Universidad del valle de Guatemala
Algoritmos y estructura de datos sección 20

Jose Pinto 25063
Valeria Hernandez 25086

Hoja de trabajo 3

Introducción:

Para esta hoja de trabajo se implementaron cinco algoritmos de ordenamiento: Gnome Sort, Merge Sort, Quick Sort, Radix Sort, e Insertion Sort por elección nuestra. El objetivo fue comparar el rendimiento práctico de cada algoritmo mediante la medición de tiempos de ejecución utilizando el profiler de IntelliJ, y contrastar dichos resultados con el rendimiento teórico esperado según su complejidad en notación Big-O. Se realizaron pruebas con tamaños de entrada desde 10 hasta 3000 elementos, utilizando tanto datos desordenados (escenario promedio) como datos previamente ordenados (mejor escenario).

Profiler utilizado:

Se utilizó el Profiler de IntelliJ IDEA para medir el tiempo de ejecución de cada sort, siguiendo el procedimiento descrito a continuación:

1. Se ejecutó el programa generando arreglos de tamaños 10, 100, 500, 100, 2000 y 3000
2. Se midió el tiempo de ejecución del método sort para cada algoritmo
3. Se realizaron pruebas tanto con datos desordenados como con datos previamente ordenados
4. Se registraron los tiempos en milisegundos

En algunos casos fue difícil medir los tiempos, especialmente cuando la cantidad de datos era pequeña, porque la ejecución era extremadamente rápida.

Notas adicionales:

A continuación se presentan aclaraciones y decisiones de diseño relevantes tomadas durante la implementación:

- Los algoritmos Gnome Sort, Insertion Sort, Merge Sort y Quick Sort se implementaron de forma genérica utilizando T extends Comparable<T>, ya que requieren comparar elementos entre sí.
- Radix Sort no es un algoritmo comparativo, por lo que no puede implementarse utilizando Comparable. En su lugar, trabaja directamente con valores numéricos y sus dígitos.
- Para la ejecución de Radix Sort, fue necesario utilizar un arreglo de tipo int[], ya que el algoritmo realiza operaciones aritméticas sobre los valores.
- El arreglo principal se maneja como Integer[] para permitir el uso de métodos genéricos basados en Comparable y facilitar la reutilización del mismo conjunto de datos entre los distintos algoritmos.
- Antes de aplicar cada algoritmo de ordenamiento, el arreglo original fue clonado, garantizando que todos los algoritmos trabajaran con los

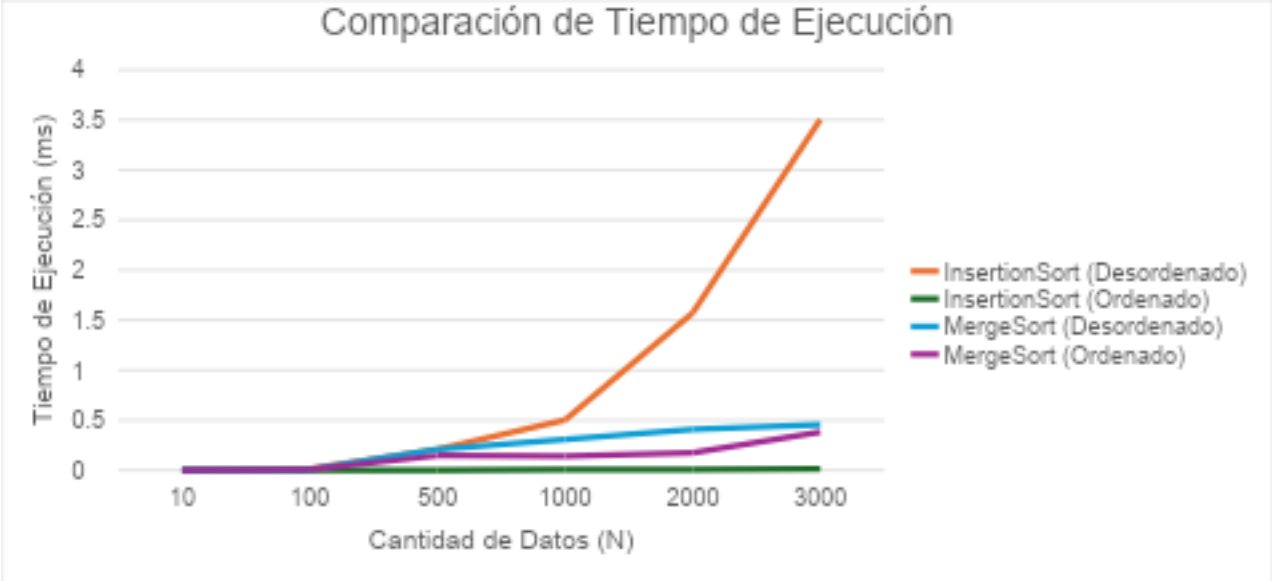
mismos datos iniciales.

