Alejandro Fonseca Forero - A00362167

Diego Alejandro Garcia - A00362738

David Steven Montoya - A00362450

Marco Fidel Vásquez - A00362057

**Diseño de experimentos - Complejidad temporal**

**Planeación y realización**

Delimitación del objeto de estudio

El fenómeno de estudio de este experimento es el comportamiento de dos algoritmos de ordenamiento: el bubble sort, merge sort. Esto es importante debido a que es necesario tener claridad en cuál es el más eficaz en un entorno real, en donde es común hallar una variedad de casos donde es necesario ordenar cadenas de datos de diferentes longitudes.

Variables de respuesta

La variable de respuesta que será tomada en cuenta en este estudio es el tiempo y la complejidad espacial, ya que son las principales métricas utilizadas para evaluar la eficiencia de los algoritmos. Las unidades utilizadas para la medición de tiempo son los milisegundos, pues proveen la precisión necesaria para la evaluación de eficiencia.

Nuestra herramienta de medición consiste en la clase Stopwatch de C#, el método System.currentTimeMillis, y el método de python time.time (multiplicado por 1000).

Dado a que los resultados con estas herramientas varían en cada medición, se probará cada caso de los algoritmos 100 veces y de esto se sacará el promedio de tiempo para tener un resultado más confiable.

Factores a estudiar

Factores controlables:

* Algoritmo de Ordenamiento utilizado.
* Tamaño del arreglo.
* Estado de los valores en el arreglo.
* Lenguaje de programación.
* Cantidad de procesos que se están ejecutando en el computador mientras se ejecuta el algoritmo.

Factores no controlables:

* Lógica del algoritmo.
* Procesos fundamentales del sistema operativo.

Factores estudiados:

* Algoritmo de Ordenamiento utilizado.
* Tamaño del arreglo.
* Lenguaje de programación.
* Estado del arreglo.

Niveles de cada factor

|  |  |
| --- | --- |
| Variables | Niveles |
| Algoritmo de Ordenamiento utilizado. | 2 niveles |
| Tamaño del arreglo. | 3 niveles |
| Lenguaje de programación. | 2 niveles |
| Estado del arreglo. | 3 niveles |

Algoritmo de ordenamiento utilizado

|  |  |
| --- | --- |
| Nivel | Valor |
| 1 | Bubble sort |
| 2 | Merge sort |

Tamaño del arreglo

|  |  |
| --- | --- |
| Nivel | Valor |
| 1 | 10^2 |
| 2 | 10^3 |
| 3 | 10^4 |

Lenguaje de programación

|  |  |
| --- | --- |
| Nivel | Valor |
| 1 | Java |
| 2 | C# |

Estado del arreglo

|  |  |
| --- | --- |
| Nivel | Valor |
| 1 | En orden aleatorio |
| 2 | En orden descendente |
| 3 | En orden ascendente |

Para la medición de espacio, se extraerá la cantidad de RAM utilizada para cada proceso.

**Organización del trabajo experimental**

La implementación de los algoritmos estarán en Visual Studio Community para C# y python. Para Java, la implementación será realizada en Eclipse IDE.

El experimento será realizado en solo una computadora, para evitar cambios en los resultados producto de las diferentes especificaciones de cada uno. En cada experimento se harán 100 repeticiones de cada algoritmo según los tratamientos definidos en la siguiente tabla:

