En el presente informe, referente a la práctica de "Algoritmos de Ordenamiento No Elementales", trataremos 3 (tres) algoritmos de ordenamiento (ShellSort, MergeSort, QuickSort), en los cuales analizaremos y compararemos los diferentes tiempos de ejecución que tienen estos algoritmos sobre distintas secuencias (Ordenado, Aleatorio e Inverso) y tamaños de entrada (1k, 100k, 500k, 1M).

Para esto utilizaremos el lenguaje de programación java en el cual codificamos los 3 algoritmos de ordenamiento, y medimos los tiempos que le tomaba a cada algoritmo ejecutarse para cada conjunto de datos.

Esto nos sirve para entender cómo se comporta cada algoritmo sobre cada conjunto de datos y compararlos entre sí para determinar en qué situaciones un algoritmo es más eficiente que otro, sus ventajas y desventajas, entre otras cosas.

Ahora pasaremos a describir los algoritmos de ordenamiento:

ShellSort:

Este algoritmo es sensible y no estable.

Se lo conoce como una variación del Insertion Sort. En el Insertion Sort movemos los elementos de lugar 1 distancia a la vez, lo cual implica una gran cantidad de movimientos si el elemento está lejos de su lugar de inserción. En el ShellSort este paso se ve reducido al permitir mover los elementos K distancias de pasos a la vez. La distancia a mover por cada paso se hace siguiendo una serie matemática determinada por

$K = K_{-1} * 3 + 1$. Empezando por el K más grande posible de acuerdo a la cantidad de elementos de la lista. Una vez se hayan ordenado todos los elementos de la "sublista" creada con la serie K actual, se reducida el valor de K creando otra "sublista" de elementos a ordenar (toda sublista ordenándose de forma parecida al insertionSort) hasta que el valor de K sea reducido a uno y todos los elementos de todas las sublistas sean ordenados. Lo cual lleva a que la lista quede ordenada.

Este algoritmo posee una complejidad computacional de $O(n^{1.5})$ en el peor de los casos. Aunque este valor puede variar de acuerdo a serie utilizada para determinar K.

MergeSort:

Este algoritmo es no sensible y estable.

Este algoritmo de ordenamiento trabaja dividiendo la lista en dos mitades, repitiendo la división en las sublistas hasta que las mismas ya no puedan ser divididas quedando en una sublista de un solo elemento (el cual se puede considerar ordenado). Una vez hecho esto, las sublistas son ordenadas y luego fusionadas con su otra mitad de forma recursiva hasta que toda la lista termine ordenada.

Este algoritmo posee una complejidad computacional de O (N * log(N)) en sus 3 casos (peor, promedio y mejor), Aunque también para este ordenamiento necesita una lista auxiliar de tamaño O(N).

QuickSort:

Este algoritmo es no sensible y no estable.

Este algoritmo de ordenamiento trabaja con el uso de un pivote de la lista (existen diferentes formas de elegirlo) y a través de este pivote se busca mover los elementos mayores al pivote a la derecha y

los elementos menores al pivote a la izquierda, de esta forma se busca que el valor pivote quede en su posición correcta final. Una vez hecho esto, se crearán dos particiones, una a la izquierda del pivote y otra a la derecha, en las cuales se aplicará el mismo proceso de forma recursiva hasta que toda la lista que de ordenada.

Elección del pivote:

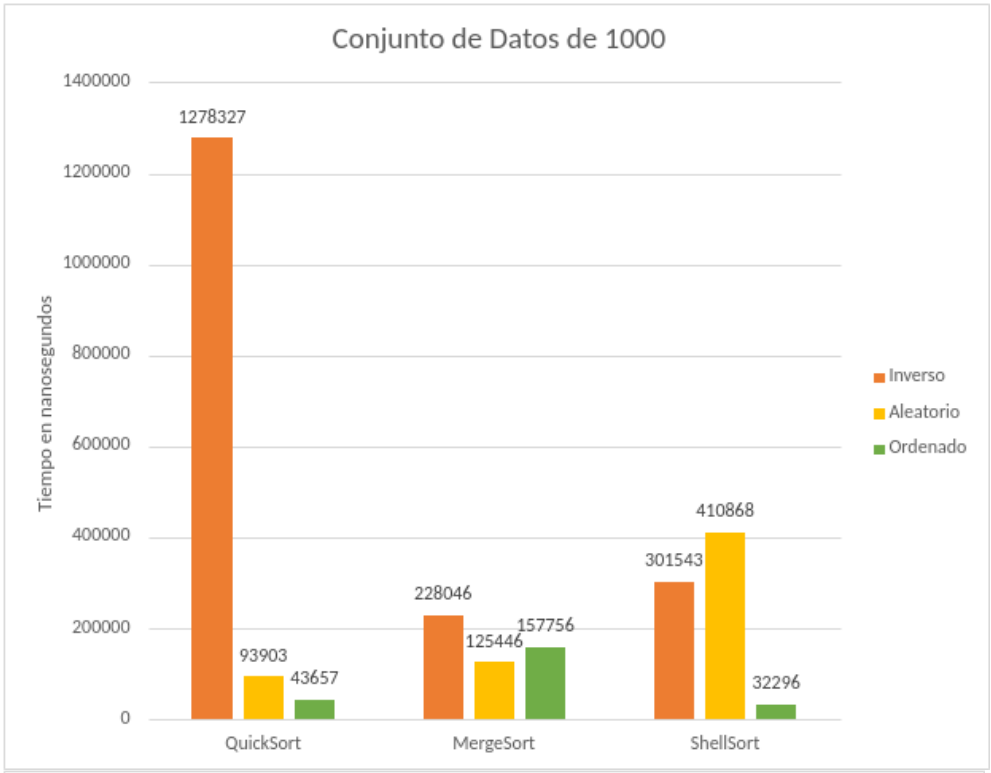
Usar el primer o último elemento del vector: funciona bien si la entrada es completamente aleatoria, pero muy deficiente si la entrada esta ordenada ($O(n^2)$)

