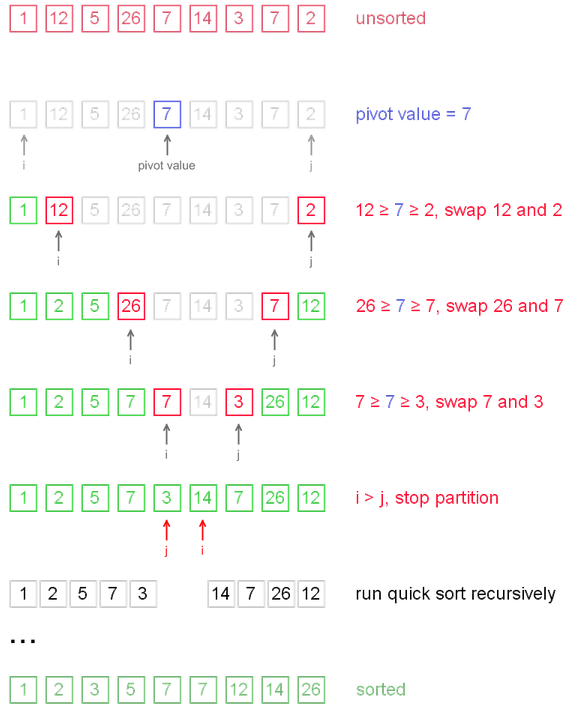
Tanto en su mejor caso como en su caso promedio, su complejidad viene dada por O(n log n), lo cual lo hace efectivo ante cantidades de datos relativamente grandes. Pero en su peor caso su complejidad viene dada por O(n2).

En la Figura 7 vemos el código del Quick sort en Java.



**Figura 7** Quick sort código en Java

En la Figura 8 veremos un ejemplo paso a paso de este algoritmo



**Figura 8** ejemplo de Quick sort

Fuente: http://www.algolist.net/Algorithms/Sorting/Quicksort

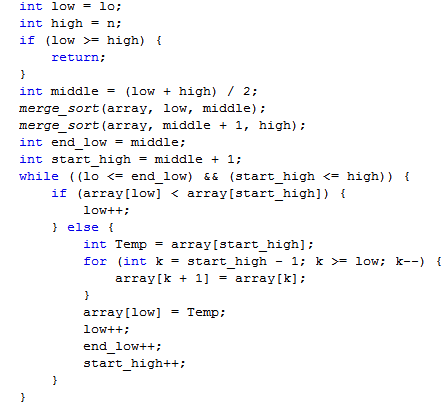
### Algoritmo Merge sort

Este algoritmo, al igual que Quick sort, se basa en el paradigma de “dividir y conquistar”. Para ordenar un arreglo dado, este método divide el arreglo en dos, luego recursivamente ordena ambas mitades independientemente, y después mezcla ambas mitades ya ordenadas para así obtener el arreglo ordenado final. Al mencionar que recursivamente ordena cada mitad implica que ese pedazo del arreglo lo volverá a dividir a la mitad y así sucesivamente hasta no poder dividirlo más.

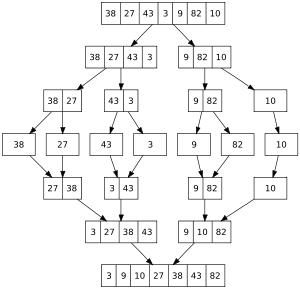
Una de sus características más atractivas es que para ordenar un arreglo de N elementos requiere un tiempo proporcional a n log n, sin importar el orden dado del arreglo. (Robert Sedgewick, 2002)[[14]](#footnote-14)

Para este algoritmo, tanto en su mejor caso, peor caso y caso promedio, su complejidad es dada por O(n log n). Es decir que será efectivo ante cantidades grandes de dato a ordenar.

En la Figura 9 vemos el código del Merge sort en Java.

  
**Figura 9** Merge sort código en Java

En la Figura 10 veremos un ejemplo paso a paso de este algoritmo



**Figura 10** ejemplo de Merge sort

Fuente: https://wiki.dcc.uchile.cl/alice/lib/exe/fetch.php?cache=&media=300px-merge\_sort\_algorithm\_diagram.png

# Metodología

En esta sección describiré de forma detallada de qué forma desarrollaré el experimento y llevaré a cabo las pruebas

## Diseño del experimento

Debido a que para cumplir el objetivo de esta experimentación me basaré principalmente en medir el tiempo de ejecución de cada algoritmo, y como sabemos que la cantidad de datos tiene una relación directamente proporcional con el tiempo, haré distintas pruebas con diferentes longitudes de arreglos de datos numéricos (500, 1000, 5000, 10000, 50000, 100000). Cada arreglo estará dispuesto de 3 formas diferentes (ascendente, descendente y aleatoria). Por lo que tendré el tiempo como una variable dependiente y como variable independiente el tamaño de la entrada o cantidad de datos a ordenar.

Para cada algoritmo, realizaré diez pruebas en cada longitud de datos numéricos, y cada una de estas longitudes será ordenada ascendente, descendente y aleatoriamente. Luego sacaré un promedio de los resultados de las diez pruebas de cada arreglo y esos serán los datos que utilizare al momento de analizar la experimentación.

Para llevar a cabo este experimento diseñaré una aplicación en Java utilizando el NetBeans IDE 7.2, el cual consistirá en poner cada algoritmo de ordenación en distintos métodos independientes. Se encargará también de crear los arreglos necesarios para el experimento. Además medirá el tiempo de ejecución así como también la cantidad de comparaciones e intercambios que habrá durante cada ordenamiento.

El arreglo ascendente tendrá un orden desde el uno hasta el número especificado subiendo de uno en uno, el arreglo descendente tendrá un orden desde el número especificado hasta el uno bajando de uno en uno. Ambos arreglos utilizaran cada número comprendido. El arreglo aleatorio será tal que una vez generado, mantendrá el mismo orden sin importar cuantas veces ejecute el programa, es decir que sí creo un arreglo de 500 datos, este mantendrá el mismo orden por más de que ejecute varias veces el programa. Este arreglo también mantendrá un mismo patrón sin importar la su longitud. Esto lo realizo con el fin de que, al tener un mismo patrón de orden, no exista ninguna diferencia entre las pruebas de los distintos algoritmos que puedan “favorecer” o “empeorar” los resultados. Además este arreglo presentará algunos números repetidos en su orden, para así ver como se desempeña el algoritmo probado ante este tipo de casos en donde se tiene un mismo dato repetido en varias ocasiones. Entonces, aunque no necesariamente se utilicen cada uno de los números comprendidos en la cantidad especificada del arreglo, la cantidad de datos a ordenar si será la especificada.

Para mayor información de cuál será el código para generar estos arreglos aleatorios revisar el anexo 1.

Debido a que las cantidades de datos que usaré serán grandes, mediré el tiempo en milisegundos utilizando el método *System.currentTimeMillis()*. De esta forma tendré valores enteros comparables. El tiempo será medido exactamente desde el inicio hasta el término de la ejecución exclusivamente de dicho algoritmo de ordenamiento. Además el programa mostrará la cantidad de comparaciones e intercambios entre datos con el fin de observar una posible relación entre estas cantidades y el tiempo.

Para mayor información sobre el programa que crearé y utilizaré consultar el anexo 2.

## Aspectos a tomar en cuenta

Hay que tomar en cuenta que las características técnicas del computador utilizado influenciarán de forma muy obvia el tiempo de ejecución, pero no la cantidad de comparaciones e intercambios. Es decir que un computador con procesador de alto rendimiento y gran cantidad de memoria RAM, ejecutará cada algoritmo en menos tiempo, y lo contrario para un computador de bajo rendimiento y poca cantidad de memoria RAM.

Pero hay que dejar en claro el hecho de que a pesar de que los tiempos de ejecución comparados entre computadoras con diferentes detalles técnicos serán diferentes, la diferencia entre cada algoritmo no se verá afectada, esto debido a que esta diferencia es totalmente dependiente de la estructura del algoritmo, eso es lo que define sus diferencias, el tipo que de estructura que tiene cada algoritmo es lo que definirá su desempeño al momento de ordenar.

