

## Trabajo Práctico N° 1

### Problema 1: Lista doblemente enlazada

Para este problema se implementó la clase Nodo. Ésta representa un elemento en una lista doblemente enlazada.

Los atributos son:

- dato
- anterior
- siguiente

En la clase ListaDobleEnlazada los atributos son:

- cabeza
- cola
- tamano

La clase Nodo la implementamos una vez que creamos la clase ListaDobleEnlazada, en la cuál a medida que creabamos los métodos los íbamos probando en el test brindado por la cátedra. Los métodos creados son:

- \_\_len\_\_
- vacía
- agregar
- \_\_iter\_\_
- copiar
- agregar\_al\_inicio
- agregar\_al\_final
- insertar
- extraer
- ordenar
- \_quick\_sort
- \_particionar
- \_\_add\_\_
- concatenar
- invertir

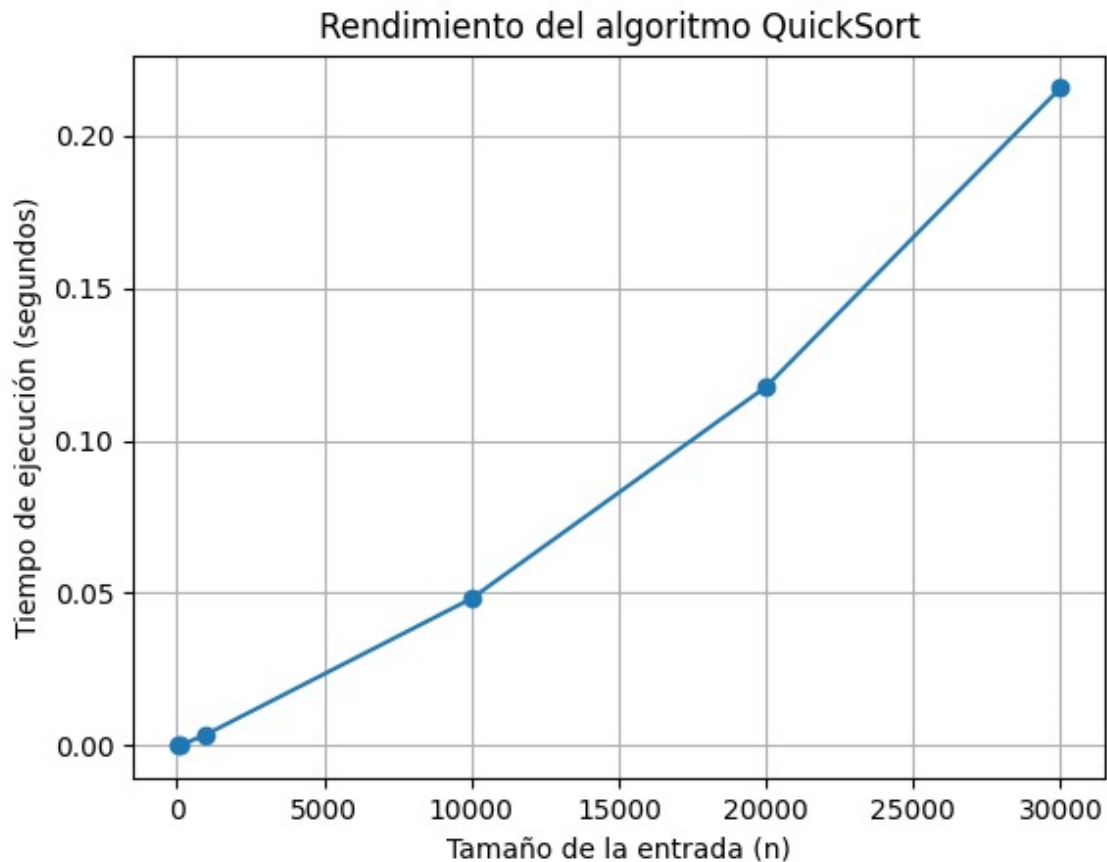
Ran 15 tests in 0.024s

OK

El algoritmo de ordenamiento que elegimos fue el QuickSort, en cada llamada a “\_quick\_sort” se selecciona un pivote, se dividirá el rango en 2 subrangos, uno de ellos tendrá elementos menores y otro elementos mayores al pivote. Se realiza una llamada recursiva en cada subrango, en cada llamada el rango original se reduce a la mitad, esto pasará hasta que solo quede un elemento en el rango o esté vacío.

