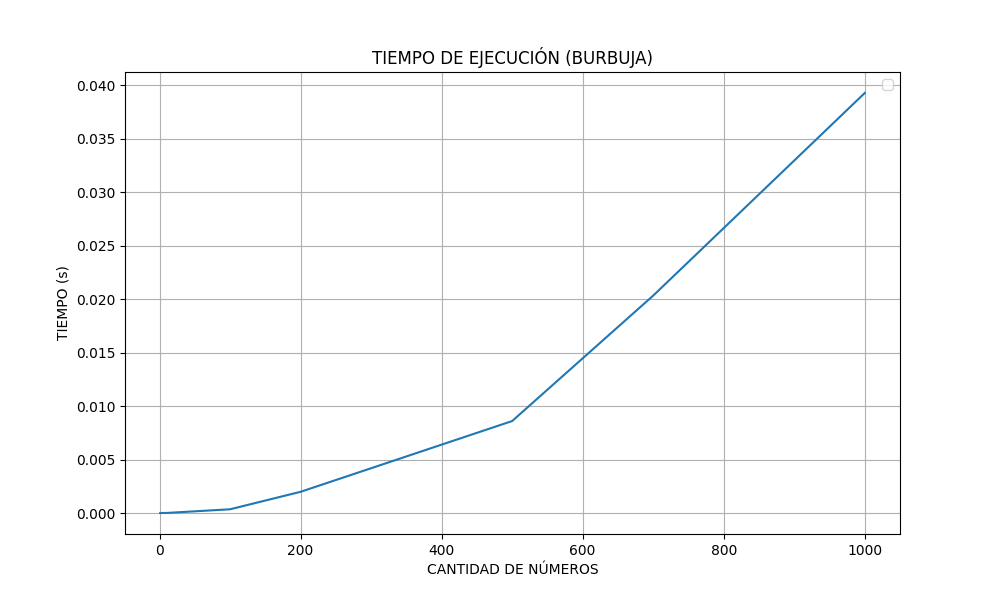
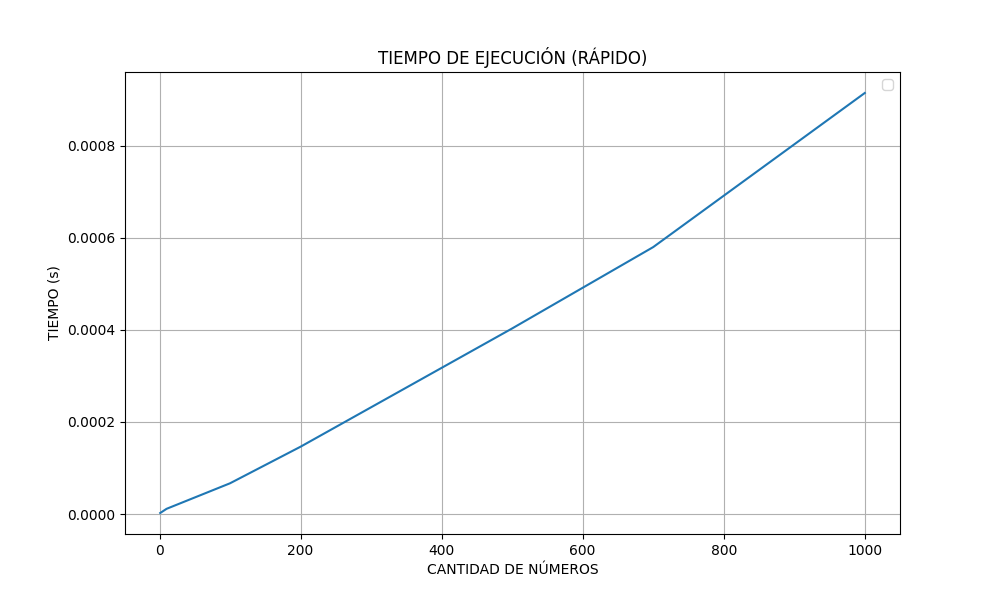
**ANALISIS A PRIORI**