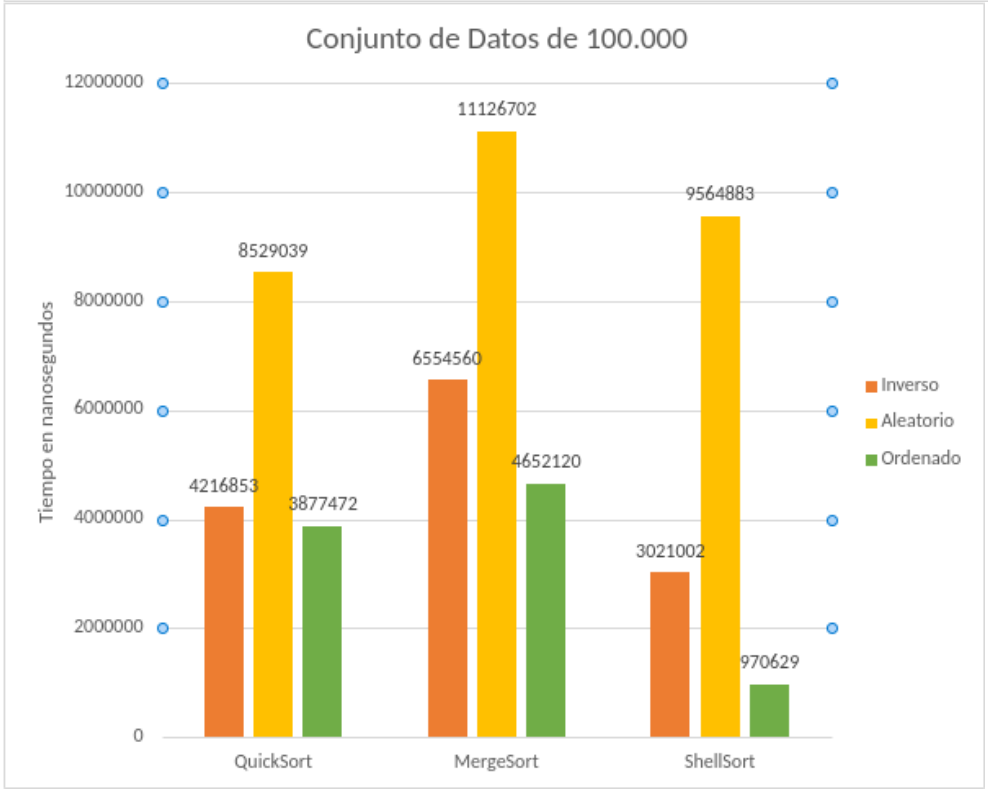
Pivote aleatorio de la lista: no requiere ningún calculo adicional lo cual lo hace más rapido. Pero puede provocare que el algoritmo aumente su orden de cálculo para ciertas permutaciones de elementos de la lista.

