**Morales J.G, Campillo J., Bellaiza J.**

**Análisis de experimentos**

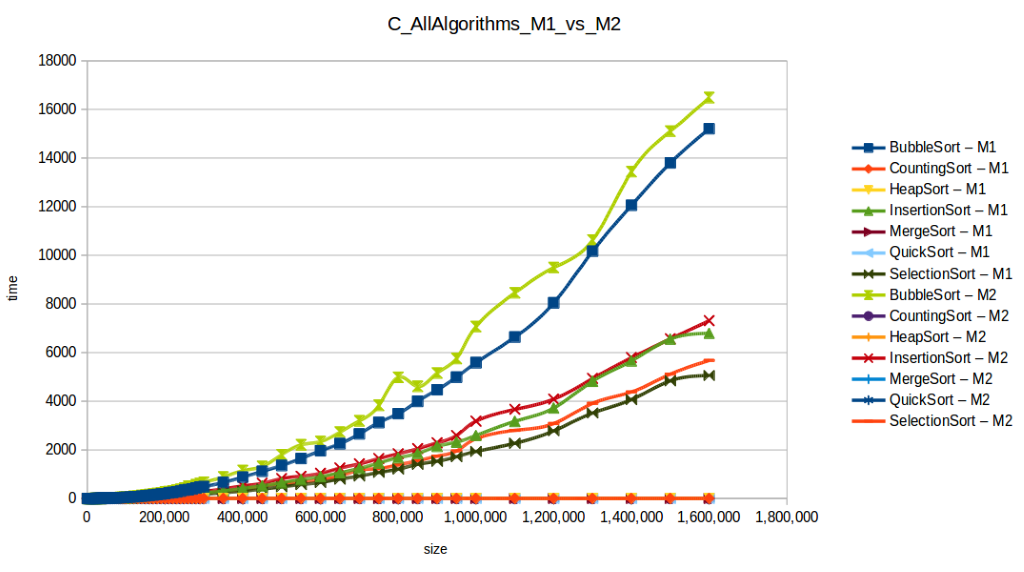
**Planeación y realización**

* **Entendimiento y delimitación del problema:**

El ordenamiento de datos es un proceso que se lleva a cabo en muchísimas aplicaciones, se puede ver desde el ordenamiento de cosas muy particulares hasta el ordenamiento de volúmenes muy grandes. La importancia del ordenamiento de datos radica en generar mayor rapidez a la hora de necesitar acceder a estos. Existen muchos algoritmos que pueden suplir esta necesidad, pero la potencia de estos depende en gran medida del tamaño de los datos que se quieren ordenar, esta potencia o relación se puede determinar por medio de las **complejidades.** La complejidad temporal nos arroja una notación matemática que nos determina qué pasará con el tiempo a medida que la cantidad de datos que ordena el algoritmo crece. A esta predicción se le conoce como **cuota superior.** No es la única que existe, también encontramos **cuota promedio y cuota inferior.** Lo que sucede con estas dos últimas cuotas es que estas nos brindan una notación, pero en términos de datos pequeños o de tamaño medio, la cual no nos conviene dado que, en la industria, se manejan volúmenes muy extensos de datos. Existe otro tipo de complejidad, la cual se encarga de establecer una cuota sobre cómo se comportará el gasto de espacio de memoria de la máquina en un algoritmo, esta se conoce como **complejidad espacial.** Teniendo en cuenta los conceptos anteriores, queremos determinar los comportamientos de los siguientes dos algoritmos: **Comb Sort y Counting Sort.**

* **Variables de respuesta:**

La mejor forma de poder determinar el comportamiento de un algoritmo de ordenamiento a medida que va creciendo la cantidad de datos es por medio del **tiempo**, de este modo nuestra variable de respuesta principal será el **tiempo**, así mediremos lo que se demora en ordenar **n** cantidad de datos. Por el lado de espacio de memoria, la variable que tendremos en cuenta es **espacio** gasto por el algoritmo.



* **Factores e influencia que genera sobre las variables de respuesta:**

Son muchos los factores que pueden afectar en poca o gran medida, el comportamiento de los algoritmos, en este caso, algoritmos de ordenamiento. De acuerdo con ello elegimos los siguientes:

* **Complejidad Temporal**
* Tamaño del arreglo (10^1, 10^2, 10^3, 10^4, etc)

El tamaño variable del arreglo nos brindara una visión muy extensa del comportamiento en tiempo de los algoritmos cuando el tamaño del arreglo se vaya tornando más grande.

