José Pablo Donado López

Hoja de trabajo No. 3

Realizar: Programa para usar algoritmos de ordenamiento (sort).
Realizarse: Individual**.**

Objetivos:
a. Comparación de la complejidad de tiempo de corrida de algoritmos de sort.
b. Utilización de Profilers para medición de tiempos de corrida.
c. Control de versiones del programa.
d. Emplear pruebas unitarias para validar la efectividad de los algoritmos en el lenguaje de programación Java

Programa a realizar:
Implementar los algoritmos de la siguiente lista en su programa.

1. Gnome sort
2. Merge sort
3. Quick sort
4. Radix Sort
5. Selection Sort
6. Shell Sort
7. Heap Sort

Puede utilizar como referencia el código que se encuentra en el libro **Java Structures, de *Duane A. Bailey*** que usamos en el curso, puede utilizar los algoritmos creados por sus compañeros en clase u otro similar. Recuerde siempre incluir la referencia a la fuente que utilizó, si se creó el algoritmo con IA, colocar la pregunta que se le hizo a la IA para obtener el algoritmo

Es interesante también la información sobre sorts del siguiente sitio: http://panthema.net/2013/sound-of-sorting/

Desarrolle un programa que genere al azar y guarde en un archivo, números enteros. Debe generar hasta 3000 números, los números son enteros entre 0 y 10000.

Luego utilice su programa para leer los datos de ese archivo, guardarlos en un arreglo y ordenarlos, usando cada uno de los algoritmos de sort. Utilice el Profiler VisualVM (https://visualvm.github.io/idesupport.html) para medir el tiempo empleado en el ordenamiento para: 10 números hasta 3000 números. El propósito será generar una gráfica con el eje Y mostrando el tiempo y el eje X mostrando la cantidad de números ordenados. Usted puede seleccionar los intervalos que considere adecuados en el eje X.

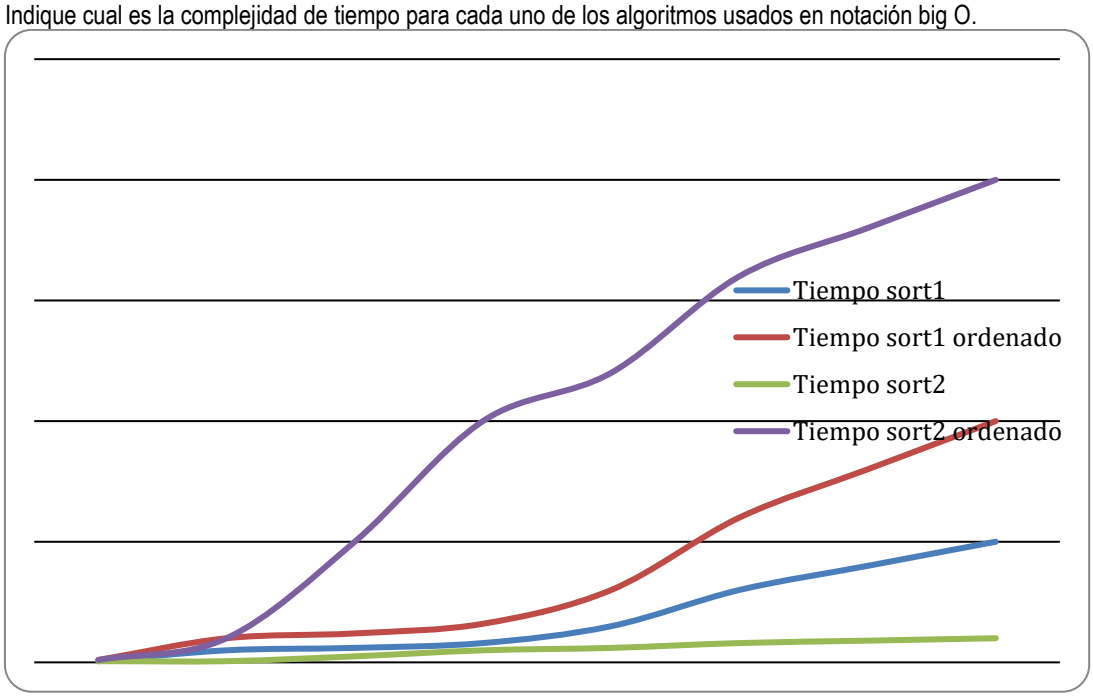
El usuario deberá poder escoger si desea ordenar los datos de forma ascendente o de forma descendente,