<https://docs.google.com/spreadsheets/d/1WeuEzbhDXrCqhWLW7I_DSQd6ENEXy6lSoOAmovjgA5w/edit?usp=sharing>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritmo utilizado | Lenguaje de programación | Tamaño del arreglo utilizado | Estado del arreglo utilizado | tratamiento |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 2 |
| 1 | 2 | 1 | 1 | 3 |
| 2 | 2 | 1 | 1 | 4 |
| 1 | 1 | 2 | 1 | 5 |
| 2 | 1 | 2 | 1 | 6 |
| 1 | 2 | 2 | 1 | 7 |
| 2 | 2 | 2 | 1 | 8 |
| 1 | 1 | 3 | 1 | 9 |
| 2 | 1 | 3 | 1 | 10 |
| 1 | 2 | 3 | 1 | 11 |
| 2 | 2 | 3 | 1 | 12 |
| 1 | 1 | 1 | 2 | 13 |
| 2 | 1 | 1 | 2 | 14 |
| 1 | 2 | 1 | 2 | 15 |
| 2 | 2 | 1 | 2 | 16 |
| 1 | 1 | 2 | 2 | 17 |
| 2 | 1 | 2 | 2 | 18 |
| 1 | 2 | 2 | 2 | 19 |
| 2 | 2 | 2 | 2 | 20 |
| 1 | 1 | 3 | 2 | 21 |
| 2 | 1 | 3 | 2 | 22 |
| 1 | 2 | 3 | 2 | 23 |
| 2 | 2 | 3 | 2 | 24 |
| 1 | 1 | 1 | 3 | 25 |
| 2 | 1 | 1 | 3 | 26 |
| 1 | 2 | 1 | 3 | 27 |
| 2 | 2 | 1 | 3 | 28 |
| 1 | 1 | 2 | 3 | 29 |
| 2 | 1 | 2 | 3 | 30 |
| 1 | 2 | 2 | 3 | 31 |
| 2 | 2 | 2 | 3 | 32 |
| 1 | 1 | 3 | 3 | 33 |
| 2 | 1 | 3 | 3 | 34 |
| 1 | 2 | 3 | 3 | 35 |
| 2 | 2 | 3 | 3 | 36 |

El tiempo solo será tomado durante el proceso de ordenamiento, ya que en el código también se exportarán los datos.

Los pasos que se llevarán a cabo en cada uno de los tratamientos serán los siguientes:

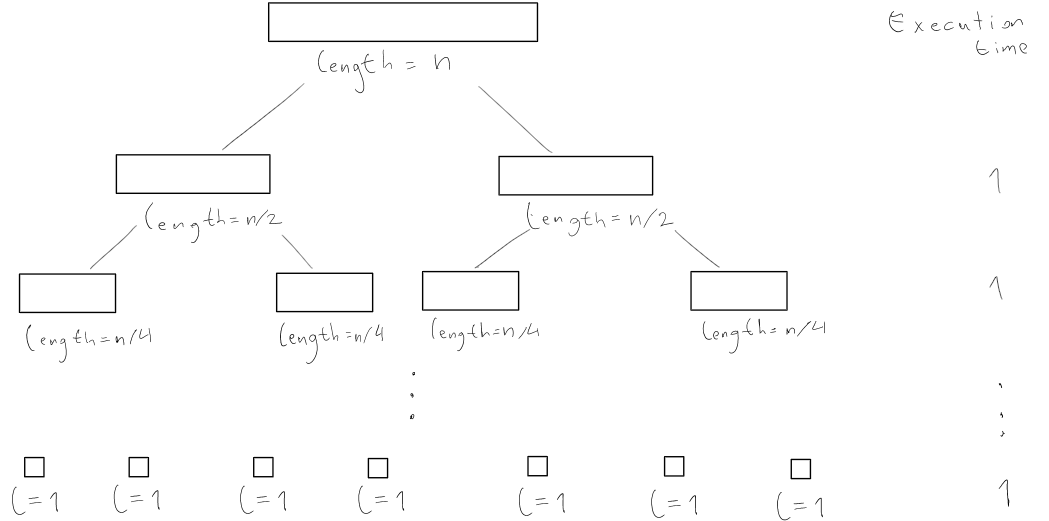
1. Generar el arreglo a ordenar.
2. Ejecutar el algoritmo de ordenamiento y tomar el tiempo que se demora y la memoria que usa.
3. Repetir los pasos 1 y 2 cien veces.
4. Generar una tabla con los tiempos y memorias de cada iteración, además de sus promedios.
5. Exportar la tabla en un archivo formato csv.

Los resultados del experimento serán registrados en un archivo con formato .csv, donde irán el tiempo utilizado en milisegundos, la memoria consumida en kilobytes y el tratamiento al cual pertenecen esos resultados. El documento se dividirá en dos hojas: En una estarán todos los tiempos tomados en cada tratamiento, y en otra estarán los promedios.

