# Trabajo Práctico N°1
# Algoritmos y Estructuras de Datos

**Villaverde, Federico**
**Pauligk, Maximo**

# Objetivos

- Aplicar los conceptos que resulten necesarios para la implementación de algoritmos de ordenamiento.
- Implementar tipos abstractos de datos (TAD).
- Analizar y determinar la complejidad de los algoritmos implementados.
- Utilizar manejo de excepciones para validación de datos y operaciones en los TADs.
- Uso de pruebas unitarias para corroborar el correcto funcionamiento de los códigos implementados.

# Problema 1

Consignas:
Implementar en python los siguientes algoritmos de ordenamiento:

a) Ordenamiento burbuja
b) Ordenamiento quicksort
c) Ordenamiento por residuos (radix sort)

Corroborar que funcionen correctamente con listas de números aleatorios de cinco dígitos generados aleatoriamente (mínimamente de 500 números en adelante).

Medir los tiempos de ejecución de tales métodos con listas de tamaño entre 1 y 1000. Graficar en una misma figura los tiempos obtenidos. ¿Cuál es el orden de complejidad O de cada algoritmo? ¿Cómo lo justifica con un análisis a priori?

Comparar ahora con la función built-in de python **sorted**. ¿Cómo funciona sorted? Investigar y explicar brevemente.

Resolución:

De acuerdo a la consigna, implementamos en python, en el archivo “Trabajo Practico_1/proyecto_1/modules/Ordenamientos.py” de nuestro repositorio, los metodos de ordenamiento solicitados.

**Ordenamiento burbuja:** Es un método de ordenamiento simple que recorre una lista de n elementos unas (n_1) y comparando elementos adyacentes entre sí desplazando al mayor de éstos a la derecha. A medida de que realiza pasadas, los elementos mayores se acumulan en el final y disminuye el nro de operaciones.

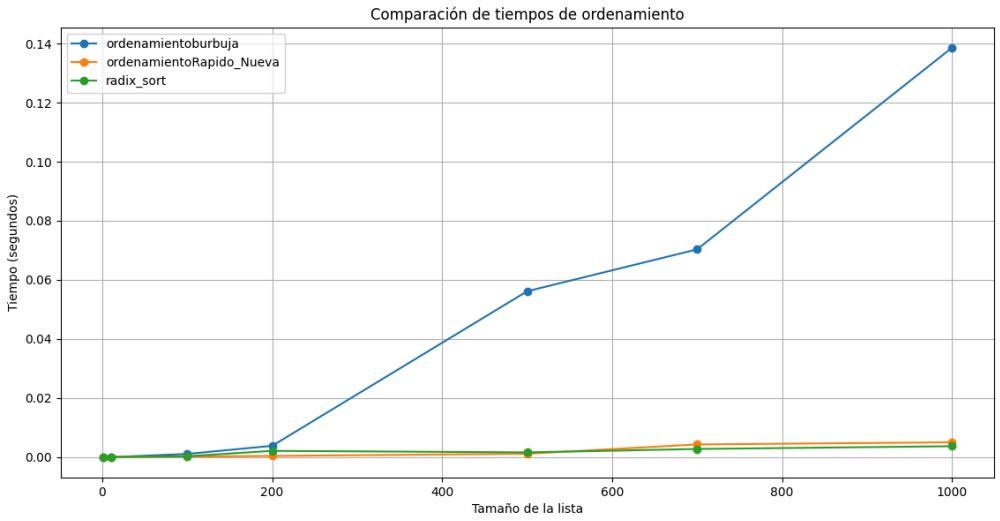
f(n) = n^2 => O(n^2)