Este experimento se desarrolló bajo el siguiente ambiente:

* Intel Core i5 650 3.20 Ghz
* 8 GB de RAM
* HDD 1 TB
* AMD Radeon HD 6970 2GB VRAM
* Windows 7 Home Premium 64 bits
* NetBeans IDE 7.2

# Pruebas

En esta sección se realizarán las pruebas de tiempo de ejecución de cada algoritmo en los distintos órdenes dispuestos para determinar su rendimiento, además se contarán el número de intercambios y comparaciones.

## Bubble sort

En la Tabla 1 (ver anexo 3.1) vemos los resultados del rendimiento de este algoritmo en el orden ascendente.

En la Tabla 2 (ver anexo 3.2) vemos los resultados del rendimiento de este algoritmo en el orden descendente

En la Tabla 3 (ver anexo 3.3) vemos los resultados del rendimiento de este algoritmo en el orden aleatorio

En la tabla 4 vemos resumido el rendimiento del Bubble sort en los distintos órdenes dispuestos

|  |  |  |  |
| --- | --- | --- | --- |
| **Bubblesort** | | | |
| Número de datos a ordenar | Tiempo en milisegundos Ascendente | Tiempo en milisegundos Descendente | Tiempo en milisegundos  Aleatorio |
| 500 | **3** | **3** | **4** |
| 1000 | **4** | **5** | **6** |
| 5000 | **30** | **34** | **61** |
| 10000 | **109** | **126** | **233** |
| 50000 | **2670** | **3085** | **5743** |
| 100000 | **10790** | **12498** | **22647** |

**Tabla 4**. Tiempo de ejecución del Bubble sort en los distintos órdenes

## Selection sort

En la Tabla 5 (ver anexo 3.4) vemos los resultados del rendimiento de este algoritmo en el orden ascendente.

En la Tabla 6 (ver anexo 3.5) vemos los resultados del rendimiento de este algoritmo en el orden descendente

En la Tabla 7 (ver anexo 3.6) vemos los resultados del rendimiento de este algoritmo en el orden aleatorio

En la tabla 8 vemos resumido el rendimiento del Selection sort en los distintos órdenes dispuestos

|  |  |  |  |
| --- | --- | --- | --- |
| **Slectionsort** | | | |
| Número de datos a ordenar | Tiempo en milisegundos Ascendente | Tiempo en milisegundos Descendente | Tiempo en milisegundos  Aleatorio |
| 500 | **3** | **3** | **3** |
| 1000 | **4** | **5** | **4** |
| 5000 | **29** | **27** | **30** |
| 10000 | **108** | **95** | **109** |
| 50000 | **2722** | **2343** | **2710** |
| 100000 | **10745** | **9418** | **10752** |

**Tabla 8**. Tiempo de ejecución del Selection sort en los distintos órdenes

## Insertion sort

En la Tabla 9 (ver anexo 3.7) vemos los resultados del rendimiento de este algoritmo en el orden ascendente.

En la Tabla 10 (ver anexo 3.8) vemos los resultados del rendimiento de este algoritmo en el orden descendente

En la Tabla 11 (ver anexo 3.9) vemos los resultados del rendimiento de este algoritmo en el orden aleatorio

En la tabla 12 vemos resumido el rendimiento del Insertion sort en los distintos órdenes dispuestos

|  |  |  |  |
| --- | --- | --- | --- |
| **Insertionsort** | | | |
| Número de datos a ordenar | Tiempo en milisegundos Ascendente | Tiempo en milisegundos Descendente | Tiempo en milisegundos  Aleatorio |
| 500 | **0** | **7** | **5** |
| 1000 | **0** | **7** | **7** |
| 5000 | **0** | **22** | **15** |
| 10000 | **1** | **69** | **38** |
| 50000 | **3** | **1622** | **827** |
| 100000 | **5** | **6521** | **3305** |

**Tabla 12**. Tiempo de ejecución del Insertion sort en los distintos órdenes

## Quick sort

En la Tabla 13 (ver anexo 3.10) vemos los resultados del rendimiento de este algoritmo en el orden ascendente.

En la Tabla 14 (ver anexo 3.11) vemos los resultados del rendimiento de este algoritmo en el orden descendente

En la Tabla 15 (ver anexo 3.12) vemos los resultados del rendimiento de este algoritmo en el orden aleatorio

En la tabla 16 vemos resumido el rendimiento del Quick sort en los distintos órdenes dispuestos

|  |  |  |  |
| --- | --- | --- | --- |
| **Quicksort** | | | |
| Número de datos a ordenar | Tiempo en milisegundos Ascendente | Tiempo en milisegundos Descendente | Tiempo en milisegundos  Aleatorio |
| 500 | **0** | **0** | **0** |
| 1000 | **0** | **0** | **1** |
| 5000 | **4** | **4** | **5** |
| 10000 | **9** | **5** | **7** |
| 50000 | **22** | **19** | **21** |
| 100000 | **25** | **21** | **23** |

**Tabla 16**. Tiempo de ejecución del Quick sort en los distintos órdenes

## Merge sort

En la Tabla 17 (ver anexo 3.13) vemos los resultados del rendimiento de este algoritmo en el orden ascendente.

En la Tabla 18 (ver anexo 3.14) vemos los resultados del rendimiento de este algoritmo en el orden descendente

En la Tabla 19 (ver anexo 3.15) vemos los resultados del rendimiento de este algoritmo en el orden aleatorio

En la tabla 20 vemos resumido el rendimiento del Merge sort en los distintos órdenes dispuestos

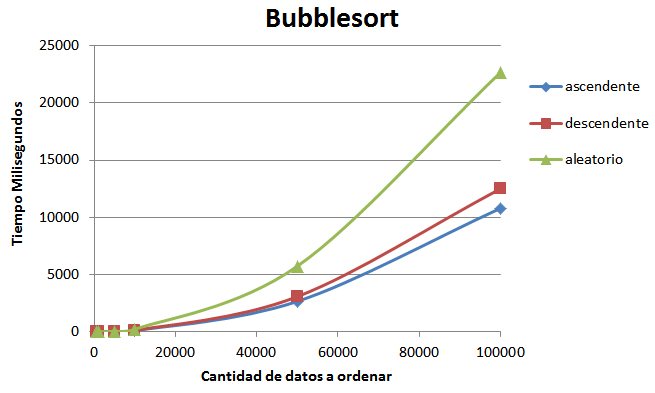
|  |  |  |  |
| --- | --- | --- | --- |
| **Mergesort** | | | |
| Número de datos a ordenar | Tiempo en milisegundos Ascendente | Tiempo en milisegundos Descendente | Tiempo en milisegundos  Aleatorio |
| 500 | **0** | **6** | **4** |
| 1000 | **1** | **10** | **7** |
| 5000 | **8** | **25** | **25** |
| 10000 | **10** | **58** | **49** |
| 50000 | **14** | **1250** | **820** |
| 100000 | **17** | **5141** | **3135** |

**Tabla 20**. Tiempo de ejecución del Merge sort en los distintos órdenes

# Análisis

En esta sección analizaré cada algoritmo basándome en los resultados obtenidos en las pruebas y los iré comparando conforme vaya analizando.

## Bubble sort

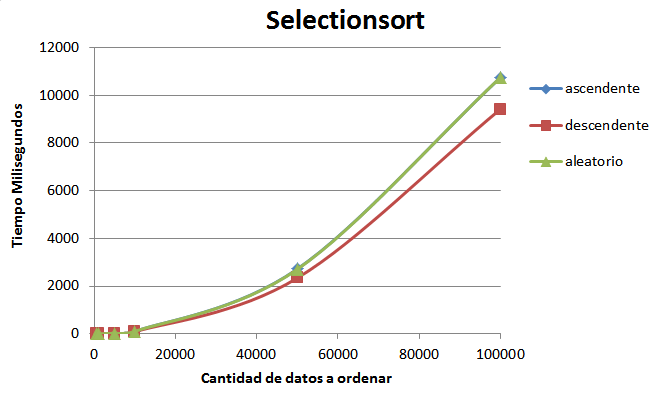


**Gráfica 1**. Se muestra en un diagrama de dispersión el rendimiento del algoritmo Bubblesort en los distintos órdenes y cantidades de datos dispuestos

En la Gráfica 1 vemos claramente que este algoritmo tiene una complejidad de O(n2) al momento de ordenar los datos, por lo que en cantidades más pequeñas de datos la diferencia del tiempo es relativamente pequeña mientras que con cantidades más grandes este tiempo crece exponencialmente. En la gráfica también podemos ver que el peor caso es cuando el arreglo está dispuesto en orden aleatoria, su mejor caso cuando el arreglo ya está ordenado (ascendente) y un caso promedio cuando se está dispuesto en orden descendente. La diferencia del rendimiento del algoritmo en su peor caso, es mucho mayor a las demás.