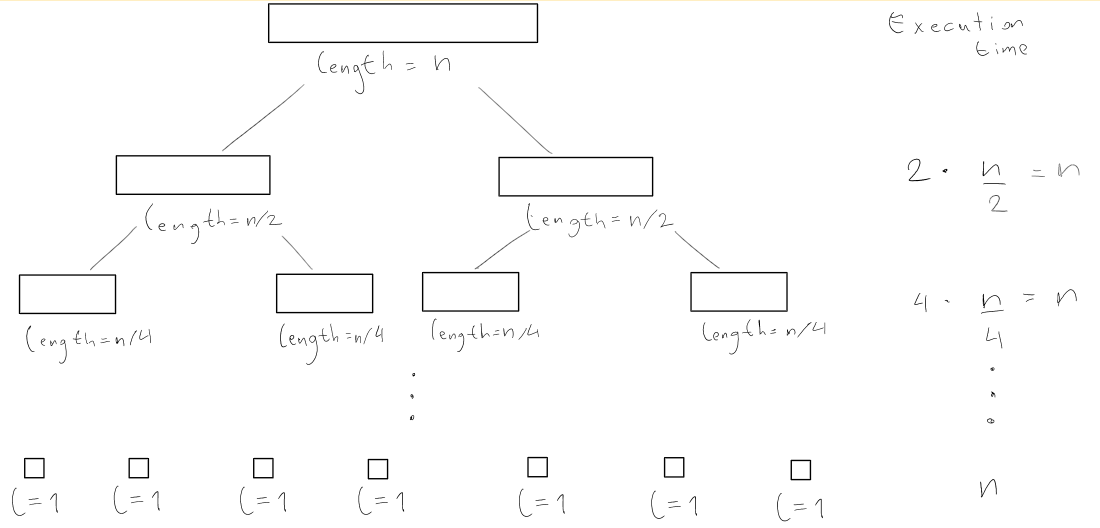
**Análisis temporal**

**Merge Sort**

Merge Sort es un algoritmo recursivo, que divide un array de tamaño n en dos arrays de tamaño n/2 hasta tener arrays de tamaño 1. El proceso de división tiene una complejidad temporal de Θ(1), y se realiza tal cual como en el siguiente árbol de recursión:



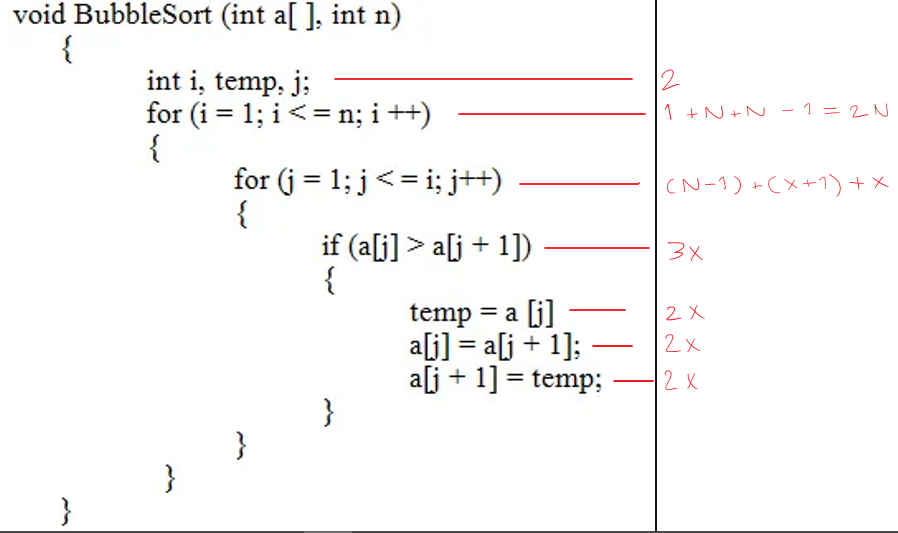
Cuando el proceso de división termina y se llega al último nivel (n arrays de tamaño 1), el siguiente paso es realizar el proceso de merge, que une nuevamente a los arrays, el cual tiene una complejidad temporal de Θ(n). El árbol de recursión abajo muestra este proceso:

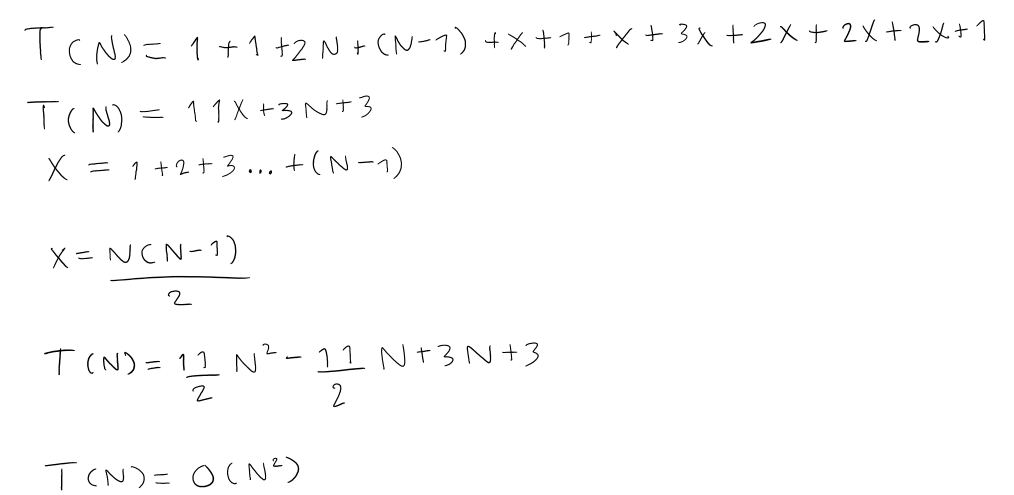


Como el Θ(1) del proceso de división es menor al Θ(n) del proceso de merge, se toma el proceso de división y merge como Θ(n).

Como este proceso se realiza en cada nivel, debemos obtener Θ(x \* n) donde x es todos los niveles (o subcasos) del caso principal. Sabemos que recorrer todos los casos en los que un caso n se divide hasta llegar a un caso 1 tiene una complejidad temporal de Así, sabemos el valor de x y la complejidad temporal completa del Merge Sort es

**Bubble Sort**





En la versión estándar del bubble sort, debemos hacer N iteraciones. En cada iteración comparamos e intercambiamos de ser requerido. Dado un array de tamaño N, la primera iteración realiza (N-1) comparaciones. La segunda iteración (N-2). De esto se llega a:



Es decir, la complejidad temporal en el caso estándar es de O()

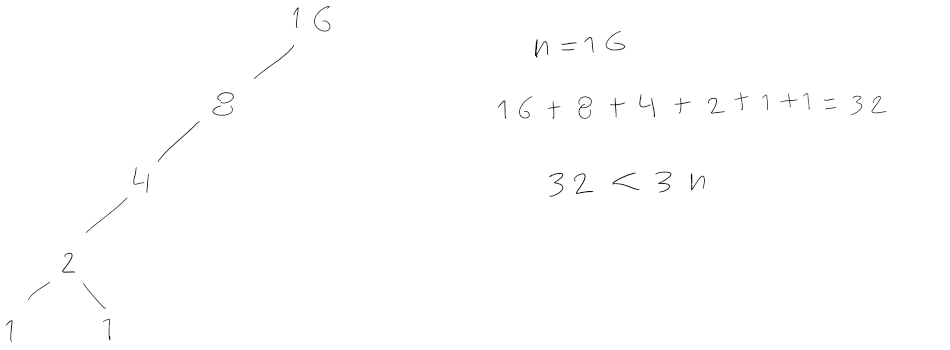
El mejor caso es en el cual el array ya está ordenado. En este caso se lee cada posición para ver si son necesarios intercambios. De igual forma se toman N iteraciones, entonces, el mejor caso para el bubble sort es igualmente O()

En el peor caso, el array está revertido. Así que hacemos (N-1) comparaciones en la primera iteración, (N-2) en la segunda, y así sucesivamente. Por ende, el peor caso para bubble sort es también O()