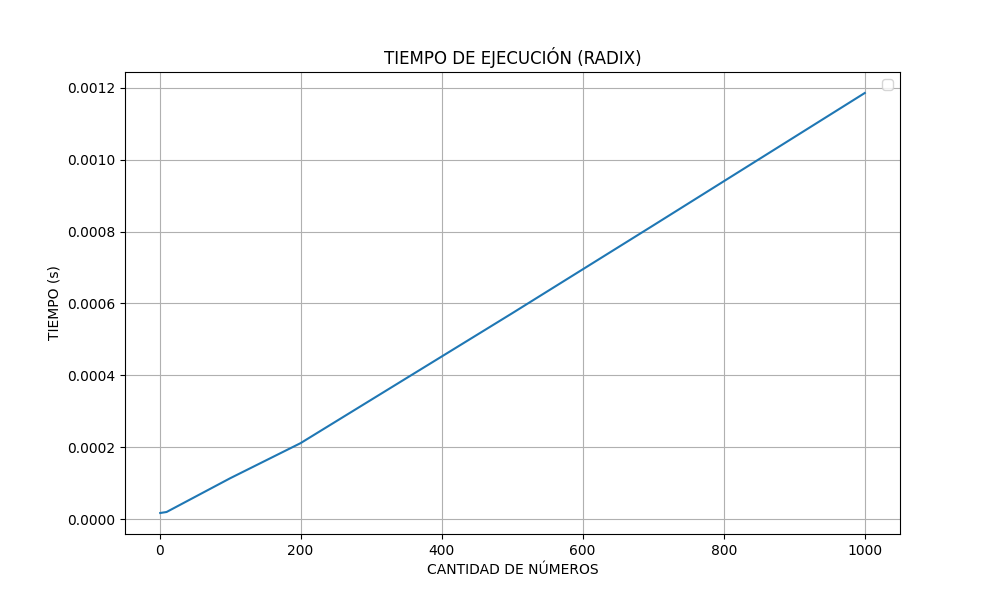
a) Ordenamiento Burbuja: dado que código tiene dos ciclos for anidados, concluimos que el orden de complejidad es . La gráfica realizada mediante código nos confirma que el orden de complejidad es



b) Ordenamiento Quicksort (rápido): realizando el mismo análisis que con el ordenamiento burbuja, podemos concluir que el orden de complejidad es , dado que tiene dos ciclos while anidados. La gráfica también nos confirma el análisis a priori.



c) Ordenamiento Radix (residuos): observando el código podemos determinar que el orden de complejidad depende de la cantidad de dígitos de los números de la lista y de la cantidad de números en la lista, por lo tanto, podemos concluir que el orden de complejidad es . La gráfica muestra un orden lineal de complejidad.

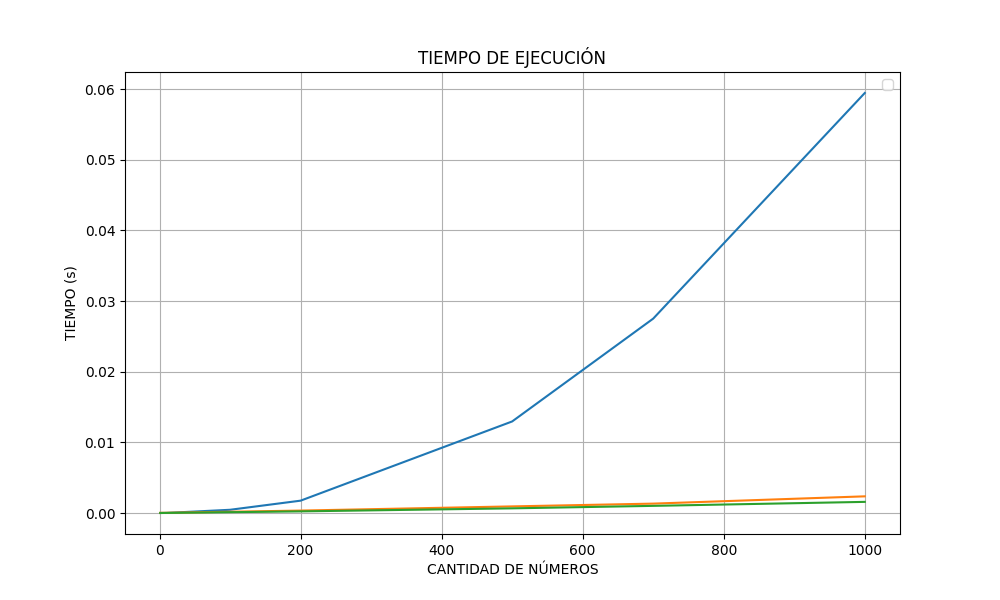


**GRÁFICA COMPARATIVA ENTRE LOS TRES TIPOS DE ORDENAMIENTOS**

Ordenamiento Burbuja.

Ordenamiento Quicksort.