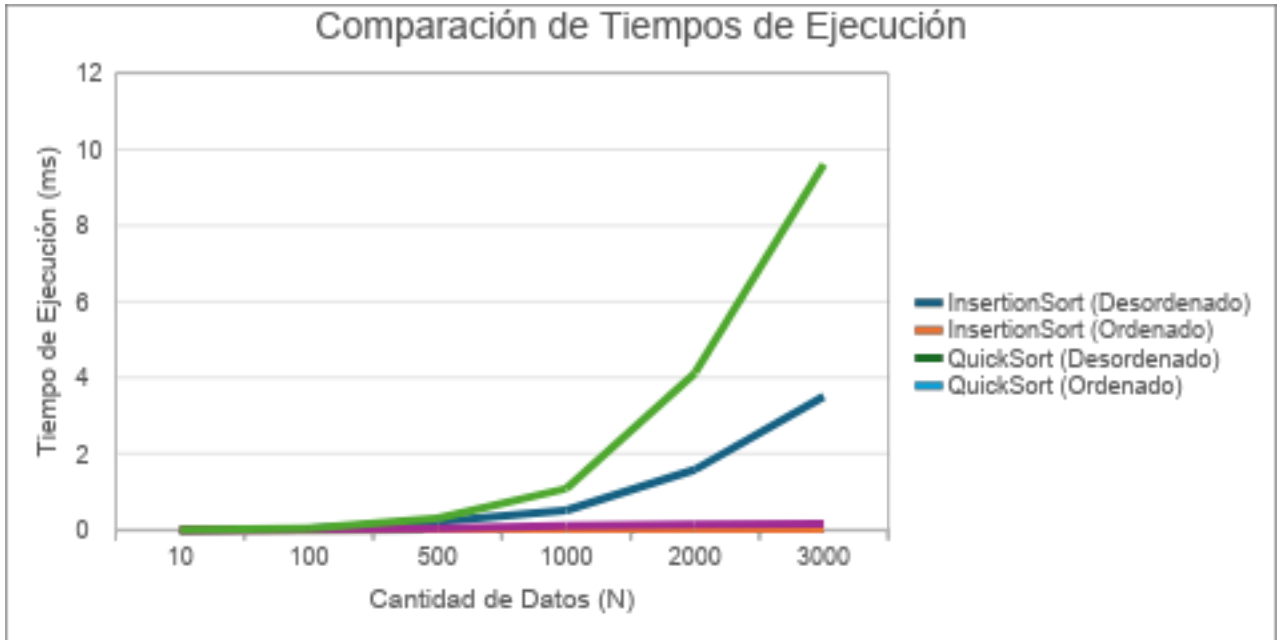
Resultados Obtenidos en cada Algoritmo

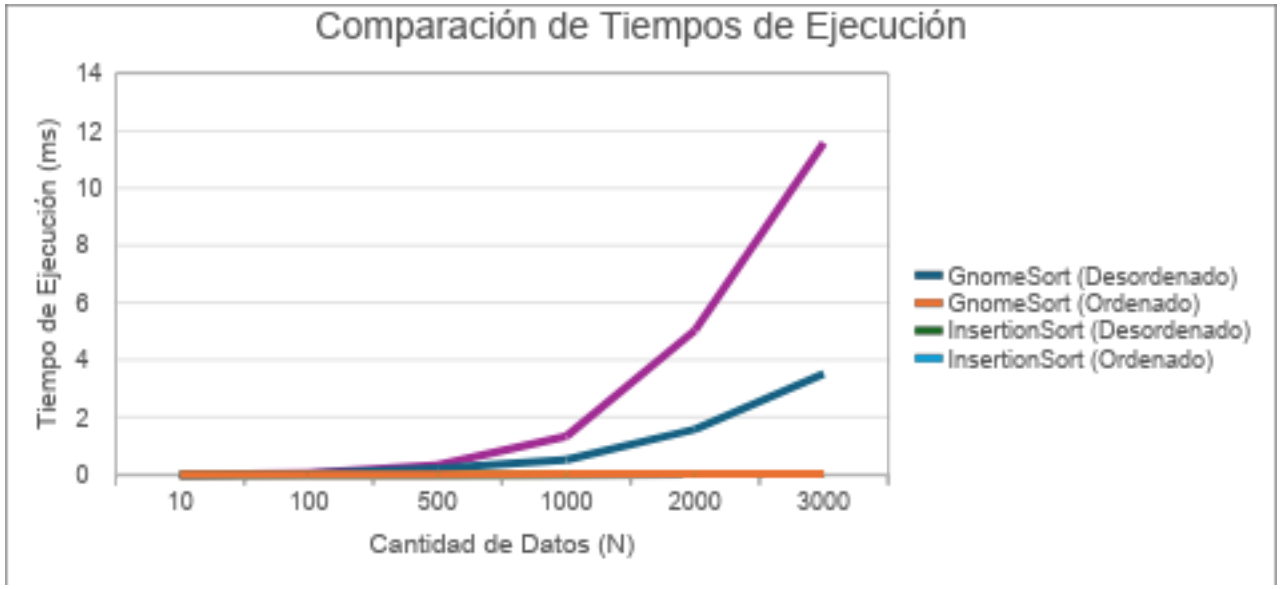
- **Gnome Sort:**
  - Cuando los datos estaban desordenados se notó claramente su crecimiento cuadrático $O(n^2)$. A medida que aumentaba la cantidad de elementos, el tiempo crecía bastante rápido. Con 3000 datos desordenados fue el más lento, alcanzando 11.57 ms, lo que confirma que no es eficiente para arreglos grandes.
  - Cuando los datos ya estaban ordenados, el tiempo bajó muchísimo a 0.0075 ms. En este caso el algoritmo solo recorre el arreglo una vez para verificar que todo está en orden, por lo que se comporta de forma lineal O(n).
- **Insertion Sort:**
  - Con datos desordenados mostró un comportamiento muy similar a Gnome Sort, creciendo de manera cuadrática $O(n^2)$. Para 3000 elementos el tiempo fue de 3.50 ms, lo cual demuestra que tampoco es ideal para cantidades grandes de elementos cuando el arreglo no está ordenado.
  - Fue el algoritmo más eficiente de los seleccionados cuando los datos ya estaban ordenados. El ciclo interno no necesitó ejecutarse, reduciendo el tiempo a 0.013 ms (con un comportamiento lineal).
- **Merge Sort:**
  - Merge Sort mostró un crecimiento más controlado, acorde a su complejidad O(n log n). Para 3000 datos desordenados tardó aproximadamente 0.45 ms, manteniendo un rendimiento eficiente comparado con los algoritmos cuadráticos.