* **Complejidad Espacial**
* Lenguaje de programación (Java, C#, Python, etc)

En los diferentes lenguajes de programación varía la asignación de espacios de memoria de acuerdo al tipo de variable que se desea almacenar. Por lo anterior podríamos darnos cuenta de cómo afecta a la complejidad espacial el hecho de que se programe en otro lenguaje.

* **Algoritmos a usar**
* Algoritmo de Ordenamiento (Burbuja, Selection, QuickSort, HeapSort, etc)

Se desea comparar dos algoritmos que manejan funciones de complejidad iguales. Así estaríamos comparando dos algoritmos en sus dos principales formas de comportamiento; **Complejidad temporal y complejidad espacial.** En nuestro caso usaremos el **Mergesort** y **CombSort** los cuales cuentan con un **complejidad temporal** de **O(nlogn)** y **complejidad espacial** de **O(n) y O(1)** respectivamente.

* **Nivel de los factores:**

**Tamaño del arreglo:**

* Nivel 1: 10^4 de posiciones en el arreglo. (Bajo)
* Nivel 2: 10^5 de posiciones en el arreglo. (Medio)
* Nivel 3: 10^6 de posiciones en el arreglo. (Alto)

**Lenguaje de programación:**

* Nivel 1: Lenguaje de programación C# .
* Nivel 2: Lenguaje de programación Java.

* **Planeación del trabajo experimental:**

**Primera fase:**

* Construcción del diagrama de clases y objetos de acuerdo a la propuesta utilizada.
* Codificación de los algoritmos en los diferentes lenguajes propuestos anteriormente.
* Codificación de las pruebas unitarias de los diferentes algoritmos, a manera de comprobar que estén ordenando correctamente.

**Segunda fase:**

* Medición de los tiempos en los cuales se tarda cada algoritmo en realizar el proceso de ordenamiento, en cada nivel del factor anteriormente mencionado, el cual era, el crecimiento del tamaño del arreglo. Registrar los datos tomados en una hoja electrónica.
* Medición de la cantidad de espacio gastada por cada algoritmo a la hora de realizar el proceso de ordenamiento. Registrar los datos tomados en una hoja electrónica.

**Tercera fase:**

* Realizar análisis tomando todos los datos registrados en la hoja electrónica.
* Pasar datos tomados a SPSS para analizarlos.
* Concluir teniendo en cuenta los análisis.