Como podemos ver en los anexos 3.1, 3.2 y 3.3 el número de comparaciones para este algoritmo depende totalmente de la cantidad de datos a ordenar, no importa el orden de este, siempre se dará la misma cantidad de comparaciones para un arreglo de cierta extensión, el crecimiento de este número de comparaciones es de orden cuadrático. Pero, a diferencia de las comparaciones, el número de intercambios si depende del orden dado, como podemos ver en todos los arreglos dados cuando estos estaban ya ordenados no se realizó ningún intercambio, mientras que, cuando el arreglo estaba en orden descendente, el número de intercambios fue el mismo que el de comparaciones, debido a que por su estructura, en este caso, estos intercambios se van realizando uno por uno por cada comparación.

## Selection sort

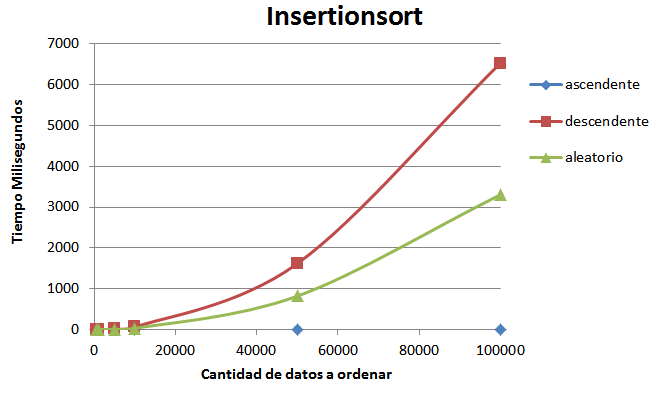


**Gráfica 2**. Se muestra en un diagrama de dispersión el rendimiento del algoritmo Selectionsort en los distintos órdenes y cantidades de datos dispuestos

En la Gráfica 2 vemos fácilmente que el orden de complejidad de este algoritmo, para todos sus casos, es de O(n2). Al igual que el Bubble sort, la diferencia del tiempo de ejecución, entre cantidades de datos cortos, es relativamente pequeña; mientras que, la diferencia del tiempo de ejecución, entre cantidades de datos largas, es mucho mayor. Como podemos ver en la gráfica, el rendimiento de este algoritmo en arreglos ascendentes y aleatorios es muy similar, casi el mismo, por lo que su peor caso y su caso promedio puede estar entre un arreglo aleatorio o uno dispuesto ascendentemente. Pero para este algoritmo su mejor caso es un arreglo ordenado descendentemente, y vemos que hay una pequeña diferencia en el tiempo de ejecución ante los demás casos.

Como podemos ver en los anexos 3.4, 3.5 y 3.6, vemos que al igual que el Bubble sort, este algoritmo realiza la misma cantidad de comparaciones sin importar el orden, pues esta depende de la cantidad de datos, pero, el Selection sort realiza una cantidad menor de intercambios que el Bubble sort, pero este número de intercambios es fijo, y tiene un orden de crecimiento lineal, es decir que sin importar el orden, siempre se realizará la misma cantidad intercambios para una cierta cantidad de datos a ordenar, pues la operación de *intercambiar* en este algoritmo se ejecutará siempre. Esta es la razón por la cual el tiempo de ejecución en todos los casos es muy similar.

## Insertion sort

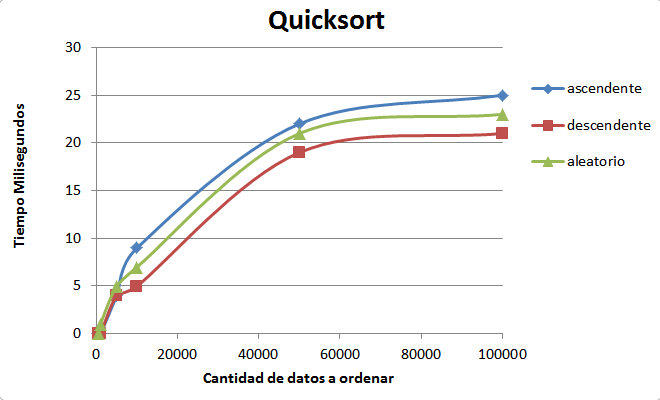


**Gráfica 3**. Se muestra en un diagrama de dispersión el rendimiento del algoritmo Insertionsort en los distintos órdenes y cantidades de datos dispuestos

En la Gráfica 3, vemos que es muy clara la diferencia del rendimiento de este algoritmo en distintos órdenes. Como vemos para su peor caso, el cual es cuando el arreglo está en orden descendente, su complejidad es de O(n2) al igual que en su caso promedio, que es cuando el arreglo está en orden aleatorio. Pero aun así la diferencia de tiempo entre ambos casos es grande. Ahora para su mejor caso, que es cuando el arreglo está dispuesto en forma ascendente, su complejidad es de O(n), es decir que es lineal y cómo podemos ver en la gráfica, esta función está muy cercana a cero milisegundos.

Al ver los anexos 3.7, 3.8 y 3.9 podemos notar que a diferencia de los anteriores algoritmos, el número de comparaciones para este caso si depende del orden dispuesto de los datos a ordenar, al igual que el número de intercambios. Cuando el arreglo esta dispuesto de forma descendente, realiza la misma cantidad de intercambios y comparaciones que el Bubblesort. Pero aun así este algoritmo tiene un tiempo de ejecución menor que el de los dos anteriores algoritmos, esto gracias a su estructura puesto que realiza menos instrucciones innecesarias durante el ordenamiento.

## Quick sort

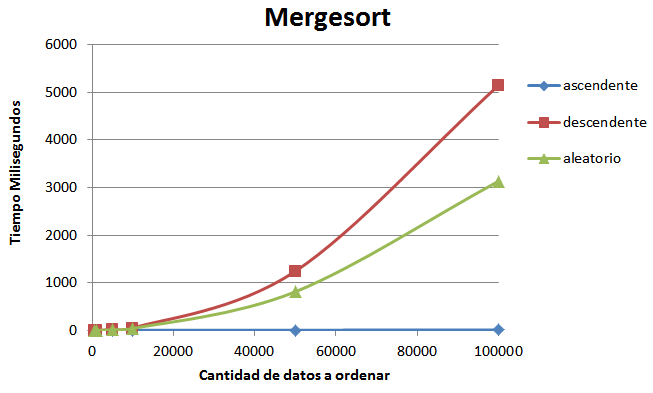


**Gráfica 4**. Se muestra en un diagrama de dispersión el rendimiento del algoritmo Quicksort en los distintos órdenes y cantidades de datos dispuestos

En la Gráfica 4 vemos que este algoritmo tiene un orden de complejidad de O(n log n) para todos sus casos, que es la razón por la cual el tiempo de ejecución es similar en altas cantidades de datos a ordenar. Para este algoritmo su peor caso es cuando el arreglo está dispuesto ascendentemente, esto debido a que depende principalmente del pivote. Su caso promedio es en un arreglo con datos aleatorios y para su mejor caso es un arreglo con datos en orden descendente. Como su tiempo de ejecución es de forma logarítmica, hace este algoritmo sea muy eficiente para ordenar cantidades de datos muy grandes, ya que la diferencia del tiempo de ejecución será relativamente pequeña.

Referenciándonos a los anexos 3.10, 3.11 y 3.12 vemos que al igual que el Insertion sort, el número de comparaciones y de intercambios del Quick sort sí depende el orden dado y la cantidad de datos a ordenar, pero estos valores son mucho menores y ambas operaciones tienen un crecimiento similar a una función lineal. Tanto el número de comparaciones como de intercambios, son mucho menor al del resto de los algoritmos, y es por esto que el tiempo de ejecución también es mucho menor al de los demás.