  - Los tiempos se mantuvieron muy similares independientemente del estado de los datos (0.30 ms desordenado vs. 0.14 ms ordenado, para n=3000). Esto comprueba que el algoritmo siempre realiza las mismas divisiones y combinaciones, sin importar el orden inicial.
- **Quick Sort:**
  - Con datos desordenados fue muy rápido, ordenando 3000 elementos en 0.15 ms, lo que coincide con su comportamiento promedio O(n log n). Fue uno de los más eficientes en este escenario.
  - Al recibir los 3000 datos ya ordenados, el tiempo subió a 9.61 ms. Esto evidenció el peor caso teórico del algoritmo (O(n^2)), provocado porque se utilizó el último elemento como pivote, lo que generó particiones poco equilibradas cuando el arreglo ya estaba ordenado.
- **Radix Sort:**
  - Fue el algoritmo más rápido y más constante en general. Con 3000 datos desordenados tardó solo 0.05 ms, mostrando un crecimiento casi lineal.
  - Reflejó su complejidad O(nk) al ordenar 3000 datos en 0.05 ms (desordenado) y 0.06 ms (ordenado). Al no basarse en comparaciones directas entre elementos, sino en la distribución por dígitos numéricos, el orden inicial de los datos no tuvo un impacto significativo en su excelente rendimiento.

