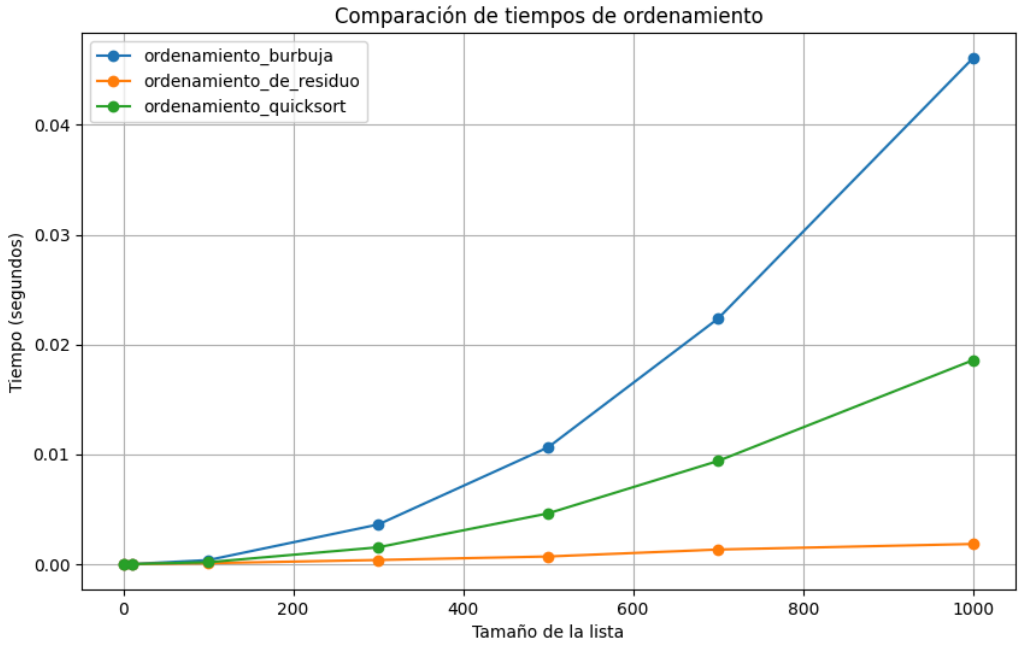
Proyecto 1.

A medida que escribíamos los códigos de los algoritmos y teniendo en cuenta la teoría viste en la materia podíamos intuir los órdenes de magnitud de cada uno de ellos: Cuando desarrollamos el ordenamiento burbuja intuíamos que su orden de magnitud sería n^2 dado a los dos bucles for anidados que posee; cuando analizamos el ordenamiento de residuos pudimos observar que sucedía un caso similar, pero que a diferencia del primero mencionado posee un bucle while y dentro de él un bucle for, haciéndonos debatir entre sí era n o n^2. Similar sucedía en el código de ordenamiento "quicksort" dado a que poseía dos bucles for anidados.

A la hora de realizar los gráficos pudimos ver que no estábamos muy equivocadas cuando discutimos el orden de magnitud de los primeros dos, pero que a diferencia de lo que pensábamos, el tercer algoritmo de ordenamiento poseía un orden de magnitud muy distinto:



Basándonos en el gráfico, el orden de complejidad de cada algoritmo de ordenamiento son los siguientes:

- ➔ Ordenamiento burbuja: O (n^2)
- ➔ Ordenamiento de residuo: O (k*n)
- ➔ Ordenamiento quicksort: O(n* log n)

En la curva azul, que corresponde al ordenamiento burbuja, confirmamos nuestra predicción de $O(n^2)$ ya que a medida que aumenta n (tamaño de la lista) aumenta el tiempo de forma cuadrática.