Deberá correr el programa también para calcular el tiempo que lleva ordenar con su algoritmo una colección de datos YA ordenada. Esto quiere decir que luego de ordenar los datos del arreglo, debe correr nuevamente su programa con esos datos y medir la cantidad de tiempo utilizada para este nuevo ordenamiento, también deberá medir el tiempo cuando el arreglo está ordenado de forma ascendente y se pide ordenar de forma descendente.

Tareas:

a. Realizar una gráfica (en Excel u otro programa similar) de los tiempos de corrida de sus algoritmos. Ver ejemplo de gráfica a continuación:

NOTA: aunque en la gráfica se muestran solamente dos sorts, en la que su grupo elabore deben estar representados los seis algoritmos de sort.
NOTA: incluya también en la gráfica el rendimiento TEÓRICO esperado, de acuerdo la complejidad de tiempo del algoritmo usado O( ). Indique cual es la complejidad de tiempo para cada uno de los algoritmos usados en notación big O.



b. Debe dejar evidencia de todo el desarrollo en el repositorio de git (o sistema similar). Indicar cómo acceder a su repositorio y si es necesario, agregar a su catedrático y auxiliar para que tengan acceso al mismo.
c. Un documento en Word o PDF que explique que profiler utilizó, como lo empleó y los resultados que se obtuvo en cada algoritmo de sort. Incluya la gráfica obtenida.
d. Incluya pruebas unitarias para cada uno de los algoritmos de sort implementados.

Debe subir a Canvas todos los productos elaborados en los incisos a, c, d y los enlaces a su repositorio de github (o similar).

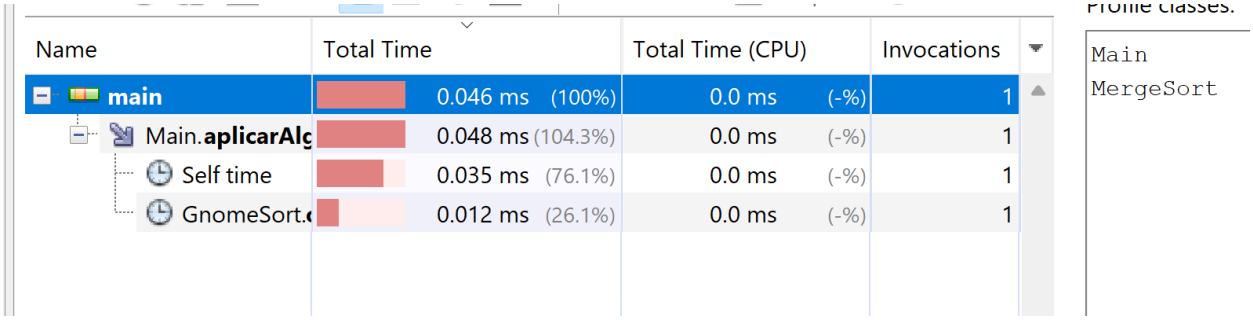
Calificación: Debe utilizarse un profiler para que se califique su programa.

Aspecto	Puntos
Estilo de codificación: comentarios, identación, nombres de variables significativas.	4
Implementación de los sorts (3 puntos cada uno)	21
Uso del repositorio: existen más de cinco versiones guardadas, la última versión es igual a la colocada en el Blackboard	10
Medición del tiempo de corrida usando un Profiler (5 puntos por generación de los valores de forma aleatoria, 5 puntos por ordenar la estructura desordenada, 5 puntos por ordenar de forma ascendente a descendente, 5 puntos de explicación de uso de profiler)	20