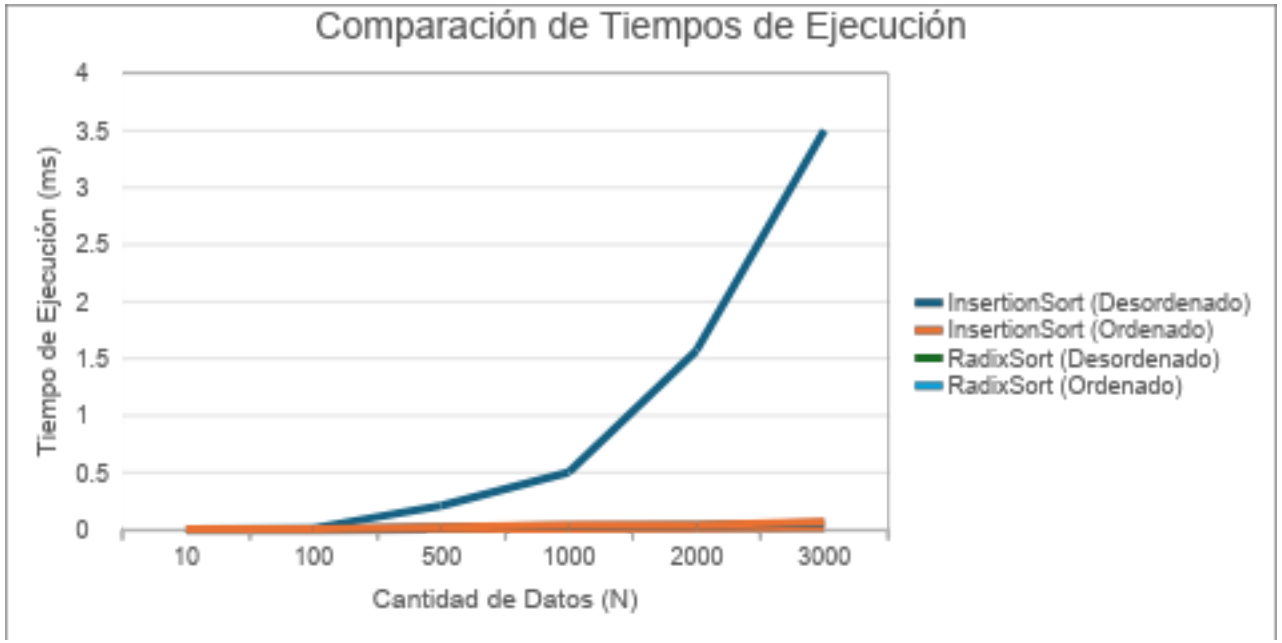
Resultados obtenidos:

Cantidad	GnomeSort (Desordenado)	GnomeSort (Ordenado)	InsertionSort (Desordenado)	InsertionSort (Ordenado)	MergeSort (Desordenado)	MergeSort (Ordenado)	QuickSort (Desordenado)	QuickSort (Ordenado)	RadixSort (Desordenado)	RadixSort (Ordenado)
10	0.0024	0.0006	0.0012	0.0003	0.0029	0.0024	0.0011	0.0024	0.0033	0.0009
100	0.0293	0.0009	0.0124	0.0006	0.0095	0.0081	0.0044	0.031	0.0063	0.006
500	0.3157	0.0015	0.2094	0.0026	0.2099	0.1445	0.0339	0.2963	0.0224	0.022
1000	1.3256	0.0034	0.5036	0.0045	0.3096	0.1416	0.1046	1.075	0.0311	0.0382
2000	5.0296	0.0069	1.5729	0.0092	0.4062	0.1739	0.1431	4.1249	0.0375	0.0365
3000	11.5799	0.0075	3.5037	0.013	0.453	0.3818	0.1561	9.6134	0.0567	0.0659