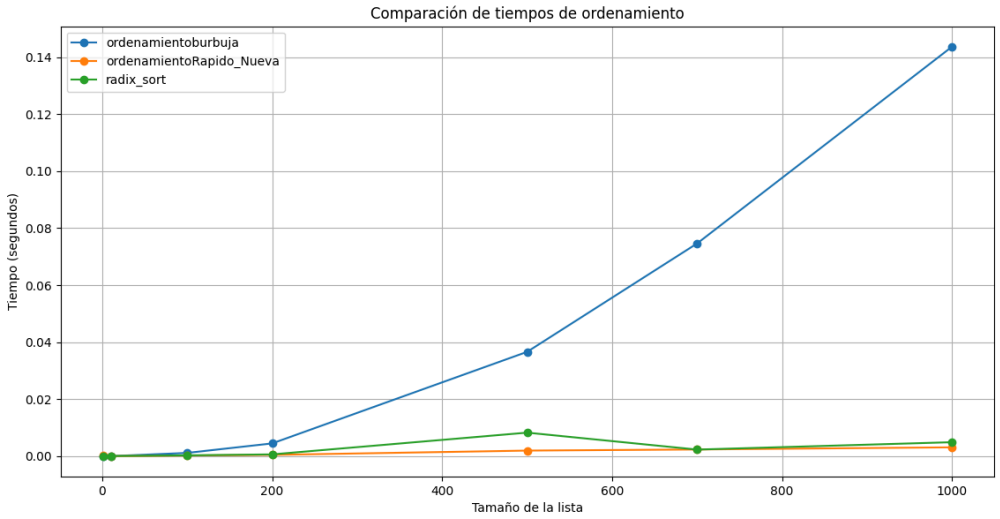
Como se trata de dos ciclos anidados, tiene una complejidad de O(n^2). Si bien es un algoritmo sencillo de entender y de implementar, es poco eficiente en cuanto a tiempo cuando se trata de listas con un gran numero de elementos. Métodos como quicksort y radix

sort ofrecen un rendimiento significativamente mejor, optimizando el tiempo de ejecución y reduciendo la cantidad de operaciones necesarias para ordenar grandes volúmenes de datos.

**Ordenamiento Rápido:** Es un método de ordenamiento recursivo, que se basa en elegir un pivote (en nuestro caso el primer elemento de la lista) y ordenar la lista de forma tal que los valores menores a éste queden a su izquierda y los mayores a éste a su derecha. A éstas dos sublistas (la de la izquierda del pivote y a la derecha de este) vuelven a llamar a la función recursiva, y asi sucesivamente hasta ordenar todos los items de la lista.
El rendimiento promedio de ésta, si está bien elegido el pivote y si la lista no está previamente ordenada es de O(n log n), mientras que en el peor de los casos es O(n^2). Una de las ventajas de este algoritmo es que aporta la posibilidad de mejorar su desempeño en escenarios desfavorables. Eligiendo eficientemente el pivote se puede reducir considerablemente la probabilidad de obtener el peor caso, mejorando así su desempeño en diversas situaciones.

**Ordenamiento por Residuos (radix sort):** Es un método de ordenamiento no comparativo que utiliza el método counting_sort(). Su procedimiento, a grandes rasgos, se basa en ordenar los elementos según las unidades, luego las decenas, luego las centenas, así sucesivamente hasta completar los dígitos del mayor de los números de la lista . Cuenta con una complejidad de O(nk); donde n es la cantidad de elementos de la lista y k la cantidad de dígitos del mayor de sus elementos. Es muy rápido cuando los valores son enteros que tienen pocos dígitos y valores acotados.






Burbuja => O(n^2)
QuickSort => O(n log n)
Radix Sort => O(nk), donde k es el número de dígitos del mayor elemento

Como vemos en la gráfica, cuando se tratan de listas de pocos elementos (menos de 200) los tres métodos tienen un desempeño similar. Sin embargo, al aumentar la cantidad de elementos, la cantidad de operaciones realizadas por el método de burbuja tiene una crecimiento cuadrático mientras que el resto tienen un crecimiento más lento, casi uniforme.

