|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Nombre** | **Complejidad** | **Mejor caso** | **Complejidad** | **Peor caso** | **In** | **A** | **Es** |
| **ShellSort** | O(nlog(n)) | El mejor caso sucede cuando los datos están organizados en gran parte, dado que no tendrá que hacer swap de datos tantas veces. | O(n^2) | El peor caso sucede cuando los datos están completamente desorganizados, porque tiene que hacer muchas comparaciones y accesos en el arreglo | **X** | **X** |  |
| **MergeSort** | O(n log(n)) | La complejidad es la misma para ambos casos, dado que accede a todos valores del arreglo, o los arreglos y los compara entre sí | O(n log(n)) | La complejidad es la misma para ambos casos, dado que accede a todos valores del arreglo, o los arreglos y los compara entre sí |  |  | **X** |
| **QuickSort** | O(nlogn) | El mejor caso sucede cuando todos los elementos del arreglo están desorganizados | O(n^2) | El peor caso sucede cuando hay valores del arreglo que son únicos | **X** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **ShellSort** | **MergeSort** | **QuickSort** |
| **Prueba 1** | 514ms | 287ms | 306ms |
| **Prueba 2** | 453ms | 286ms | 291ms |
| **Prueba 3** | 457ms | 285ms | 306ms |
| **Promedio** | 474ms | 286ms | 301ms |

## Análisis

Se puede llegar a la conclusión de que el algoritmo **MergeSort** fue el más eficiente durante el ordenamiento del arreglo, obteniendo el mejor tiempo promedio en ejecución. Esto debe ser porque la información debe estar desorganizada, dejando al ShellSort en su peor caso. A su vez, también debe contener algunos valores repetidos, ya que el segundo algoritmo más rápido es el Quicksort, en cuyo caso su complejidad se ve aumentada.

Siendo el orden de más eficiente a menos eficiente:

1. MergeSort
2. QuickSort
3. ShellSort