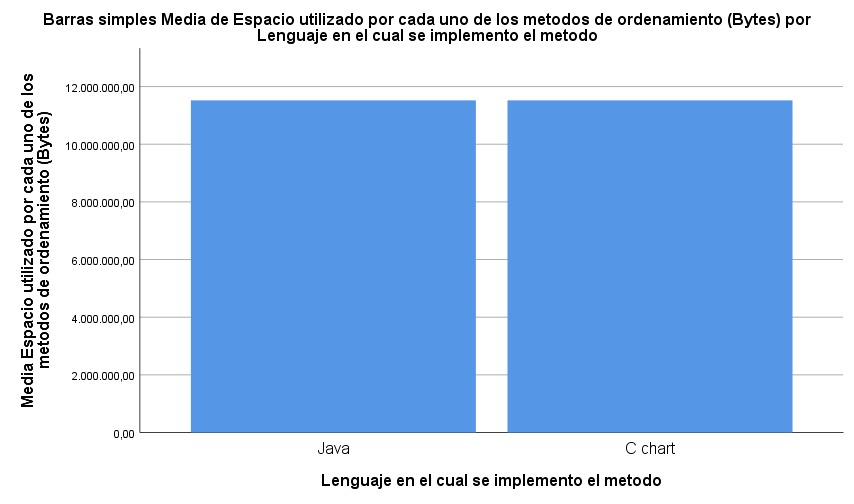
**Análisis**

**Figura 1**



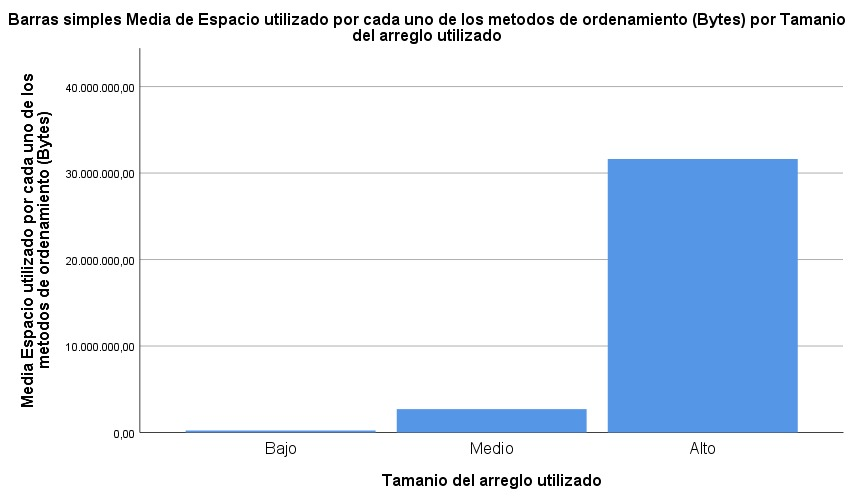
En la tabla anterior podemos apreciar con claridad la gran diferencia que tienen estos dos algoritmos de ordenamiento con respecto al espacio usado en la memoria de la máquina. Este resultado ya se esperaba con anterioridad, ya que sabíamos que la complejidad espacial del MergeSort ( O(n)) era mucho mayor a la del CombSort ( O(1)), es decir, que para el merge el espacio consumido por memoria es igual a multiplicar la cantidad de datos por su peso,en cambio el Comb no depende de la cantidad de datos, su complejidad espacial es constante .

**Figura 2**



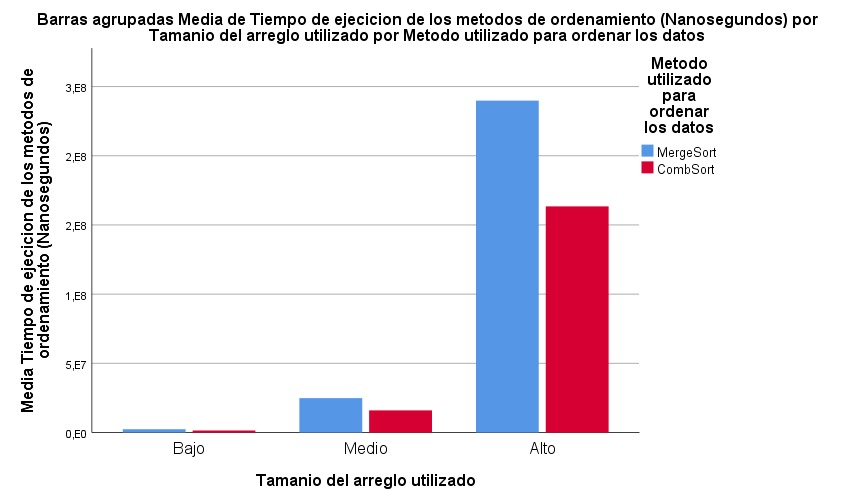
En esta gráfica se observa que sin importar el lenguaje de programación que se utilice (En este caso Java y C#) la complejidad espacial no va a variar, y será prácticamente la misma en ambos casos.

**Figura 3**



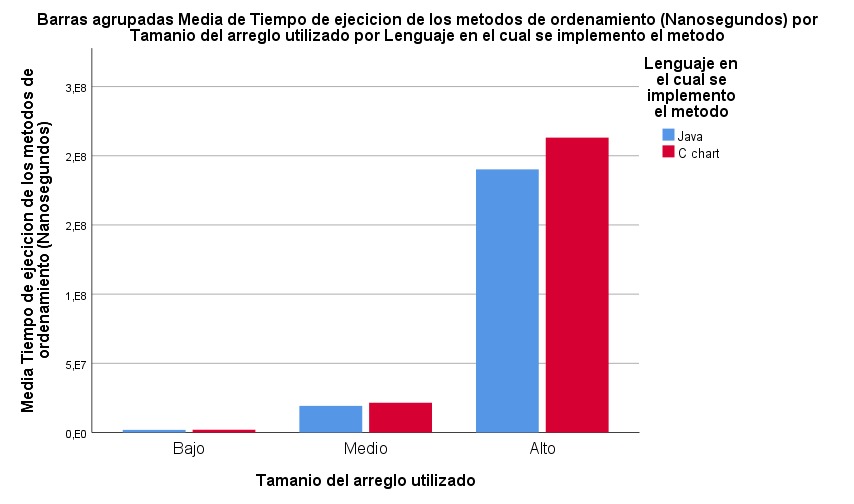
El consumo de espacio utilizado en memoria depende del tamaño del arreglo, entre mayor cantidad de datos, mayor consumo de espacio.