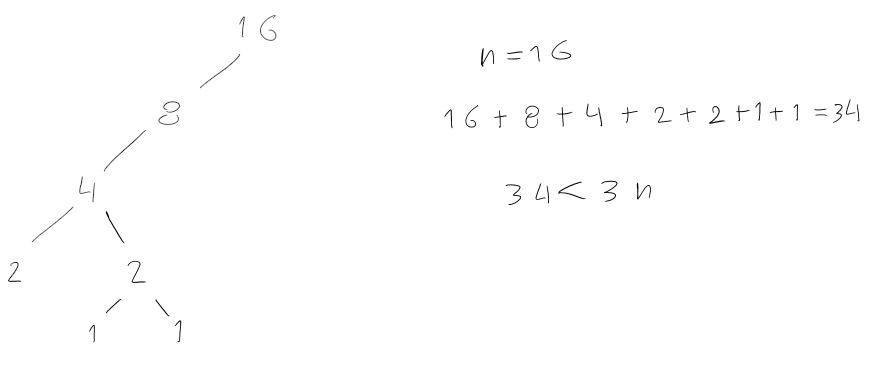
**Análisis espacial**

**Merge Sort**

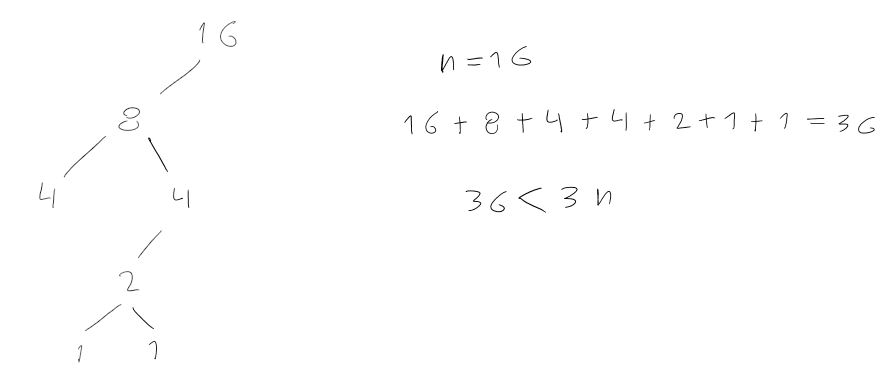
Al momento de ejecutar el algoritmo de Merge Sort, este se trabaja secuencialmente, y en ningún momento de esa secuencia la memoria usada es mayor a 3n. A continuación se muestra un ejemplo con un array de tamaño 16.



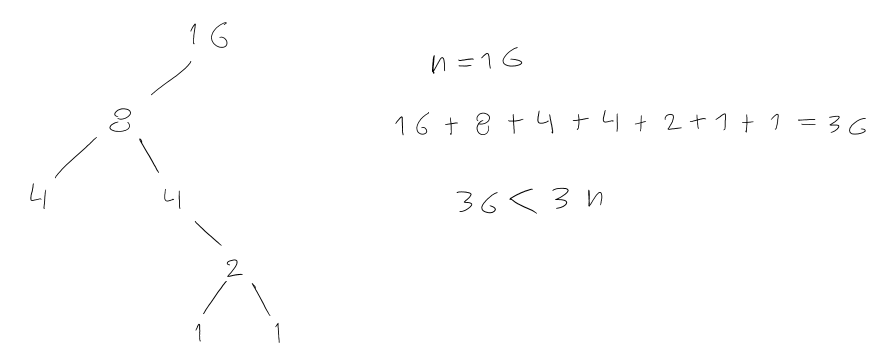
Al llegar al final de casos, empieza el proceso de merge:



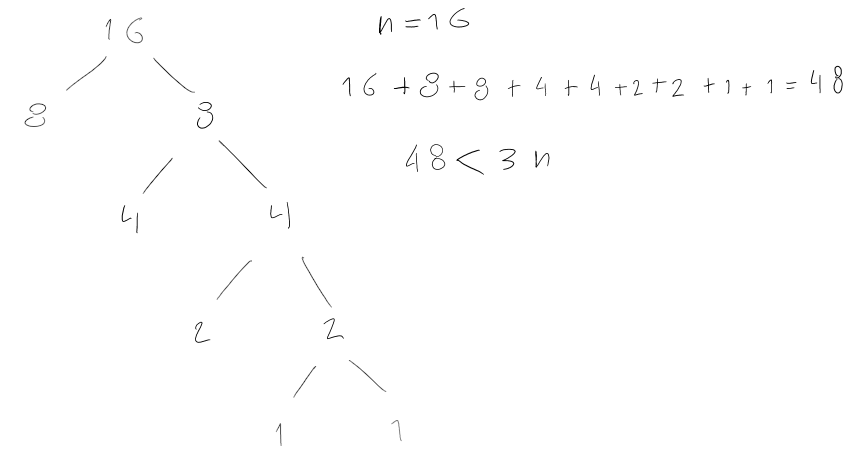
Otro merge:



Otro merge:

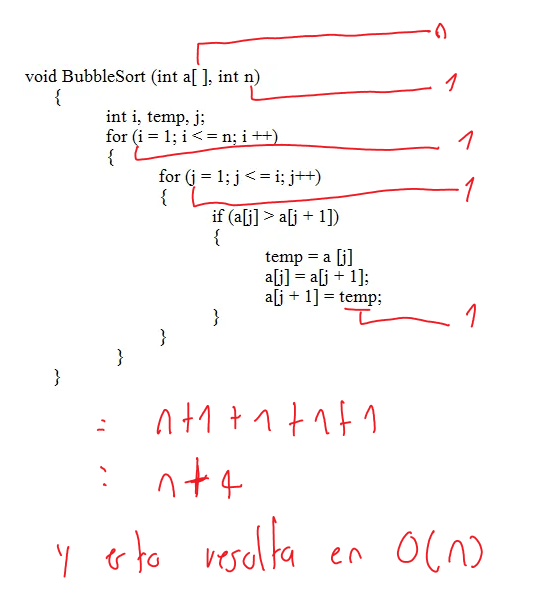


Merge final:



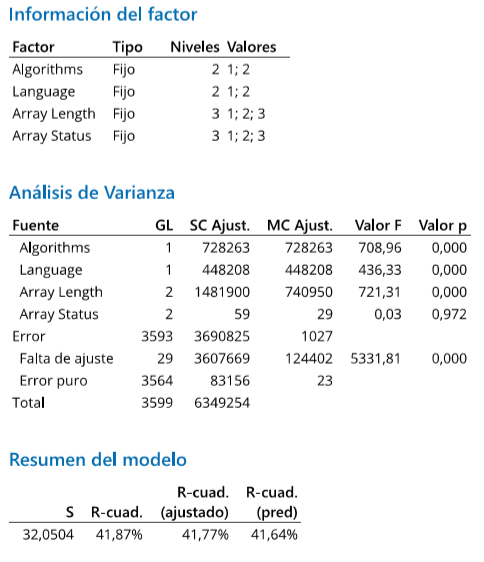
Entonces tenemos O(3n) = O(n). Así, la complejidad espacial del Merge Sort es O(n).

**Bubble Sort**



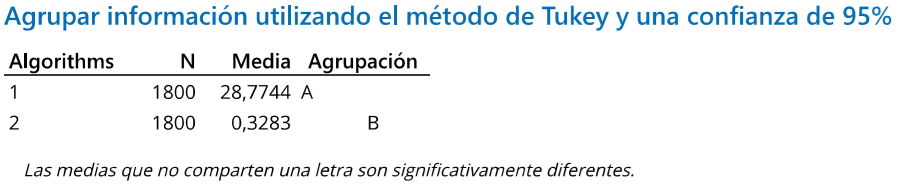
**Análisis**

A continuación se mostraran algunos de los datos obtenidos haciendo uso de la ANOVA y las pruebas de TUKEY. En caso de requerirse mayor información, ingrese a los Anexos 1 y 2.