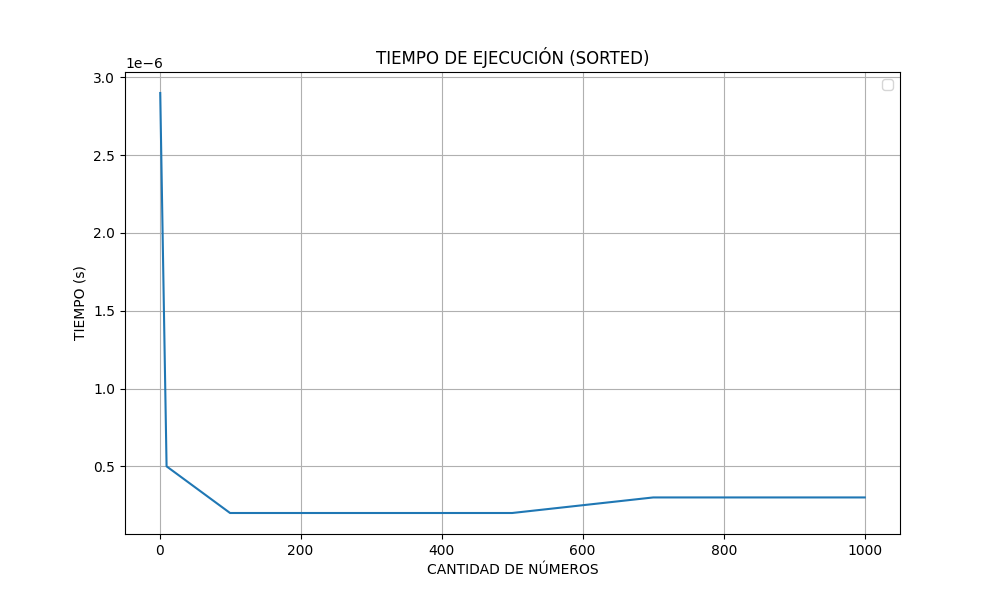
Ordenamiento Radix.



**FUNCIÓN BUILT-IN SORTED**

La función sorted funciona creando una lista nueva en donde muestra los elementos de manera ordenada, no opera sobre la lista original. Teniendo las gráficas de los cuatro algoritmos de ordenamiento, podemos observar que sorted es mucho más rápido que los demás. Creemos que se debe a que es una función built-in dentro del lenguaje Python, lo que hace menor el tiempo de ejecución.

|  |  |
| --- | --- |
| ALGORITMOS | COMPLEJIDAD |
| sorted | O(n) |
| Burbuja | O(n2) |
| Radix | O(kn) |
| Quicksort | O(n2) |



**INFORME**

El problema uno consistía en implementar algoritmos de ordenamiento en Python, los cuales eran:

1. Ordenamiento burbuja: opera comparando dos datos adyacentes y los va intercambiando si no están en orden. De esta manera, repite hasta que la lista quede ordenada completamente.
2. Ordenamiento Quicksort: establece un dato de la lista como pivote (en nuestro caso, primer dato) y luego compara los datos de la lista con el pivote y posiciona los menores a la izquierda y los mayores a la derecha. Posteriormente, divide en dos sublistas: la de la izquierda, con todos los datos menores al pivote y, la de la derecha, con los datos mayores al pivote. Procede a ordenar recursivamente las sublistas para después unirlas en una lista ya ordenada.
3. Ordenamiento radix: analiza el último digito de cada dato de la lista y en una sublista suma uno en cada índice que coincida con el último digito de cada dato. Suma los datos adyacentes de la sublista, obteniendo en la última posición la cantidad total de datos en la lista original. Con el último digito del último dato de la lista original busca el índice que coincida en la sublista y posiciona el dato de la lista original en una nueva lista en la posición en la cual el índice es igual al dato en la sublista, cuya posición sea igual al último digito del último dato de la lista original.

Repite esto la cantidad de veces necesarias según la cantidad de dígitos de los datos en la lista original.

En el siguiente link se puede ver gráficamente lo que va realizando un ordenamiento radix: <https://www.cs.usfca.edu/~galles/visualization/RadixSort.html>.

Luego de la implementación de cada ordenamiento, comprobamos el funcionamiento de los mismos mediante una lista de 500 números de cinco dígitos. Posteriormente, nos basamos en un código visto en clase para lograr medir los tiempos de ejecución de cada ordenamiento y graficarlos. También realizamos la gráfica correspondiente a la función sorted, junto con una breve explicación de su funcionamiento; para hacer una comparación con los algoritmos de ordenamiento implementados y así, corroborar el orden de complejidad y comparar la eficiencia de cada uno.