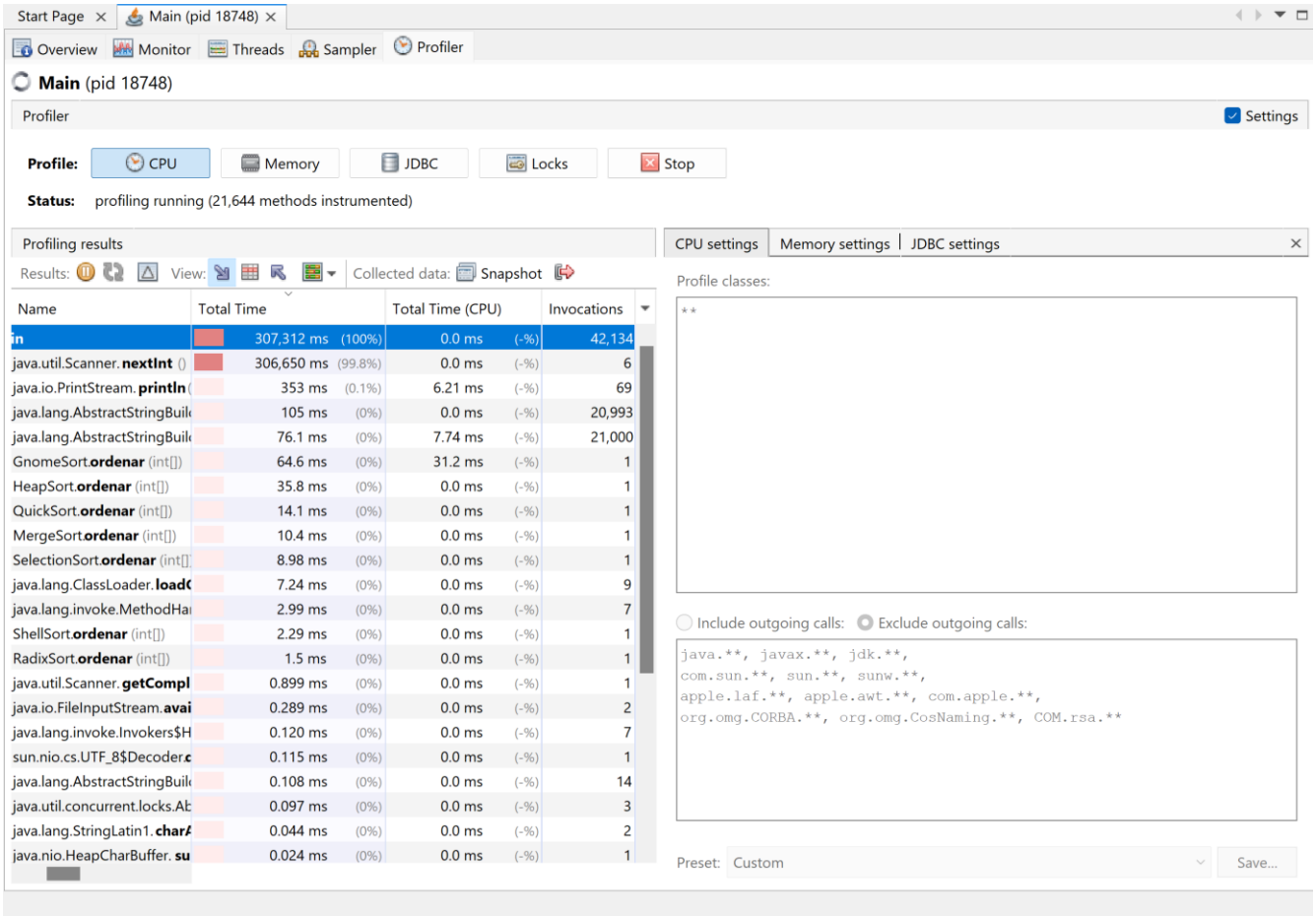
Gráfica con tiempos de corrida (5 puntos de explicación de la complejidad teórica de cada algoritmo, 10 puntos de obtención y gráfica de tiempo de los profilers, 10 puntos por cálculo de los tiempos teóricos)	25
Pruebas Unitarias para los sorts	20
TOTAL:	100

Sección C: Profiler y Resultados de Eficiencia
Un documento en Word o PDF que explique que profiler utilizó, como lo empleó y los resultados que se obtuvo en cada algoritmo de sort. Incluya la gráfica obtenida.