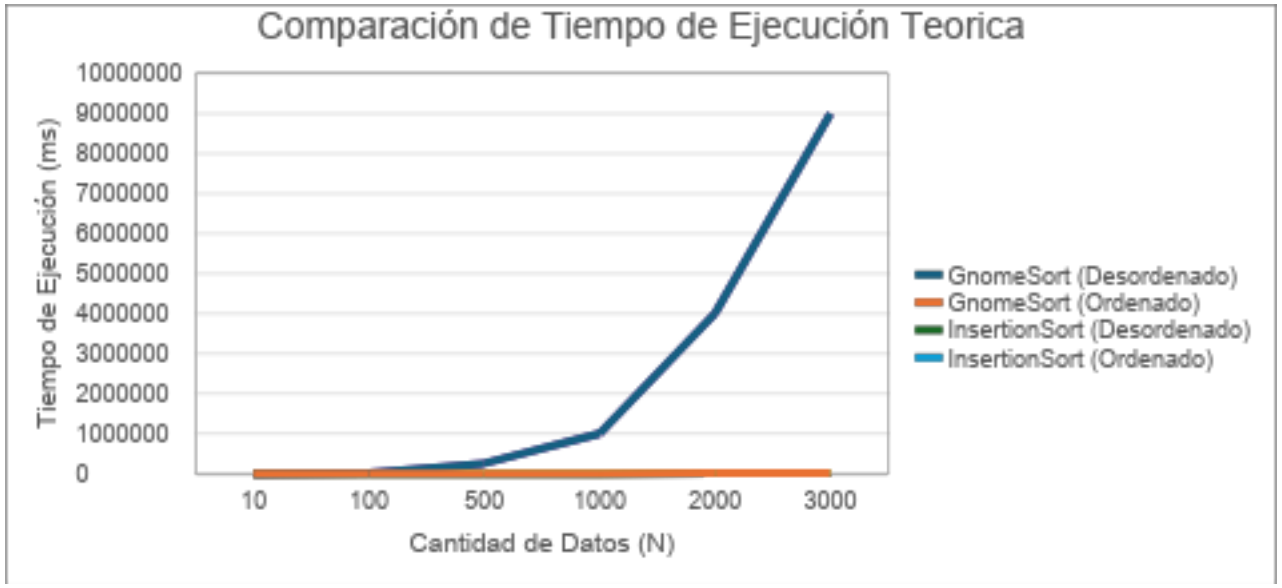

Comparación de Tiempos de Ejecución
Tiempo de Ejecución (ms)
12
10
8
6
4
2
0
10
100
500
1000
2000
3000
Cantidad de Datos (N)
InsertionSort (Desordenado)
InsertionSort (Ordenado)
QuickSort (Desordenado)
QuickSort (Ordenado)