Si el pivote divide el rango en dos partes iguales (o aproximadamente iguales) en cada llamada recursiva el orden de complejidad de nuestro método será de  $O(n \log n)$ ; si el pivote divide el rango en partes desiguales se obtendrá un orden de complejidad de  $O(n^2)$ .

Gráfica del orden de complejidad en función del número de elementos:



Primero definimos el método llamado 'generar\_lista\_aleatoria' que toma como argumento 'n', dentro de este método creamos la lista con valores aleatorios entre 1 y 1000. Creamos el método 'medir\_tiempo\_ordenamiento' el cual también toma como argumento 'n', almacenamos los valores aleatorios dentro de la variable 'lista' y registramos el tiempo actual en la variable 'inicio'. Asignamos los valores de 'ListaDobleEnlazada' a 'lista\_doble\_ordenada', ordenamos la lista y registramos el tiempo actual en la variable 'fin'. La función 'medir\_tiempo\_ordenamiento' nos devuelve la diferencia entre el tiempo final y el tiempo inicial, es decir, el tiempo de ejecución del algoritmo de ordenamiento. Creamos una lista 'tamanos' y una lista vacía 'tiempos', para cada tamaño se mide el tiempo de ejecución del algoritmo de ordenamiento. Finalmente creamos el gráfico que muestra cómo varía el tiempo de ejecución en función del tamaño de entrada, mostrando cómo cambia el rendimiento del algoritmo de ordenamiento (QuickSort) a medida que aumenta el tamaño de los datos a ordenar.

## Problema 2: Juego de Guerra

Para este problema se implementaron tres clases principales, además de la clase `ListaDobleEnlazada` que debía ser utilizada para poder desarrollar el código :

**Carta:** representa una carta individual en el juego. Cada una de ellas tiene como atributos:

- valor
- palo
- boca\_arriba
- boca\_abajo

**Mazo:** representa el mazo de cartas. Sus atributos son:

- tamaño
- mazo: instancia de la clase `ListaDobleEnlazada`

El atributo mazo es una instancia de la clase `ListaDobleEnlazada`.

Esta clase tiene varios métodos para manipular las cartas del mazo:

- poner\_abajo
- poner\_arriba
- sacar\_arriba
- esta\_vacio

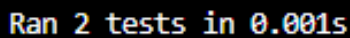
**JuegoGuerra:** implementa el juego de cartas “Guerra”.

Sus atributos son:

- jugador1 y jugador2, donde cada uno tiene un mazo de cartas, repartidas.
- turno
- max\_turnos: indica el número máximo de turnos permitidos para evitar bucles infinitos.

Los métodos son:

- iniciar\_juego se implementa la lógica del juego y determina el ganador.
- comparar\_cartas



## Problema 3: Ordenamiento de datos aleatorio

A partir del código propuesto por la cátedra, donde se creaba el archivo de datos, se crearon los métodos:

- `mezcla_directa(nombre)`: donde se lee el archivo de datos creado anteriormente y lo divide en bloques de tamaño **B**. Cada bloque se ordena uno a uno y luego se escribe nuevamente en el archivo.
- `verificar_ordenamiento(nombre)`: verificará si los datos están ordenados en el archivo. Primero lee los datos, los compara con la versión ordenada (hecho con la



función anterior) y devuelve **True** si los datos están ordenados, **False** en el caso contrario.