

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:

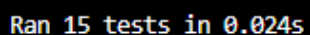
- `dato`
- `anterior`
- `siguiente`

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:

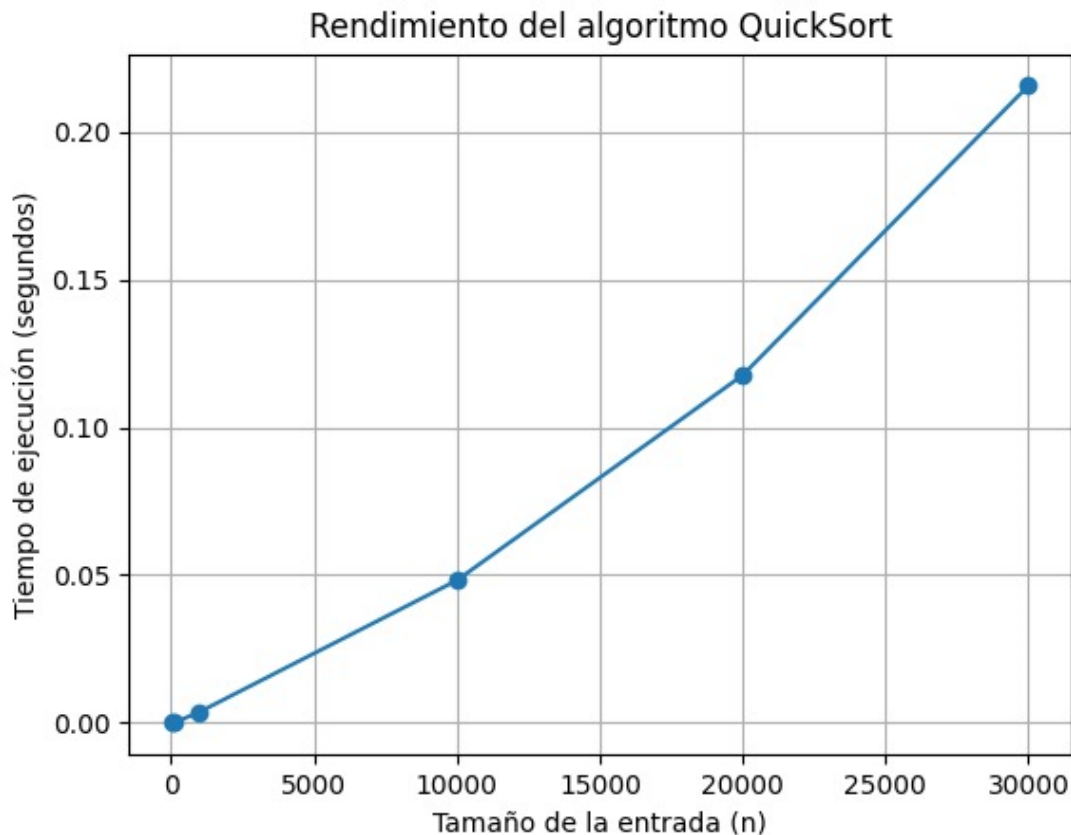
- `__len__`
- `vacía`
- `agregar`
- `__iter__`
- `copiar`
- `agregar_al_inicio`
- `agregar_al_final`
- `insertar`
- `extraer`
- `ordenar`
- `_quick_sort`
- `_particionar`
- `__add__`
- `concatenar`
- `invertir`



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.

La gráfica muestra que el número de comparaciones realizadas por Quicksort aumenta proporcionalmente a $n \log n$, lo que se corresponde con la complejidad temporal promedio del algoritmo.

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

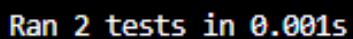
JuegoGuerra: implementa el juego de cartas “Guerra”.

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:

- ordenamiento: este método realiza el ordenamiento externo. Primero, lee el archivo de entrada en bloques de tamaño `bloque_size`, ordena cada bloque y los almacena en memoria. Luego, utiliza un heap para fusionar los bloques ordenados y escribe el resultado en el archivo de salida.
- comprobar_tamano_archivos: este método comprueba si dos archivos tienen el mismo tamaño. Esto se hace utilizando la función `getsize` del módulo `os`.



- `comprobar_ordenamiento_archivo`: este método verifica si los números en un archivo están ordenados en orden ascendente.