La prueba piloto para la clase de GnomeSort con 10 números aleatorios se muestra en la figura contigua:



Una vez se confirmó que el visualizador cumplió con su acometido, se procedió a hacer el *profiling* de cada clase de tipo Algoritmo con un parámetro de 3000 elementos para ordenar. El resultado es el siguiente:

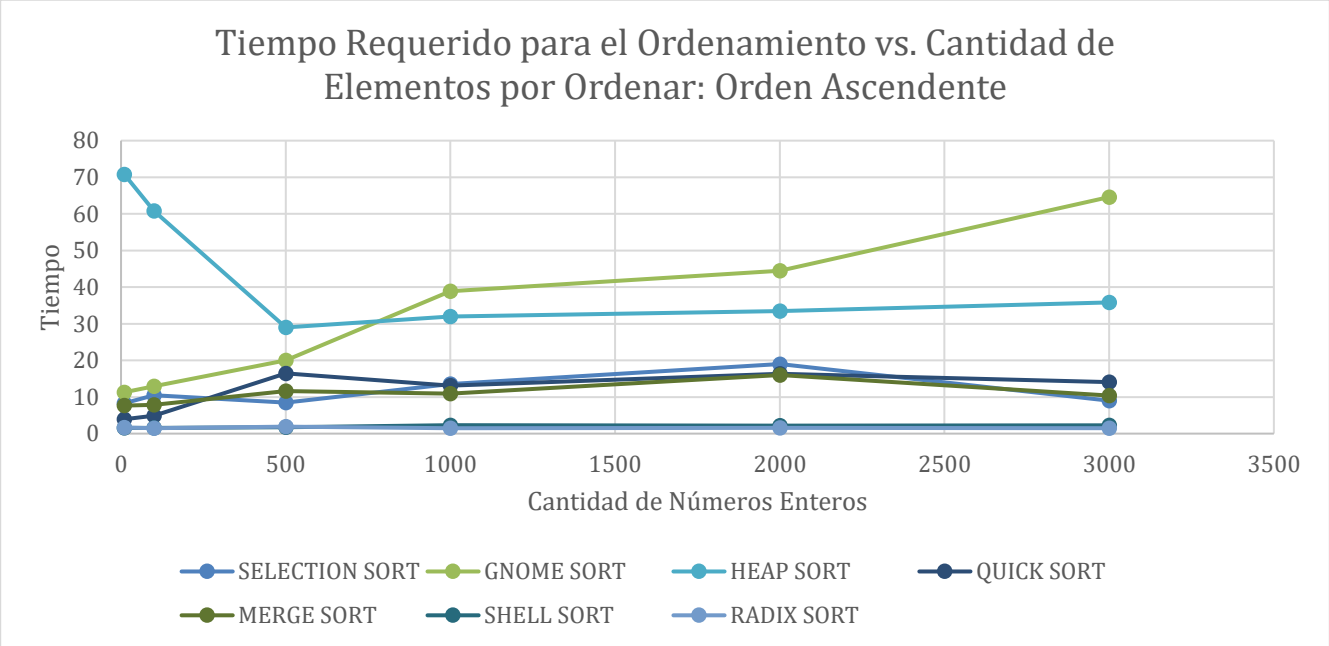


Como era de esperarse, la clase principal es la que más tarda en ejecutarse, pues el diseño no es el óptimo y tampoco cuenta con patrones de diseño *Factory*. Sin embargo, esto no comprende el objetivo de la Hoja de Trabajo, por lo que se considera irrelevante en este momento. El tiempo (dado en milisegundos) que duró cada algoritmo en efectuar el ordenamiento se registró en un cuadro de Excel.