## Merge sort



**Gráfica 5**. Se muestra en un diagrama de dispersión el rendimiento del algoritmo Mergesort en los distintos órdenes y cantidades de datos dispuestos

Al ver la Gráfica 5 nos damos cuenta que es muy similar a la Grafica 3 (Insertion sort), es decir que el rendimiento es similar, pero el orden de complejidad para su mejor caso, que es cuando está ordenado, viene dado por O(n log n). A pesar de que se supone que este mismo orden de complejidad se aplicaba para su mejor y peor caso, en esta grafica vemos que en ambos casos su complejidad se asemeja a una cuadrática, es posible que esto sea porque esto se pueda notar gráficamente con cantidades más grandes de datos. Pero lo que si podemos ver sin duda es que su peor caso es un arreglo dispuesto en orden descendente y para su caso promedio es cuando el arreglo contiene datos aleatorios.

Al ver los anexos 3.13, 3.14 y 3.15 podemos notar que para este algoritmo, el número de comparaciones e intercambios dependen del orden y cantidad de datos dados a ordenar. A pesar de que en general el rendimiento del Insertion sort y el Merge sort son parecidos, se pueden establecer diferencias claras, una de estas es que cuando el arreglo está dispuesto de forma ascendente, a pesar de que ninguno realiza algún intercambio, el insertion sort realiza mucho menos comparaciones y es por esto que en este algoritmo tiempo un tiempo de ejecución menor en este orden. Ahora aunque ambos algoritmos realizan una cantidad de intercambios muy similar, casi la misma, cuando el arreglo está dispuesto en forma descendente y aleatoria, el Merge sort lleva a cabo menos comparaciones que el Insertion sort, y es por esto que Merge sort tiene un tiempo de ejecución menor que el insertion sort en arreglos dispuestos descendentemente y aleatoriamente.

# Conclusión y evaluación

El objetivo de esta investigación fue el de encontrar el algoritmo más rápido y el más lento en cuanto a tiempo de ejecución al ordenar datos numéricos y el de encontrar una relación entre el tiempo de ejecución con el número de comparaciones e intercambios.

De forma específica, para arreglos que estaban dispuestos de forma ascendente el algoritmo con menor tiempo de ejecución entre todos fue el Insertionsort puesto que al ordenar 100,000 datos numéricos tomó alrededor de 5 milisegundos con 0 intercambios y 99,999 comparaciones, estos tres valores de tiempo, intercambios y comparaciones, fueron los más bajos entre todos los algoritmos. El algoritmo que tomó mayor tiempo en ordenar arreglos ascendentes fue el Bubblesort con 10790 milisegundos 0 intercambios y 4,999,950,000 comparaciones para ordenar 100,000 datos numéricos.

Para arreglos dispuestos de forma descendente el algoritmo Quicksort fue el que tomó menos tiempo en ordenar entre todos los demás algoritmos, para 100,000 datos numéricos se demoró 21 milisegundos con 50,000 intercambios y 70,3926 comparaciones. Estos tres valores fueron los más bajos de toda la prueba. El algoritmo que tomó más tiempo en este tipo de arreglos fue el Bubblesort con 12,498 milisegundos y 4,999,950,000 comparaciones e intercambios para 100,000 datos.

Para arreglos dispuestos de forma aleatoria el algoritmo que tomó el menor tiempo de ejecución fue el Quicksort, puesto que para 100,000 datos numéricos tomó 23 milisegundos con 399,875 intercambios y 582074 comparaciones, los valores de tiempo y comparaciones fueron los más bajos de toda la prueba. El algoritmo que tomó mayor tiempo de ejecución en este orden fue el Bubblesort para 100,000 datos a ordenar demoró 22,647 milisegundos con 2,492,430,171 intercambios y 4,999,950,000 comparaciones.

En términos generales la clasificación de los algoritmos del más rápido al más lento es:

1. Quicksort
2. Mergesort
3. Insertionsort
4. Selectionsort
5. Bubblesort

Ahora basándonos en los datos obtenidos durante las pruebas, vemos que el número de intercambios y comparaciones es un factor determinante para el tiempo de ejecución, es decir que existe una relación directamente proporcional. Al ver los resultados vemos que en la mayoría de casos los menores tiempos de ejecución iban acompañados de los menores números de intercambios y comparaciones, lo mismo para los mayores tiempos de ejecución.

Por lo general los números de comparaciones e intercambios mayores iban acompañados de los tiempos de ejecución mayores y los números de comparaciones e intercambios menores iban acompañados de los tiempos de ejecución menores. Es decir que en muchos casos, basarnos en esos números nos ayudaban a comprender mejor el por qué un algoritmo tiene un tiempo de ejecución menor que otro algoritmo. Pero hubo pequeñas excepciones puesto que el tiempo de ejecución no depende del todo de cuantas comparaciones e intercambios realice, sino también de la estructura del algoritmo.

# Bibliografía

Allain, A. (2011). *Sorting Algorithm Comparison.* Obtenido de http://www.cprogramming.com: http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Clifford, S. (2000). *Introduction to Algorithms.* Cambridge: MA: The MIT Press.

González, J. A. (2012). *Análisis y Diseño de Algoritmos Notación Asintótica .* Obtenido de http://ccc.inaoep.mx/: http://ccc.inaoep.mx/~jagonzalez/ADA/NotacionA.pdf

Joyannes Aguilar, L. (2003). *Fundamentos de programación: Algoritmos, Estructuras de datos, y Objetos.* Madrid: McGraw-Hill/Interamericana de España, S.A.U.

Knuth, D. (1968). *The Art of Computer Programming.* Standford: Addison-Wesley.

Lagos, F. (2007). *Algoritmos de Ordenamiento*. Obtenido de http://blog.zerial.org/: http://blog.zerial.org/ficheros/Informe\_Ordenamiento.pdf

MIT Lecture. (2003). *Big O Notation.* Obtenido de http://web.mit.edu: http://web.mit.edu/16.070/www/lecture/big\_o.pdf

Orcola, C. (2004). *Gestión de datos.* Universidad Tecnológica Nacional.

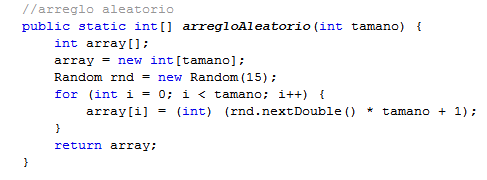
Sedgewick, R. (2002). *Algorithms in Java: Parts 1-4, Third Edition.* Addison Wesley.

*Sorting Algorithms Demonstration in Java*. (s.f.). Obtenido de http://home.westman.wave.ca/: http://home.westman.wave.ca/~rhenry/sort/