Internamente, sorted() utiliza un algoritmo llamado Timsort, que combina estrategias de Merge Sort (ordenamiento por mezcla) y Insertion Sort (ordenamiento por inserción). Esta combinación le permite detectar secuencias ya ordenadas dentro de la lista para optimizar el rendimiento general del algoritmo. Gracias a esto, Timsort es particularmente eficiente en casos donde la lista tiene patrones de orden preexistentes.

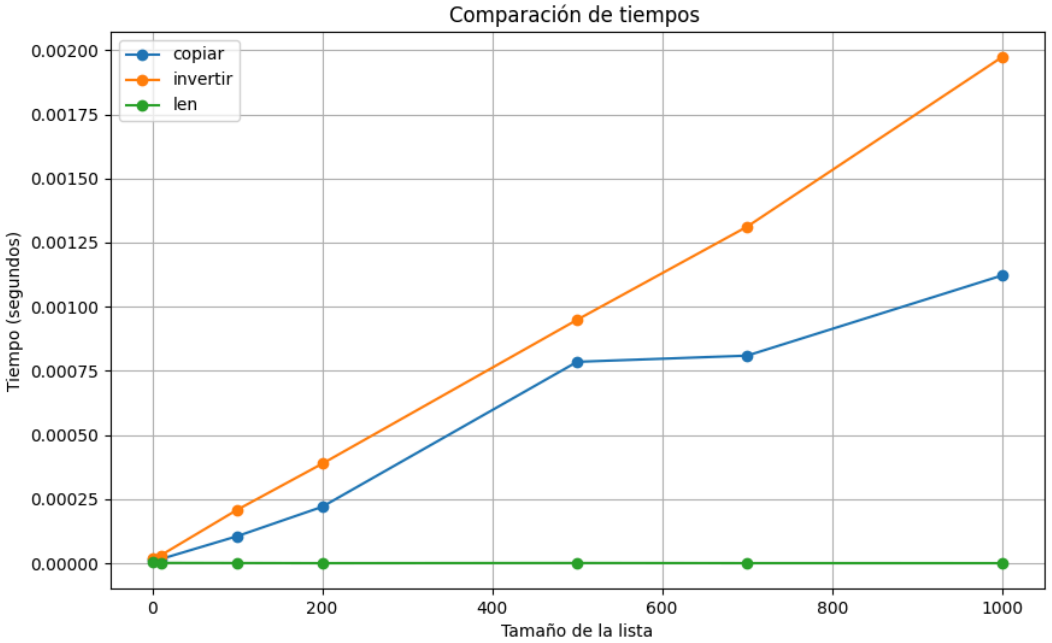
La complejidad computacional de sorted() es:

Mejor caso: O(n), cuando la lista ya está casi ordenada.

Promedio y peor caso: O(n log n), manteniéndose eficiente aún en listas desordenadas.

## PROBLEMA 2:

En el problema 2 se realizó la implementación de un código para el tipo abstracto de dato Lista Doblemente Enlazada. En ésta actividad se solicitó, al igual que la anterior, realizar un gráfico del desempeño temporal de los métodos(copiar, invertir, len) en función al tamaño de las listas con las que trabajaban.
A continuación se presenta el gráfico correspondiente:




Evidentemente el método “len()” tiene un orden de complejidad de O(1). Como el tamaño de la lista se actualiza con cada operación sobre ésta, cuando se llama a la función len(), devolver el tamaño sólo cuesta una operación, independientemente del tamaño de la lista.
Por otro lado, si realizamos un análisis a priori del código de los métodos “invertir()” y “copiar()” observaremos que se trata de funciones con un orden de complejidad O(n), ya que no hacen más que recorrer la lista una sóla vez. Si bien forman parte del mismo grupo de funciones O(n), evidentemente el método copiar tiene un desempeño ligeramente mejor a el método invertir(). Esto se  lo podemos atribuir a que el método “invertir()” además de recorrer la lista una vez para agregar los elementos de forma invertida al principio de ésta, la vuelve a recorrer para eliminar lo otros y que no se repitan, lo cual puede justificar ésta diferencia en su actividad.