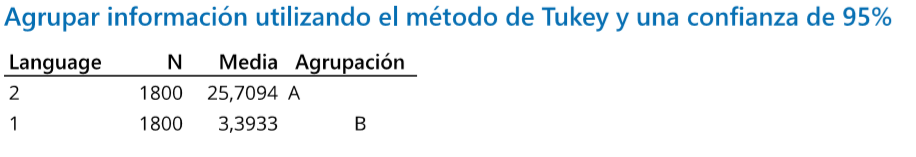
****

**Pruebas Tukey**

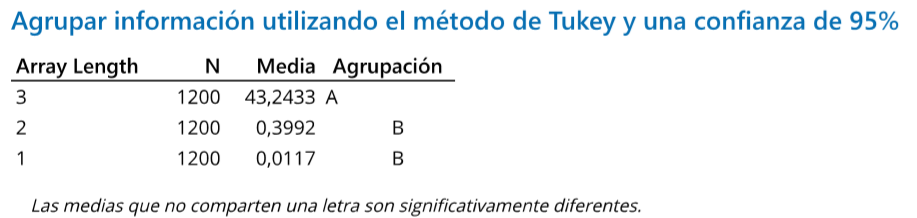
**Algoritmos:**

****

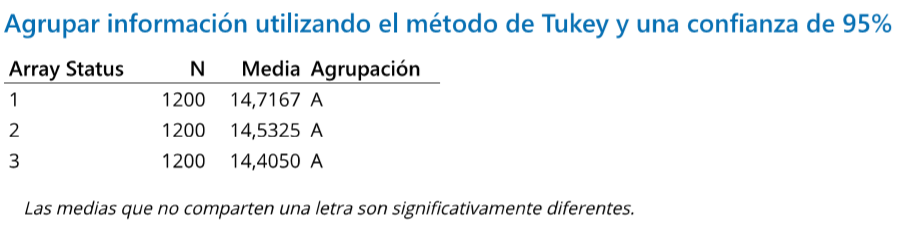
**Lenguaje:**

****

**Array Length:**

****

**Array status:**

****

**Interpretación**

Con base en los resultados del análisis estadístico podemos concluir los siguientes enunciados:

* El algoritmo Merge Sort, con una media de 0,33 ms, tiene un mejor rendimiento frente al algoritmo Bubble Sort, con una media de 28,77 ms.
* La implementación de los algoritmos de ordenamiento en Java tiene una media menor la implementación de los algoritmos en C#, pues el promedio temporal de los algoritmos implementados en Java es menor al promedio temporal de los algoritmos implementados en C#, con 3,39 ms y 25,71 ms respectivamente.
* Al utilizar diferentes tamaños de arreglos pudimos evidenciar el crecimiento cuadrático del costo temporal que tiene el algoritmo Bubble sort ya que, como se expuso anteriormente, este tiene una complejidad de O(n^2). Es por esto que al aumentar el tamaño del arreglo, el algoritmo tardaba considerablemente más en ordenar el arreglo.
* De los diferentes tamaños de los arreglos, el arreglo de tamaño 10.000 es el más tardado, con una media de 43,24 ms. Además, la media del arreglo de tamaño 1.000 y del arreglo de tamaño 100 son estadísticamente iguales, por lo que ambas son opciones viables.
* Los diferentes estados de arreglo son estadísticamente iguales, por lo que su influencia en la velocidad de ejecución es muy mínima.

**Conclusiones**

De acuerdo con los resultados del experimento:

* El algoritmo de ordenamiento Merge sort es más eficiente que el Bubble Sort en términos generales(tiempo, lenguaje, cantidad de números).
* La mejor combinación para ordenar un arreglo de números es Merge sort programado en Java.
* Java presenta un rendimiento más eficiente que C#.
* El algoritmo Merge sort se ve afectado muy levemente en rendimiento si se aumenta la cantidad de números a ordenar.

Informes de tiempo en wakatime:

Marco:

<https://wakatime.com/@5688507e-c4ef-4ed7-9f14-45a43915cbf3/projects/cuabzflhkx?start=2021-03-06&end=2021-03-12>

<https://wakatime.com/@5688507e-c4ef-4ed7-9f14-45a43915cbf3/projects/mbzzhrtzvr?start=2021-03-06&end=2021-03-12>

Alejandro:

<https://wakatime.com/@d250453c-3f9c-4e12-a29f-db2f8b7b6f41/projects/jiiykwfybp?start=2021-03-06&end=2021-03-12>

<https://wakatime.com/@d250453c-3f9c-4e12-a29f-db2f8b7b6f41/projects/azgzeeasey?start=2021-03-06&end=2021-03-12>

Diego:

<https://wakatime.com/@Diego_ds/projects/echvmcpbju?start=2021-03-06&end=2021-03-12>

Anexo 1:

[ANOVA](https://github.com/AlejandroFonseca25/sort-experiment/blob/main/Docs/Anova%20generalizado.pdf)

Anexo 2:

[Pruebas TUKEY](https://github.com/AlejandroFonseca25/sort-experiment/blob/main/Docs/prueba_tukey%20para%20anova%20generalizado.pdf)