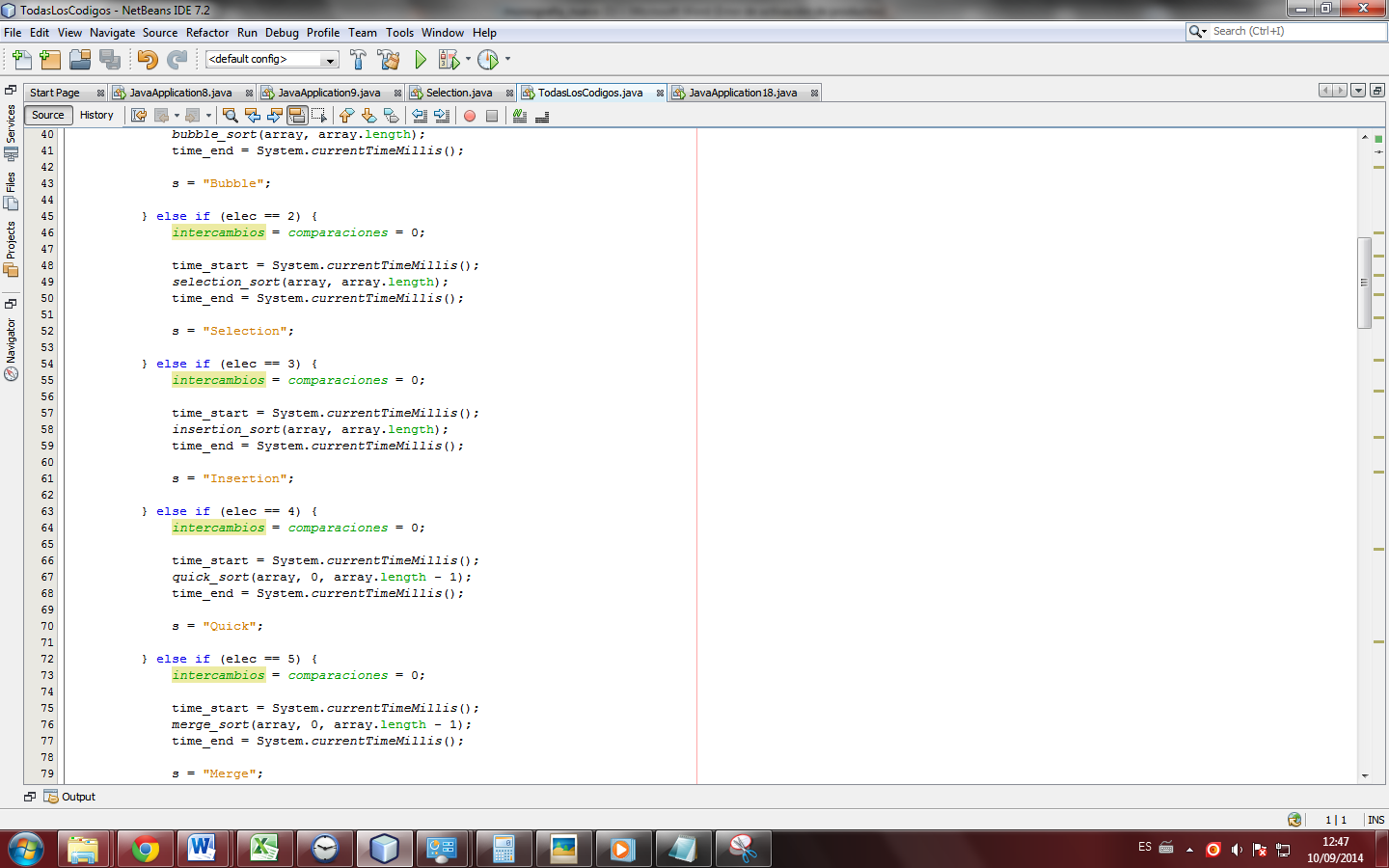
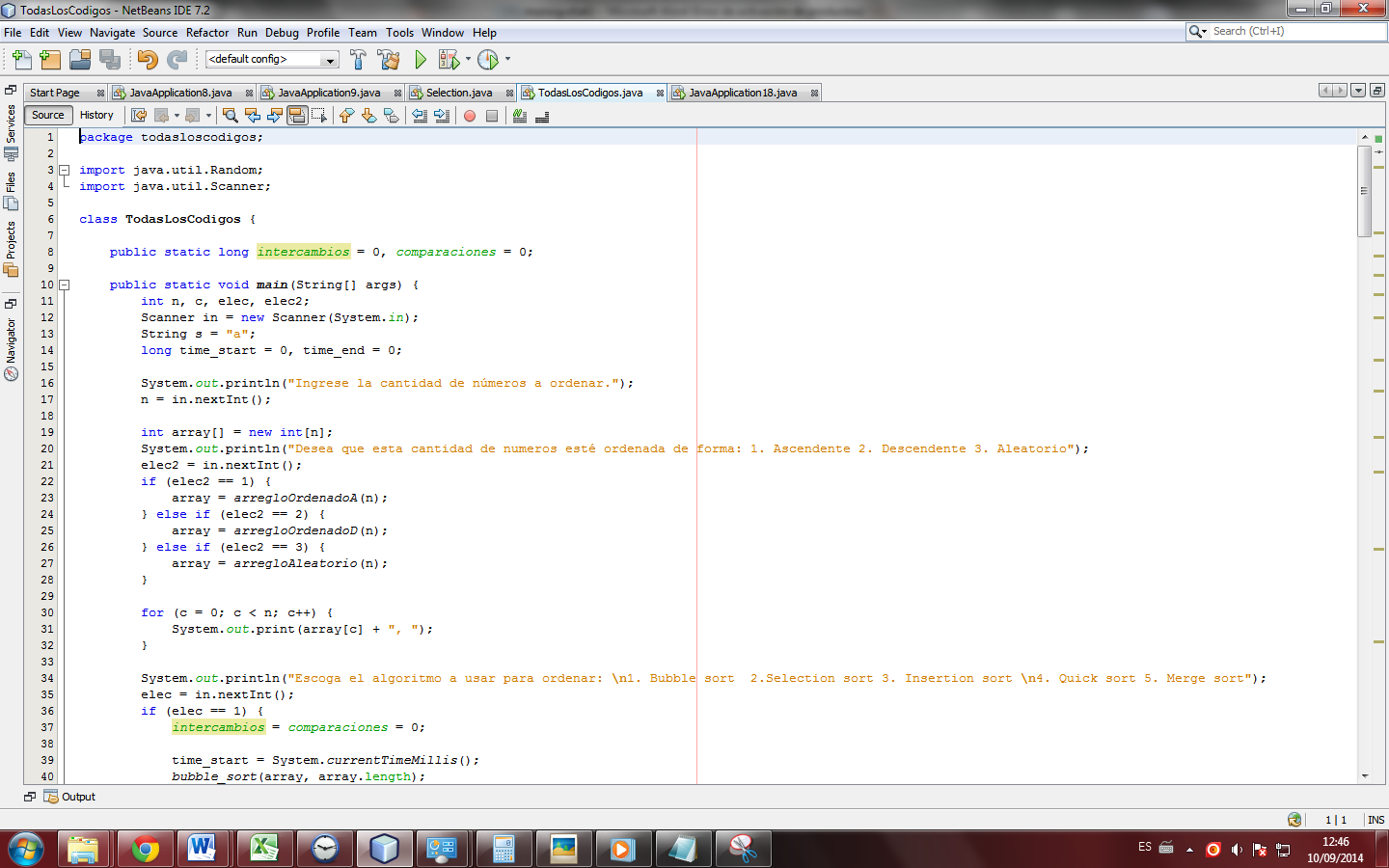
# Apéndice