**Figura 4**



En esta gráfica se puede apreciar que para cantidades pequeñas y medianas de datos ambos algoritmos tardan relativamente lo mismo en ejecutarse completamente, pero para cantidades de datos mayores, la diferencia empieza a ser muy significativa, tomándole mucho menos tiempo al Comb ejecutarse totalmente, esto implica que el tipo de método utilizado si afecta en el momento de conseguir una mejora en el tiempo de ordenamiento.

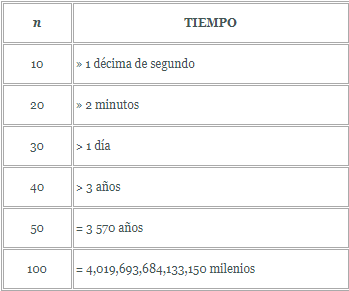
**Figura 5**



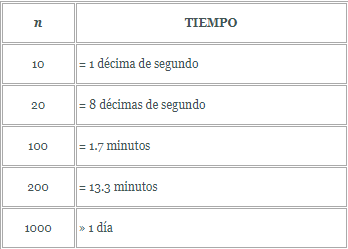
Esta situación nos proporciona información para afirmar que para cantidades bajas y medianas de datos en ambos lenguajes la media de tiempo de todos los algoritmos de ordenamiento no varía de forma significativa, pero cuando vemos el comportamiento en conjuntos de datos grandes, notamos que en Java los tiempos que toman las ejecuciones son menores en comparación a C #, aunque dado que pueden existir errores aleatorios en el momento de las pruebas, se puede decir que el tiempo de de ejecución no depende mucho del lenguaje.

**Interpretación**

Los datos y el análisis que nos proporcionó el estudio de los factores, nos brinda una guía a la hora de elegir con cuál algoritmo queremos ordenar los datos y a la vez en cual lenguaje implementarlo. Nos damos cuenta de que en el caso de estos dos algoritmos(**MergeSort y CombSort)** la diferencia en tiempo de ejecución en diferente lenguaje de programación va tener una diferencia muy pequeña, dependiendo de la envergadura del problema esta diferencia podría marcar la pauta para decidir que lenguaje elegir, imaginémonos, por ejemplo, un problema médico, en el cual la velocidad en la cual se organizan los datos tenga que ser muy exacta, es claro que en este caso tendríamos que implementarlo en el lenguaje que nos proporcione menos **error experimental**. ¿Que pasaría por si los algoritmos de ordenamientos que usamos no son eficientes?, La siguiente figura nos muestra el tiempo que tomaría un algoritmo con complejidad **O(2^n)**.



Tiempo de demora para un algoritmo con complejidad **O(n^3)**



Cabe notar que las cantidades de datos expuestas en las figuras son cantidades notablemente pequeñas en comparación con los volúmenes de datos que se manejan en la industria. Lo anterior nos permite entender lo importancia de tener un buen algoritmo, un algoritmo eficiente.

**Conclusiones**

* Es de gran importancia saber que el **Merge sort** es un algoritmo estable, esto quiere decir que no va a existir un error **ε** a la hora de ejecutar el algoritmo **n** veces. Por otro lado, el **Comb sort**, es un algoritmo inestable, esto quiere decir que existe la posibilidad de un error **ε** al ejecutar **n** veces. ¿Porque usar un algoritmo inestable si nos proporciona un error? Los algoritmos inestables son mucho más eficientes para entradas pequeñas dado que va existir la posibilidad de no ejecutar una parte del código, si no, que entre directamente a cierto bloque de código y salga inmediatamente, de este modo el error **ε** será o suficientemente pequeño.
* El **MergeSort** y el **CombSort** son algoritmos con complejidades temporales iguales, lo que nos sirve como base para predecir que los tiempos de ejecución serán muy cercanos, su diferencia radica un error experimental.