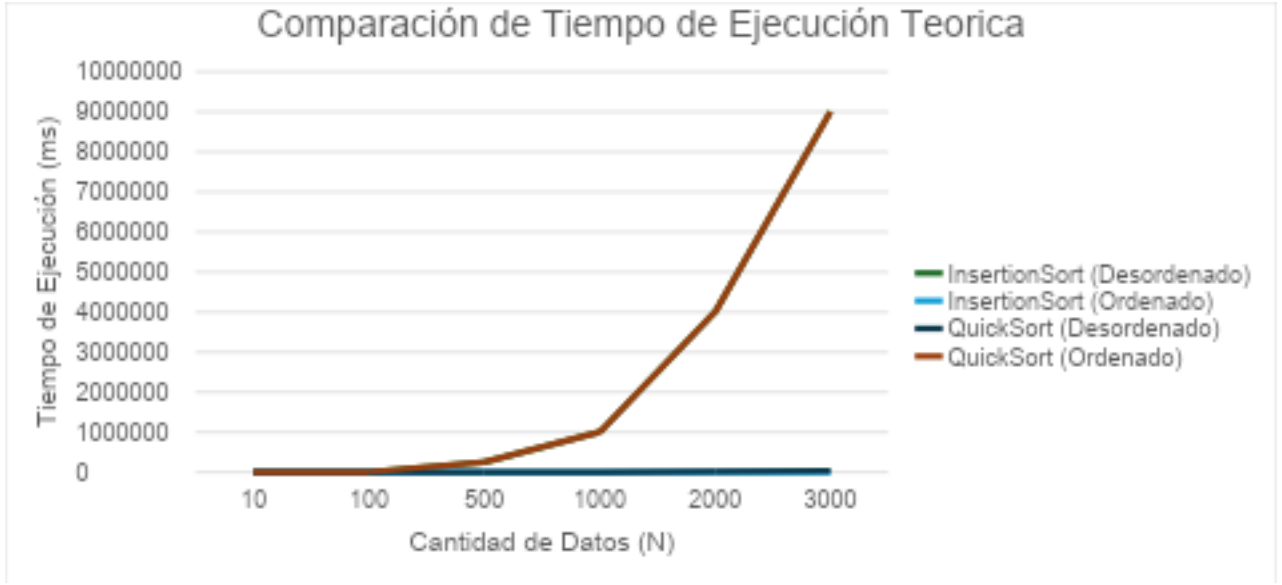

Comparación de Tiempos de Ejecución
Tiempo de Ejecución (ms)
14
12
10
8
6
4
2
0
10
100
500
1000
2000
3000
Cantidad de Datos (N)
GnomeSort (Desordenado)
GnomeSort (Ordenado)
InsertionSort (Desordenado)
InsertionSort (Ordenado)