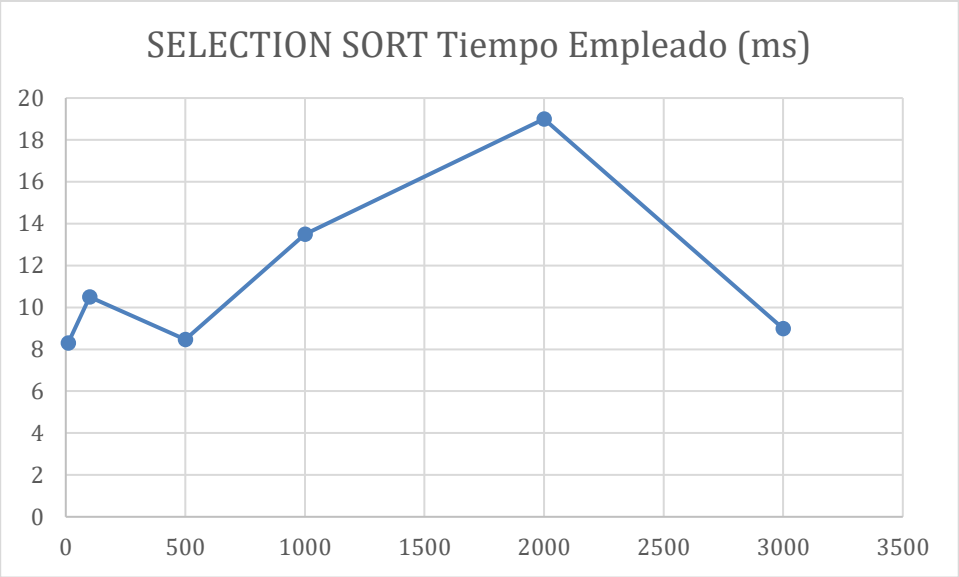
Por otra parte, se tiene que para ordenar 10 enteros se requiere un tiempo total de:

Name	Total Time		Total Time (CPU)		Invocations
main	105 ms	(100%)	43.1 ms	(100%)	7
Main.**aplicarAlg**	105 ms	(100%)	43.1 ms	(100%)	7
HeapSort.**or**	70.8 ms	(67.1%)	43.1 ms	(100.1%)	1
GnomeSort.****	11.3 ms	(10.8%)	0.0 ms	(0%)	1
SelectionSor	8.30 ms	(7.9%)	0.0 ms	(0%)	1
MergeSort.**o**	7.65 ms	(7.2%)	6.24 ms	(14.5%)	1
QuickSort.**or**	3.93 ms	(3.7%)	0.0 ms	(0%)	1
RadixSort.**or**	1.67 ms	(1.6%)	0.0 ms	(0%)	1
ShellSort.**orc**	1.58 ms	(1.5%)	0.0 ms	(0%)	1
Self time	0.174 ms	(0.2%)	0.0 ms	(0%)	7

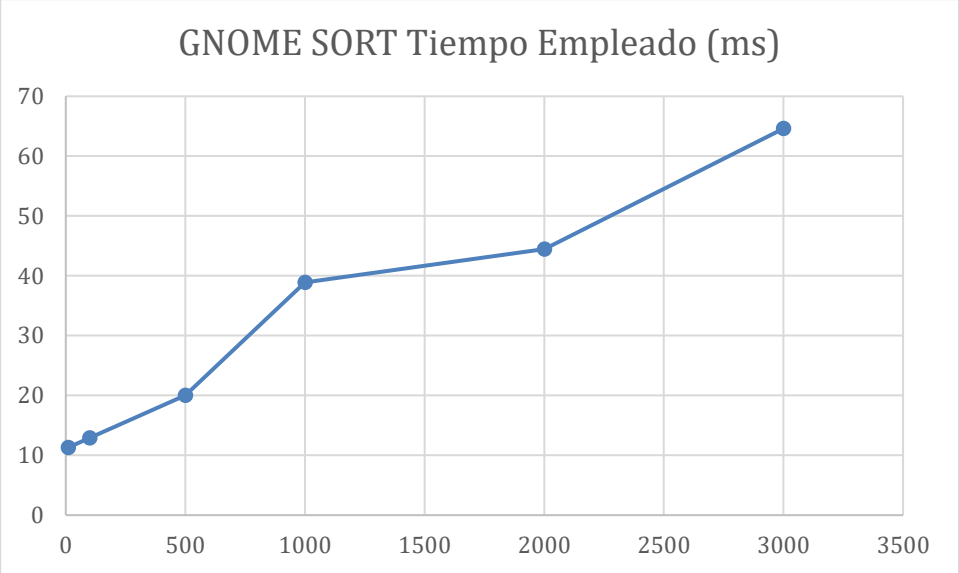
De forma gráfica, estos resultados se ven así:



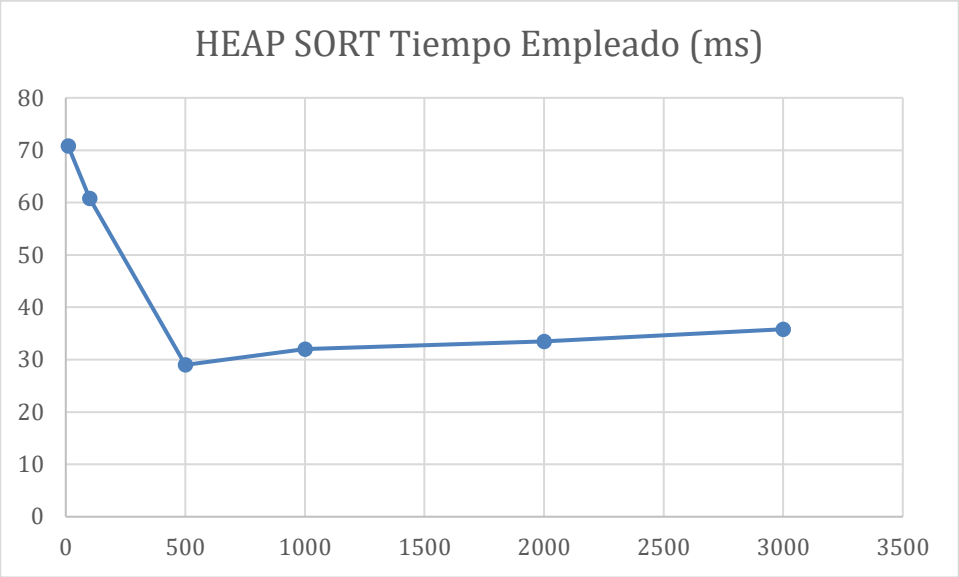
Si le damos un vistazo a solo tres algoritmos podemos notar que su comportamiento es considerablemente distinto.
En el primer caso, el Selection Sort:



El caso menos eficiente fue con los 2000 tokens.