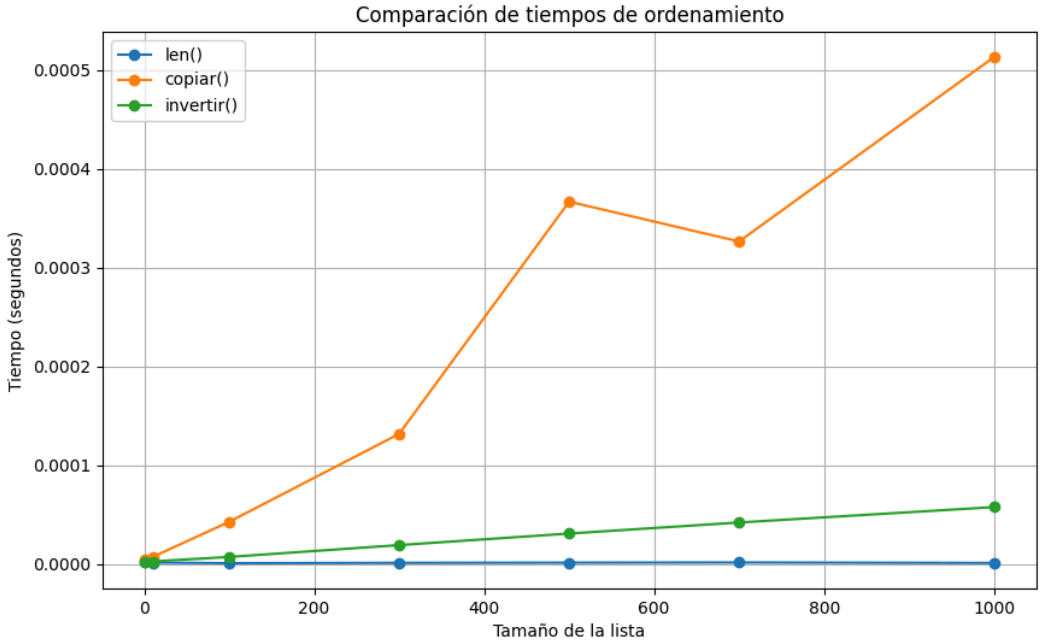
En la curva verde, en cambio, el tiempo crece logarítmicamente a medida que aumenta el tamaño de entrada.

En lo que respecta a la curva naranja, la que detalla lo que sucede con el ordenamiento residuo, es difícil decidir si se trata de un orden de magnitud n o n^2, sin embargo podríamos definir que crece el tiempo de forma lineal a medida que aumenta el tamaño de entrada.

Según https://docs.python.org/3/library/functions.html# sorted la función sorted(item) devuelve una nueva lista ordenada a partir de los elementos en el iterable. Tiene dos argumentos opcionales: key y reverse. Key nos permite aplicar una función antes de comparar los elementos; y reverse recibe un booleano, reverse = False la función ordena de menor a mayor, y si reverse = True ordena de mayor a menor.
Su orden de complejidad se puede deducir como O(n*log n), como todos los métodos de ordenamiento de la biblioteca de Python, a partir de su método de funcionamiento. Comparándolo con los que analizamos anteriormente, sorted() es uno de los más rápidos, junto con la función quicksort.

Proyecto 2.

En el este proyecto, creamos una lista doblemente enlazada y sus respectivos métodos y atributos, entre ellos, los que se observarán en el gráfico. Asimismo debimos comparar los métodos __len__(),copiar() e invertir(). Si analizamos el código consideramos que el primer método mencionado posee orden de magnitud uno O(1) dado a que devuelve el tamaño de la lista; con respecto a “copiar” podemos observar, en su código, que posee un bucle while, siendo su orden de magnitud una función lineal, es decir, es O(n). El último método a analizar es “invertir”, en lo que respecta a su código, intuimos que es n.



Cuando analizamos el gráfico, observamos que __len__() es la más eficiente de las tres, convirtiéndose prácticamente en O(1) como habíamos predicho; la curva verde, que es invertir() queda en segundo lugar al alejarse de O(1) y tendiendo a ser O(n), por crecer de forma lineal. Por último, copiar() es O(n) como se puede observar claramente, aunque si realizáramos otro análisis con una mayor cantidad de puntos veríamos con mayor claridad la linealidad.

Proyecto 3.
En el último proyecto, el del juego guerra, los resultados fueron los esperados: la funcionalidad de los tests.