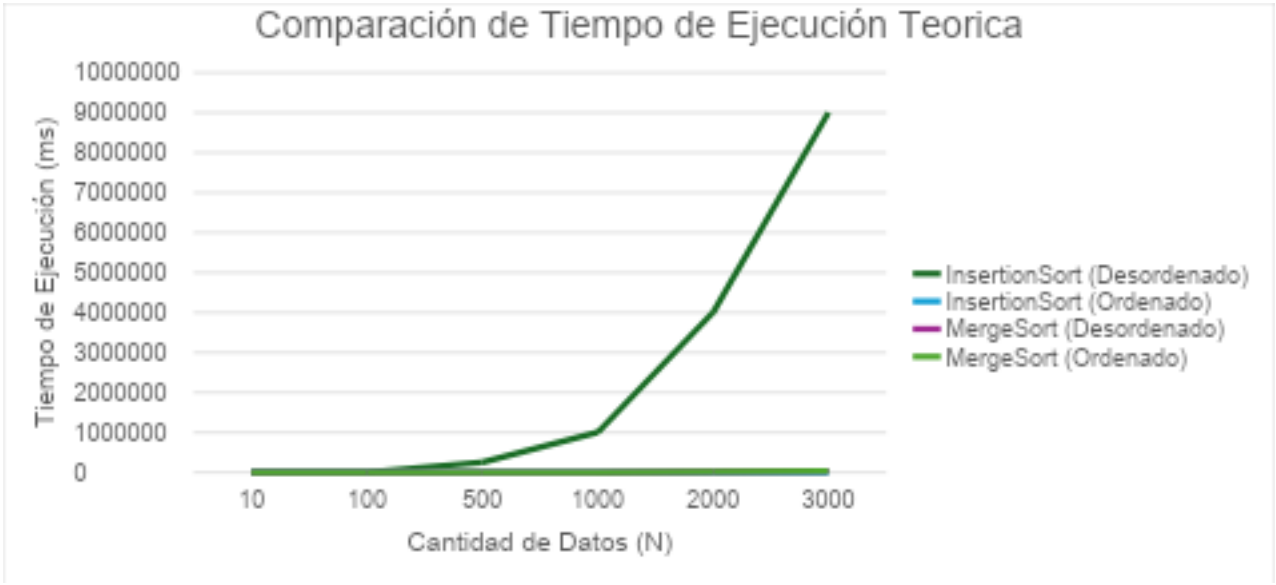


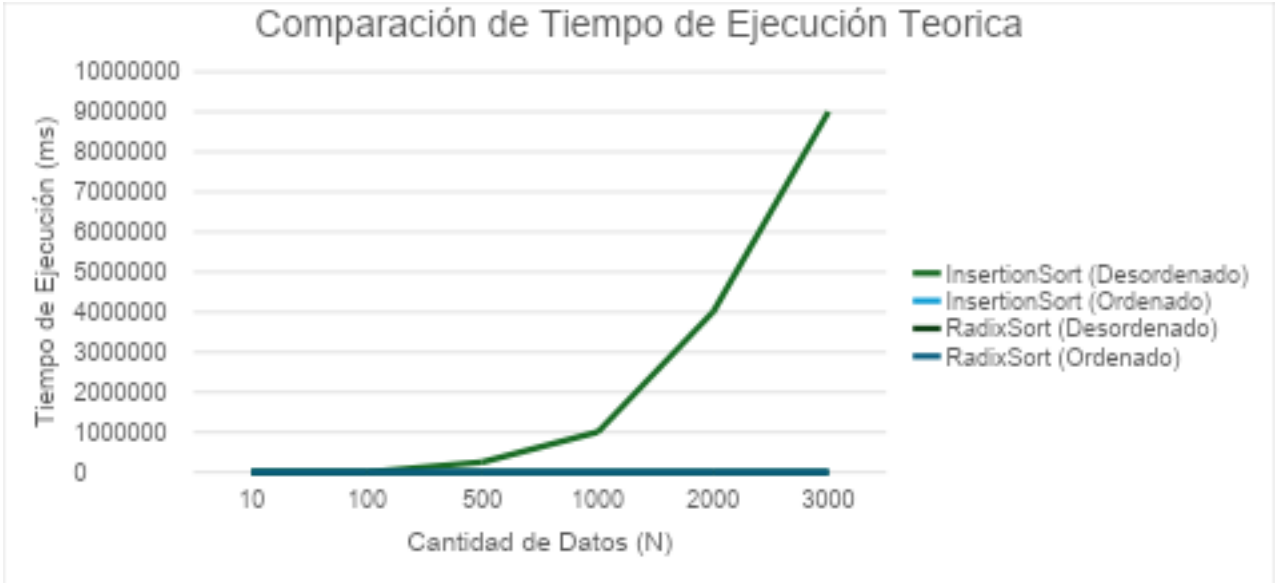
Resultados teóricos esperados:

Cantidad	GnomeSort (Desordenado)	GnomeSort (Ordenado)	InsertionSort (Desordenado)	InsertionSort (Ordenado)	MergeSort (Desordenado)	MergeSort (Ordenado)	QuickSort (Desordenado)	QuickSort (Ordenado)	RadixSort (Desordenado)	RadixSort (Ordenado)
10	100	10	100	10	33	33	33	100	10	10
100	10,000	100	10,000	100	664	664	664	10,000	100	100
500	250,000	500	250,000	500	4,482	4,482	4,482	250,000	500	500
1000	1,000,000	1000	1,000,000	1000	9,965	9,965	9,965	1,000,000	1000	1000
2000	4,000,000	2000	4,000,000	2000	21,931	21,931	21,931	4,000,000	2000	2000
3000	9,000,000	3000	9,000,000	3000	34,652	34,652	34,652	9,000,000	3000	3000


Comparación de Tiempo de Ejecución Teorica
Tiempo de Ejecución (ms)
10000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
10
100
500
1000
2000
3000
Cantidad de Datos (N)
GnomeSort (Desordenado)
GnomeSort (Ordenado)
InsertionSort (Desordenado)
InsertionSort (Ordenado)


Comparación de Tiempo de Ejecución Teorica
Tiempo de Ejecución (ms)
10000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
10
100
500
1000
2000
3000
Cantidad de Datos (N)
InsertionSort (Desordenado)
InsertionSort (Ordenado)
QuickSort (Desordenado)
QuickSort (Ordenado)


Comparación de Tiempo de Ejecución Teorica
Tiempo de Ejecución (ms)
10000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
10
100
500
1000
2000
3000
Cantidad de Datos (N)
InsertionSort (Desordenado)
InsertionSort (Ordenado)
MergeSort (Desordenado)
MergeSort (Ordenado)


Comparación de Tiempo de Ejecución Teorica
Tiempo de Ejecución (ms)
10000000
9000000
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
10
100
500
1000
2000
3000
Cantidad de Datos (N)
InsertionSort (Desordenado)
InsertionSort (Ordenado)
RadixSort (Desordenado)
RadixSort (Ordenado)