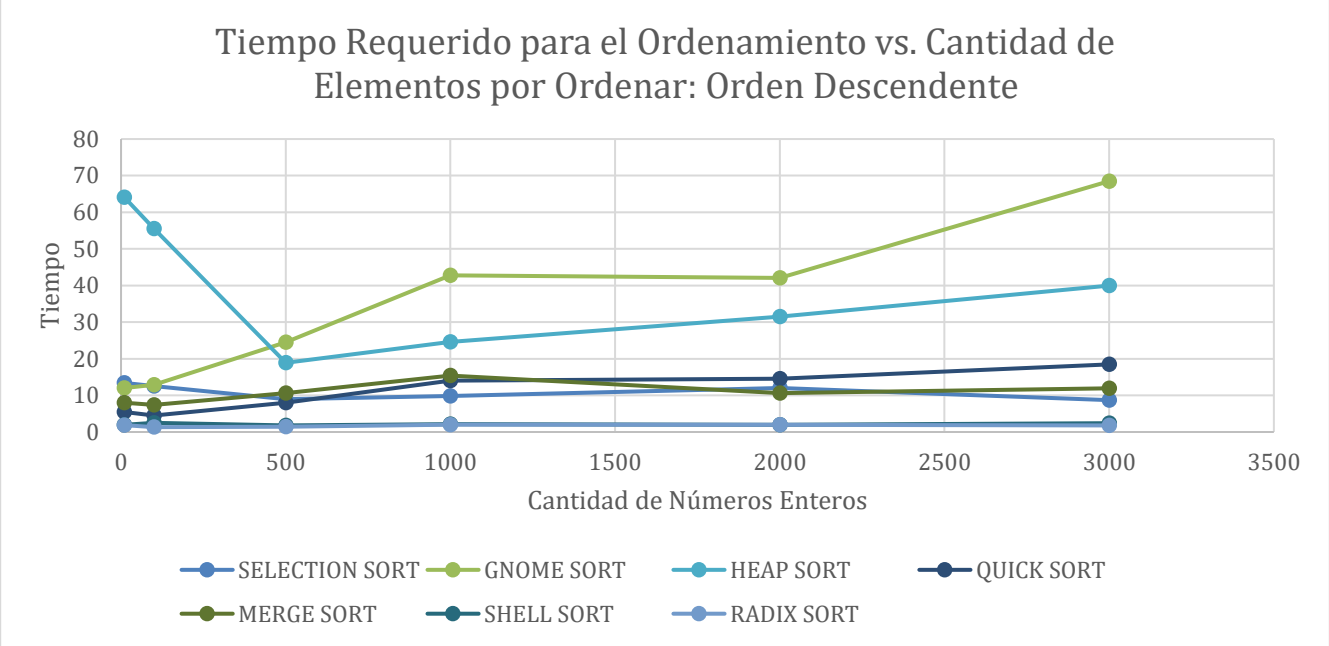


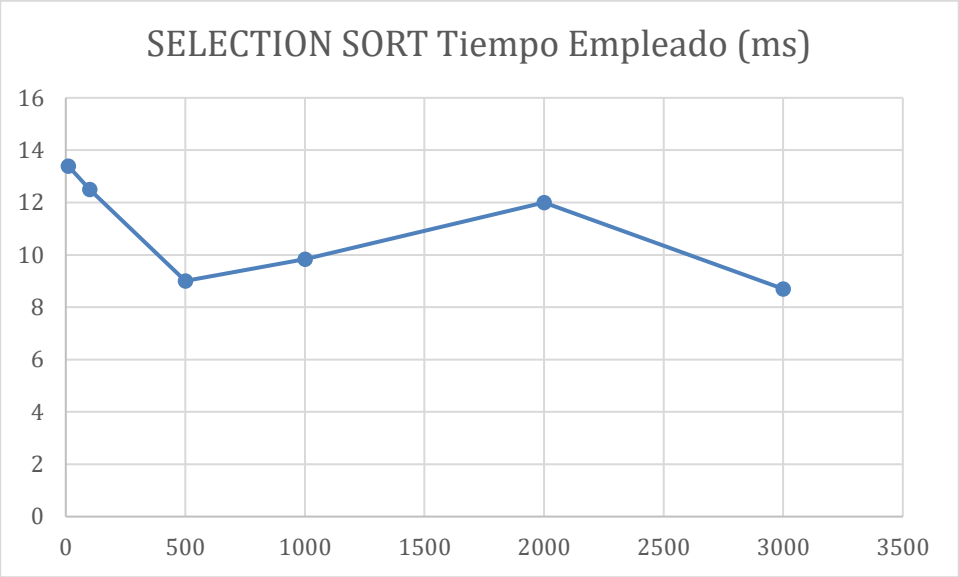
En cambio, el Gnome Sort se agravaba conforme aumentaba la cantidad de datos. Por algo se le llama stupid sort.



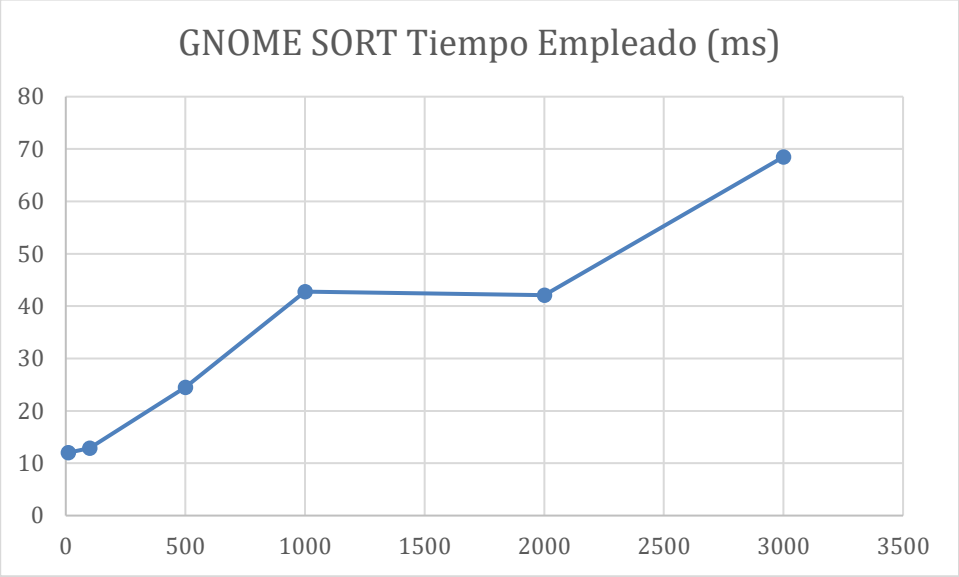
A diferencia de los otros dos algoritmos, el Heap Sort se comporta bastante estable después de los 500 datos.