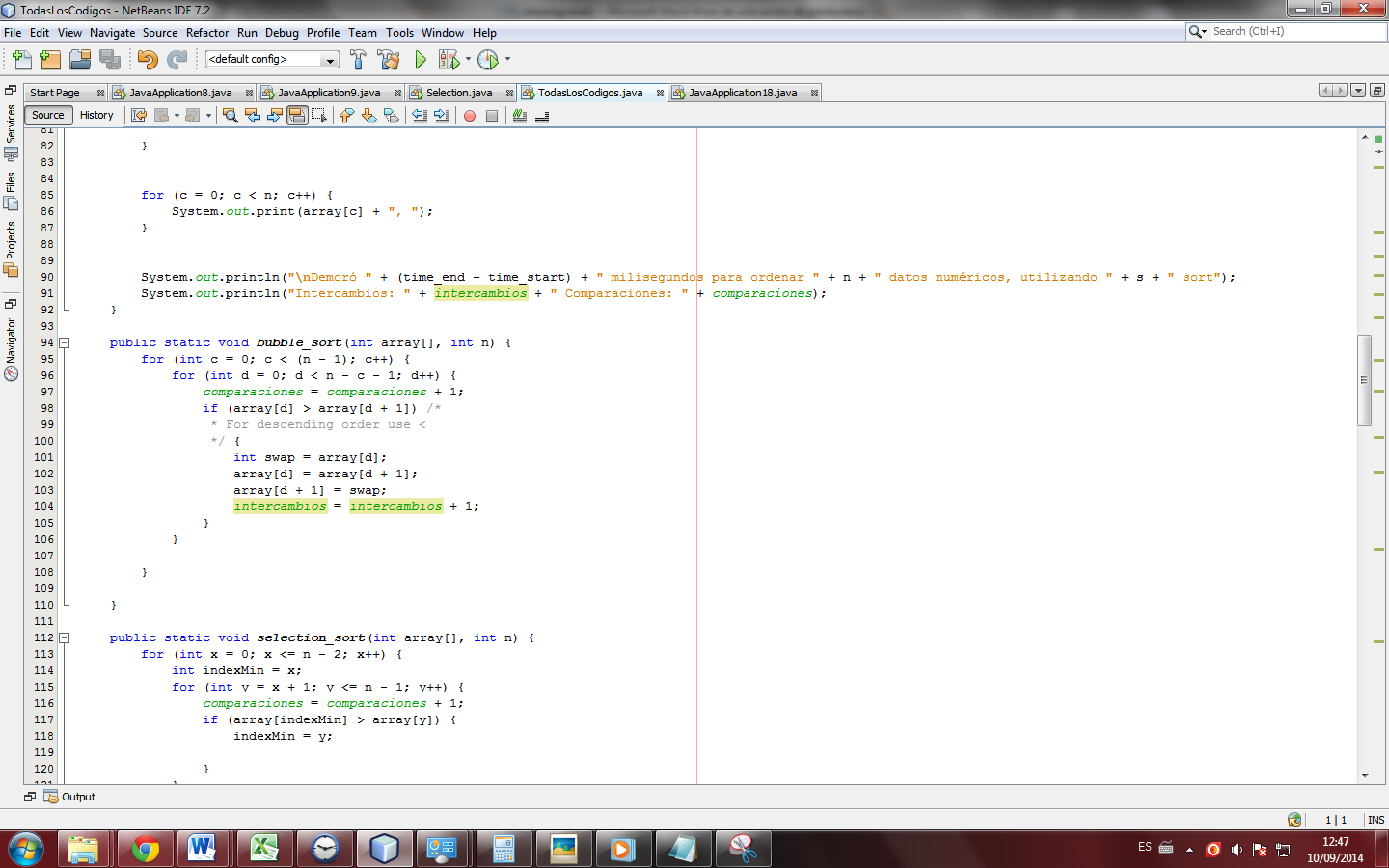
**Anexo 1**

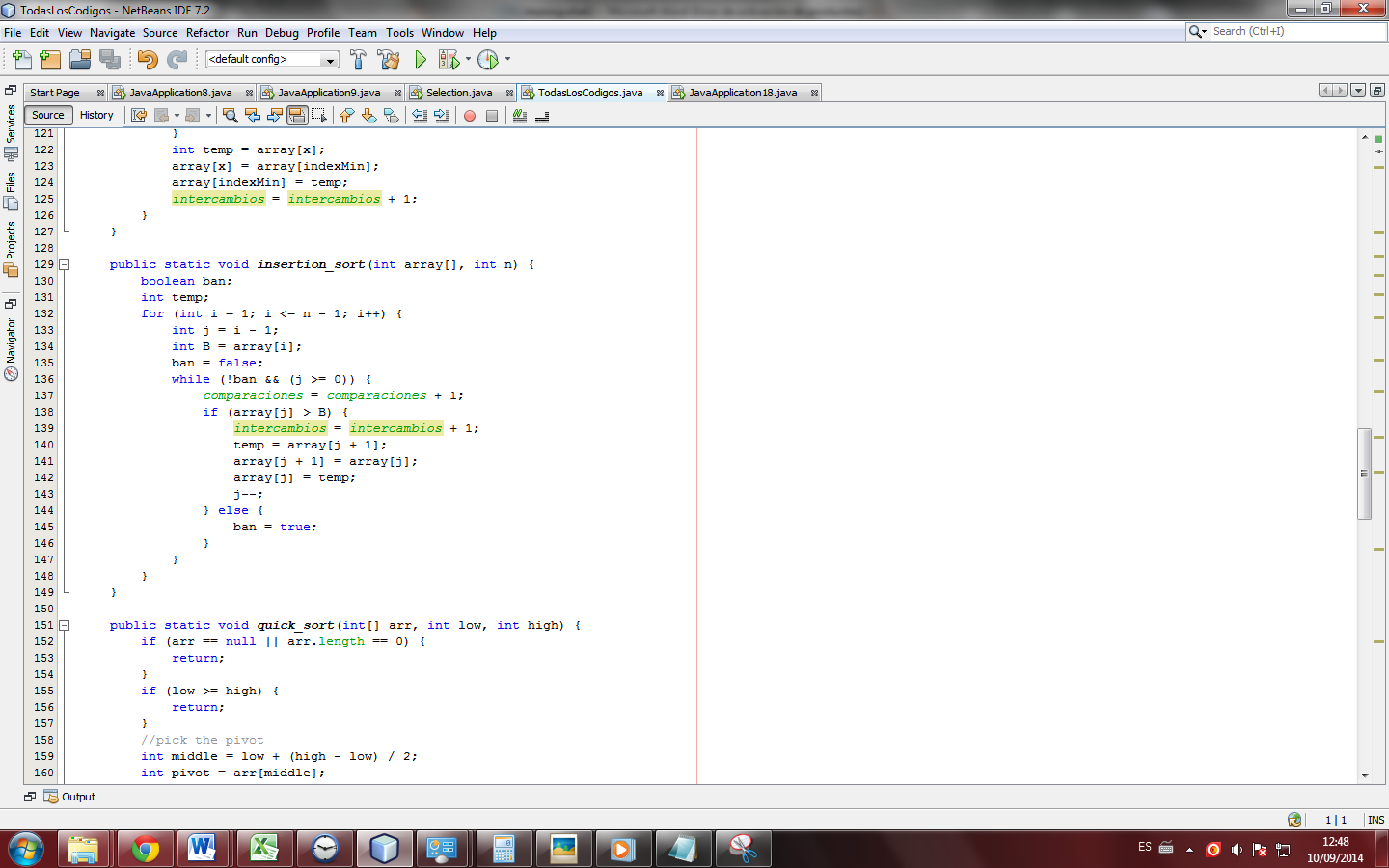
Código para generar arreglo aleatorio

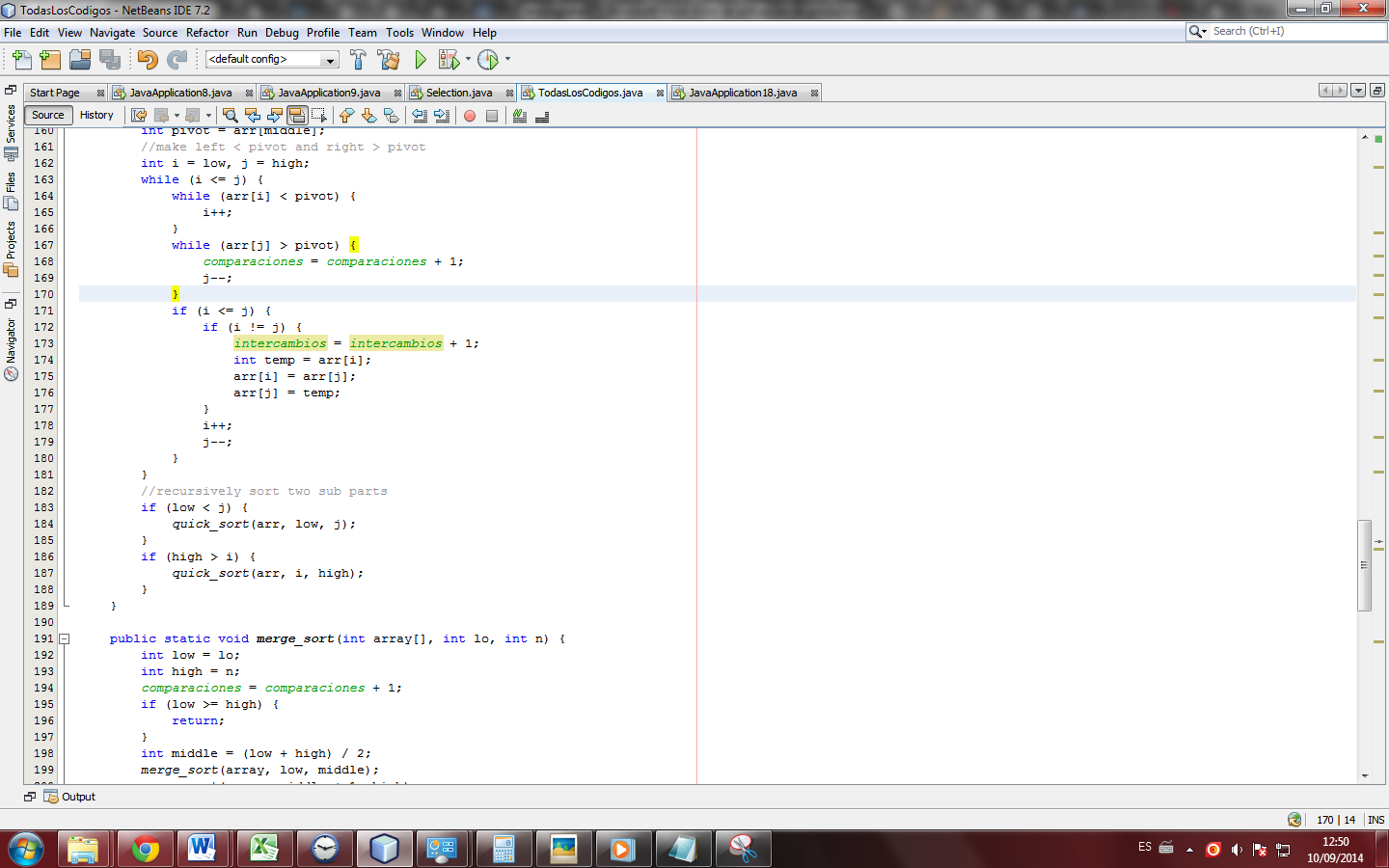


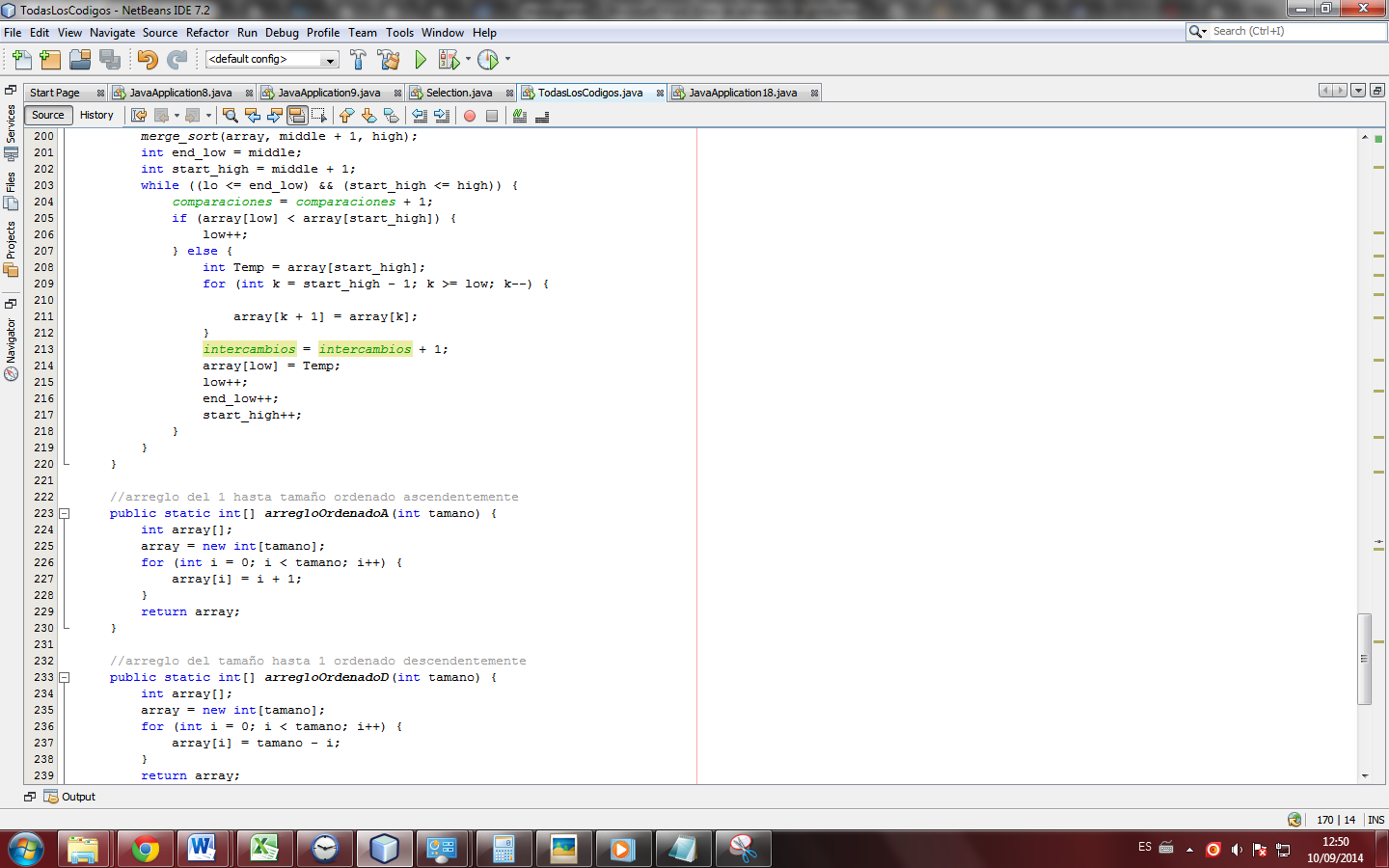
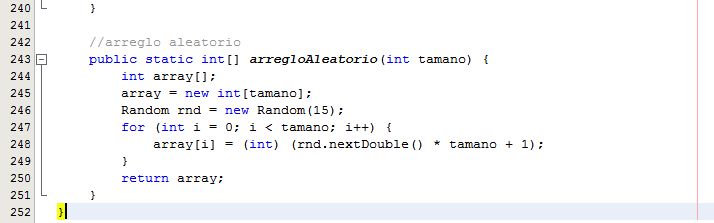
**Anexo 2**

**Código desarrollado para las pruebas**









**Anexo 3 - Tablas**

**Anexo 3.1**

|  |  |  |  |
| --- | --- | --- | --- |
| **Bubblesort - Ascendente** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **3** | 0 | 124750 |
| 1000 | **4** | 0 | 499500 |
| 5000 | **30** | 0 | 12497500 |
| 10000 | **109** | 0 | 49995000 |
| 50000 | **2670** | 0 | 1249975000 |
| 100000 | **10790** | 0 | 4999950000 |

**Tabla 1**. Tiempo de ejecución del Bubble sort en ascendente con numero de intercambios y comparaciones

**Anexo 3.2**

|  |  |  |  |
| --- | --- | --- | --- |
| **Bubblesort - Descendente** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **3** | 124750 | 124750 |
| 1000 | **5** | 499500 | 499500 |
| 5000 | **34** | 12497500 | 12497500 |
| 10000 | **126** | 49995000 | 49995000 |
| 50000 | **3085** | 1249975000 | 1249975000 |
| 100000 | **12498** | 4999950000 | 4999950000 |

**Tabla 2**. Tiempo de ejecución del Bubble sort en descendente con numero de intercambios y comparaciones

**Anexo 3.3**

|  |  |  |  |