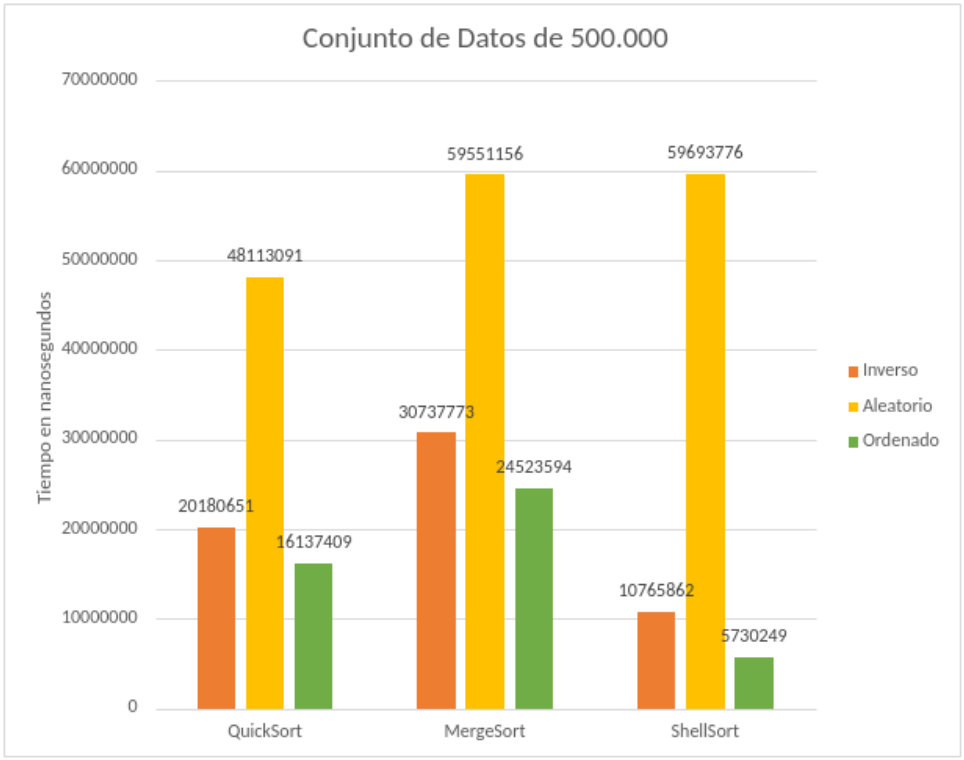
Mediana de 3: se calcula la mediana de 3 elementos del vector, eligiendo el valor del medio como pivote. Una opción más balanceada.

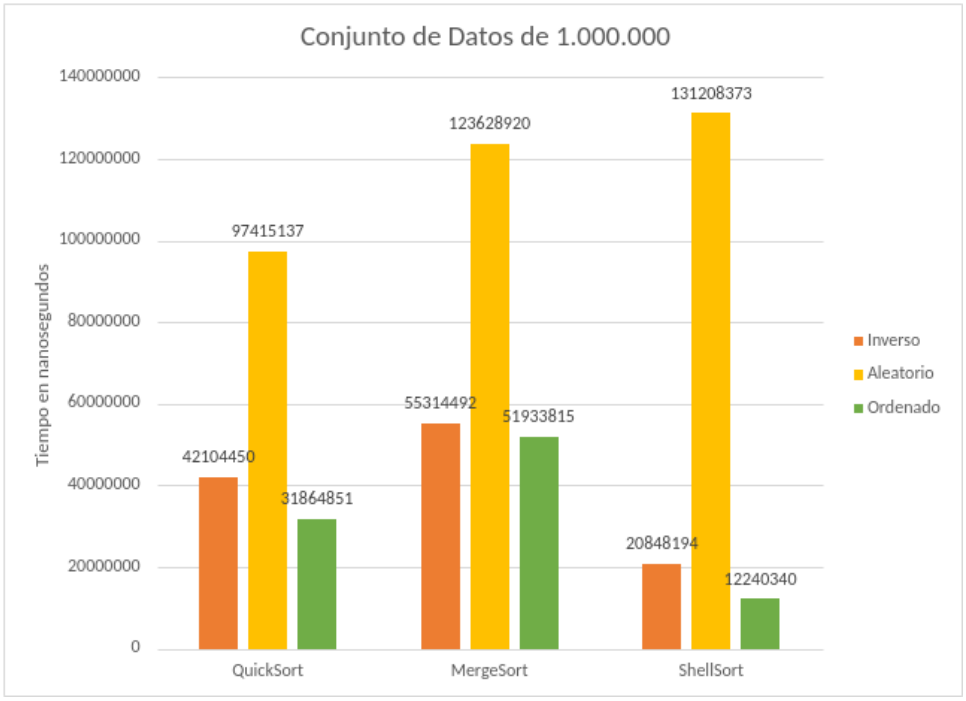
Este algoritmo posee complejidad computacional de $O(N \log (N))$

Resultado de las mediciones.









Con esta información se puede llegar a las siguientes conclusiones empíricas.

QuickSort puede ser considerablemente pesado para listas de muy pocos elementos en orden inverso. Por el alto overhead causado por cada partición. Aunque termina siendo el mejor en todos los casos al ordenar listas de elementos aleatorios.

ShellSort termina siendo consideramente mejor que el resto de los algoritmos al tener listas de elementos ordenados e inversos. Debido a que es sensible cuando el resto no lo es.

MergeSort es que el peor rendemiento posee al ordenar listas de elementos inversos al tener que realizar operaciones de ordenamientos en cada sublista.