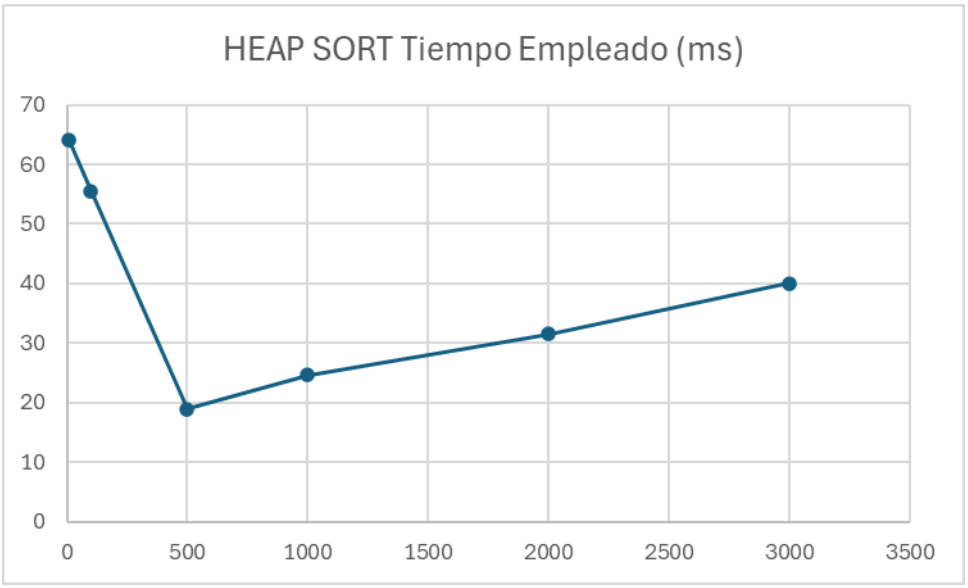
En la segunda hoja de cálculo del documento de Excel adjunto (véase el repositorio) se realizaron las mismas gráficas para el ordenamiento de manera descendente. Como era de esperarse, los resultados son similares, pero el tiempo total aumentó para la mayoría de casos.



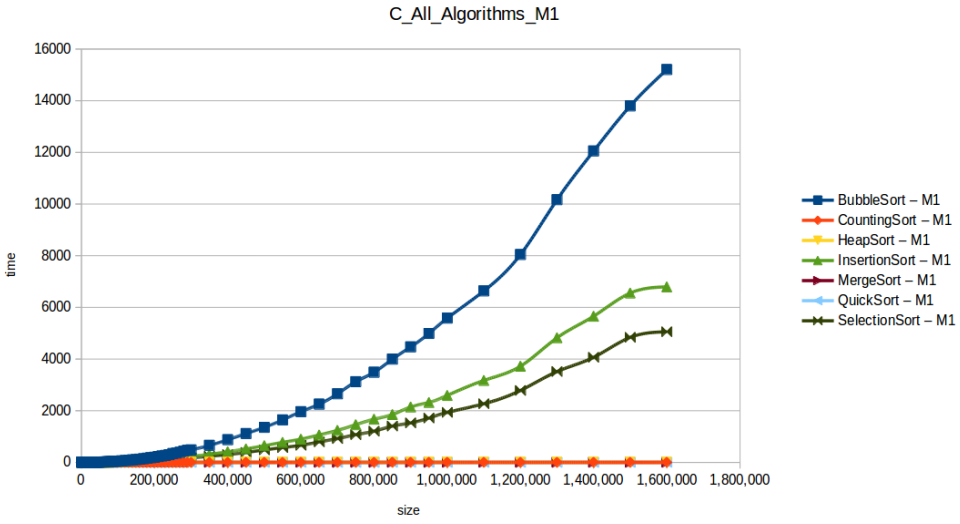


Se reiteran los tres casos pero con el ordenamiento invertido.





El rendimiento teórico está dado por el siguiente gráfico:



El cual se basa en las funciones del siguiente cuadro, el cual muestra el mejor caso y el peor para cada algoritmo:

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)