| --- | --- | --- | --- |
| **Bubblesort - Aleatorio** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **4** | 60687 | 124750 |
| 1000 | **6** | 246951 | 499500 |
| 5000 | **61** | 6251204 | 12497500 |
| 10000 | **233** | 25114270 | 49995000 |
| 50000 | **5743** | 624086748 | 1249975000 |
| 100000 | **22647** | 2492430171 | 4999950000 |

**Tabla 3**. Tiempo de ejecución del Bubble sort en aleatorio con numero de intercambios y comparaciones

**Anexo 3.4**

|  |  |  |  |
| --- | --- | --- | --- |
| **Selectionsort - Ascendente** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **3** | 499 | 124750 |
| 1000 | **4** | 999 | 499500 |
| 5000 | **29** | 4999 | 12497500 |
| 10000 | **108** | 9999 | 49995000 |
| 50000 | **2722** | 49999 | 1249975000 |
| 100000 | **10745** | 99999 | 4999950000 |

**Tabla 5**. Tiempo de ejecución del Selectionsort en ascendente con numero de intercambios y comparaciones

**Anexo 3.5**

|  |  |  |  |
| --- | --- | --- | --- |
| **Selectionsort - Descendente** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **3** | 499 | 124750 |
| 1000 | **5** | 999 | 499500 |
| 5000 | **27** | 4999 | 12497500 |
| 10000 | **95** | 9999 | 49995000 |
| 50000 | **2343** | 49999 | 1249975000 |
| 100000 | **9418** | 99999 | 4999950000 |

**Tabla 6**. Tiempo de ejecución del Selectionsort en descendente con numero de intercambios y comparaciones

**Anexo 3.6**

|  |  |  |  |
| --- | --- | --- | --- |
| **Selectionsort - Aleatoria** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **3** | 499 | 124750 |
| 1000 | **4** | 999 | 499500 |
| 5000 | **30** | 4999 | 12497500 |
| 10000 | **109** | 9999 | 49995000 |
| 50000 | **2710** | 49999 | 1249975000 |
| 100000 | **10752** | 99999 | 4999950000 |

**Tabla 7**. Tiempo de ejecución del Selectionsort en aleatorio con numero de intercambios y comparaciones

**Anexo 3.7**

|  |  |  |  |
| --- | --- | --- | --- |
| **Insertionsort - Ascendente** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **0** | 0 | 499 |
| 1000 | **0** | 0 | 999 |
| 5000 | **0** | 0 | 4999 |
| 10000 | **1** | 0 | 9999 |
| 50000 | **3** | 0 | 49999 |
| 100000 | **5** | 0 | 99999 |

**Tabla 9**. Tiempo de ejecución del Insertionsort en ascendente con numero de intercambios y comparaciones

**Anexo 3.8**

|  |  |  |  |
| --- | --- | --- | --- |
| **Insertionsort - Descendente** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **7** | 124750 | 124750 |
| 1000 | **7** | 499500 | 499500 |
| 5000 | **22** | 12497500 | 12497500 |
| 10000 | **69** | 49995000 | 49995000 |
| 50000 | **1622** | 1249975000 | 1249975000 |
| 100000 | **6521** | 4999950000 | 4999950000 |

**Tabla 10**. Tiempo de ejecución del Insertionsort en descendente con numero de intercambios y comparaciones

**Anexo 3.9**

|  |  |  |  |
| --- | --- | --- | --- |
| **Insertionsort - Aleatorio** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **5** | 60687 | 61180 |
| 1000 | **7** | 246951 | 247944 |
| 5000 | **15** | 6251204 | 6256196 |
| 10000 | **38** | 25114270 | 25124262 |
| 50000 | **827** | 624086748 | 624136737 |
| 100000 | **3305** | 2492430171 | 2492530160 |

**Tabla 11**. Tiempo de ejecución del Insertionsort en aleatorio con numero de intercambios y comparaciones

**Anexo 3.10**

|  |  |  |  |
| --- | --- | --- | --- |
| **Quicksort - Ascendente** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **0** | 0 | 1776 |
| 1000 | **0** | 0 | 4049 |
| 5000 | **4** | 0 | 27013 |
| 10000 | **9** | 0 | 59018 |
| 50000 | **22** | 0 | 351963 |
| 100000 | **25** | 0 | 753916 |

**Tabla 13**. Tiempo de ejecución del Quicksort en ascendente con numero de intercambios y comparaciones

**Anexo 3.11**

|  |  |  |  |
| --- | --- | --- | --- |
| **Quicksort - Descendente** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **0** | 250 | 1528 |
| 1000 | **0** | 500 | 3552 |
| 5000 | **4** | 2500 | 24520 |
| 10000 | **5** | 5000 | 54026 |
| 50000 | **19** | 25000 | 326972 |
| 100000 | **21** | 50000 | 703926 |

**Tabla 14**. Tiempo de ejecución del Quicksort en descendente con numero de intercambios y comparaciones

**Anexo 3.12**

|  |  |  |  |
| --- | --- | --- | --- |
| **Quicksort - Aleatorio** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **0** | 1080 | 1725 |
| 1000 | **1** | 2424 | 3307 |
| 5000 | **5** | 14949 | 21600 |
| 10000 | **7** | 31750 | 54223 |
| 50000 | **21** | 187452 | 298247 |
| 100000 | **23** | 399875 | 582074 |

**Tabla 15**. Tiempo de ejecución del Quicksort en aleatorio con numero de intercambios y comparaciones

**Anexo 3.13**

|  |  |  |  |
| --- | --- | --- | --- |
| **Mergesort - Ascendente** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **0** | 0 | 5487 |
| 1000 | **1** | 0 | 11975 |
| 5000 | **8** | 0 | 71807 |
| 10000 | **10** | 0 | 153615 |
| 50000 | **14** | 0 | 884463 |
| 100000 | **17** | 0 | 1868927 |

**Tabla 17**. Tiempo de ejecución del Mergesort en ascendente con numero de intercambios y comparaciones

**Anexo 3.14**

|  |  |  |  |
| --- | --- | --- | --- |
| **Mergesort - Descendente** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **6** | 124750 | 3215 |
| 1000 | **10** | 499500 | 6931 |
| 5000 | **25** | 12497500 | 39803 |
| 10000 | **58** | 49995000 | 84607 |
| 50000 | **1250** | 1249975000 | 482511 |
| 100000 | **5141** | 4999950000 | 1015023 |

**Tabla 18**. Tiempo de ejecución del Mergesort en descendente con numero de intercambios y comparaciones

**Anexo 3.15**

|  |  |  |  |
| --- | --- | --- | --- |
| **Mergesort - Aleatorio** | | | |
| **Numero de datos a ordenar** | **Tiempo en milisegundos** | **Intercambios** | **Comparaciones** |
| 500 | **4** | 60905 | 5154 |
| 1000 | **7** | 247465 | 11311 |
| 5000 | **25** | 6253793 | 67935 |
| 10000 | **49** | 25119302 | 145878 |
| 50000 | **820** | 624111847 | 845729 |
| 100000 | **3135** | 2492480593 | 1791474 |

**Tabla 19**. Tiempo de ejecución del Mergesort en aleatorio con numero de intercambios y comparaciones

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Clifford, S. (2000). *Introduction to Algorithms.* Cambridge: MA: The MIT Press. [↑](#footnote-ref-1)
2. Joyannes Aguilar, L. (2003). *Fundamentos de programación: Algoritmos, Estructuras de datos, y Objetos.* Madrid: McGraw-Hill/Interamericana de España, S.A.U. [↑](#footnote-ref-2)
3. Joyannes Aguilar, L. (2003). *Fundamentos de programación: Algoritmos, Estructuras de datos, y Objetos.* Madrid: McGraw-Hill/Interamericana de España, S.A.U. [↑](#footnote-ref-3)
4. Allain, A. (2011). *Sorting Algorithm Comparison.* Obtenido de http://www.cprogramming.com: http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html [↑](#footnote-ref-4)
5. Knuth, D. (1968). *The Art of Computer Programming.* Standford: Addison-Wesley. [↑](#footnote-ref-5)
6. MIT Lecture. (2003). *Big O Notation.* Obtenido de http://web.mit.edu: http://web.mit.edu/16.070/www/lecture/big\_o.pdf [↑](#footnote-ref-6)
7. González, J. A. (2012). *Análisis y Diseño de Algoritmos Notación Asintótica .* Obtenido de http://ccc.inaoep.mx/: http://ccc.inaoep.mx/~jagonzalez/ADA/NotacionA.pdf [↑](#footnote-ref-7)
8. Allain, A. (2011). *Sorting Algorithm Comparison.* Obtenido de http://www.cprogramming.com: http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html [↑](#footnote-ref-8)
9. Orcola, C. (2004). *Gestión de datos.* Universidad Tecnológica Nacional. [↑](#footnote-ref-9)
10. Sedgewick, R. (2002). Chapter 6.5 Bubble Sort. *Algorithms in Java: Parts 1-4, Third Edition.* Addison Wesley. [↑](#footnote-ref-10)
11. Sedgewick, R. (2002). Chapter 6.3 Selection Sort. *Algorithms in Java: Parts 1-4, Third Edition.* Addison Wesley. [↑](#footnote-ref-11)
12. Sedgewick, R. (2002). Chapter 6.4 Insertion Sort. *Algorithms in Java: Parts 1-4, Third Edition.* Addison Wesley. [↑](#footnote-ref-12)
13. Sedgewick, R. (2002). Chapter 7 Quicksort. *Algorithms in Java: Parts 1-4, Third Edition.* Addison Wesley. [↑](#footnote-ref-13)
14. Sedgewick, R. (2002). Chapter 8 Merging and Mergesort. *Algorithms in Java: Parts 1-4, Third Edition.* Addison Wesley. [↑](#footnote-ref-14)