
# UNIVERSIDAD NACIONAL DE ENTRE RÍOS

# FACULTAD DE INGENIERÍA

**CARRERA: Tecnicatura Universitaria en Procesamiento y Explotación de Datos**

**MATERIA: Algoritmos y Estructuras de Datos**

**ACTIVIDAD: Trabajo Práctico N°1**

**Profesores:** Javier Eduardo Diaz Zamboni

Jordán F. Infrán

Diana Vertiz del Valle

Belén Ferster

**Alumnos:** Ruiz Diaz, Enzo

Venturini, Angelo

**Fecha de Entrega: 30/09/2022**

# Trabajo Práctico N°1

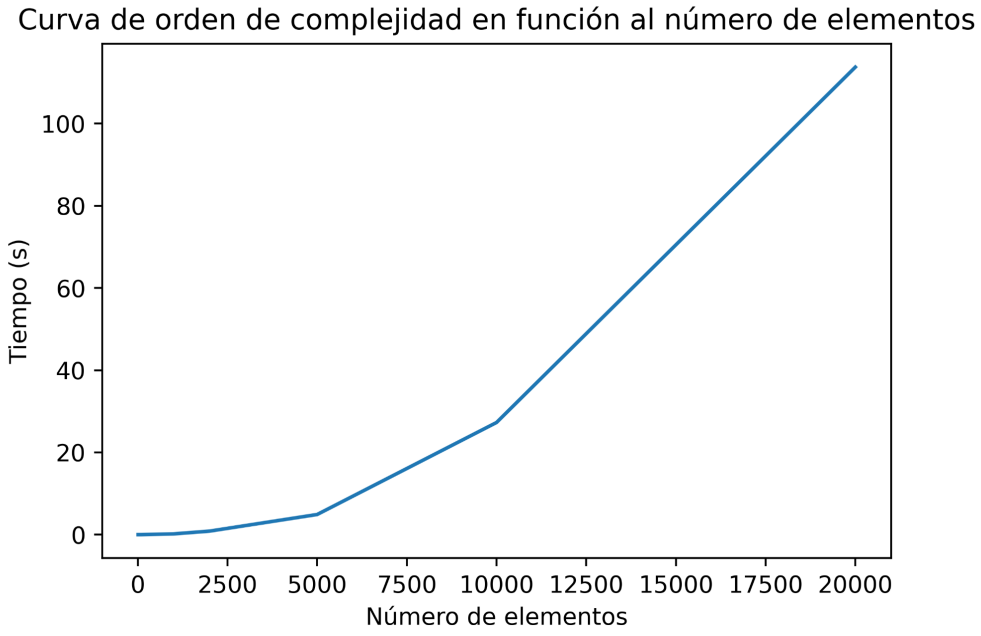
## Objetivos

- Implementar tipos abstractos de datos (TAD).
- Aplicar los conceptos vistos en clase para la implementación de un algoritmo de ordenamiento para el TAD utilizado.
- Analizar la complejidad de los algoritmos implementados.
- Utilizar manejo de excepciones para validación de datos.
- Realizar pruebas unitarias para los códigos implementados.

## Solución y Resultados

### Ejercicio 1

Implementamos un algoritmo de ordenamiento por inserción. Este algoritmo, en el peor de los casos, toma un tiempo de O($n^2$). Por otro lado, cuando ocurre el mejor de los casos, que se da cuando el arreglo se encuentra ordenado, toma el tiempo de O(n). En promedio el algoritmo de inserción requiere O($n^2$) operaciones para ordenar una lista de **n** elementos. A continuación se muestra una gráfica del orden de complejidad en función del número de elementos. La misma se realizó con 10, 100, 1000, 2000, 5000, 10000 y 20000 elementos en la lista:



# Ejercicio 2

Se utilizaron las siguientes clases para la implementación de este ejercicio:

- JuegoGuerra: Es la clase del juego como tal. Inicialmente arma el mazo, reparte las cartas a los jugadores. Luego, mediante el método iniciar_juego, da comienzo al loop de juego donde se simulan todas las rondas de los jugadores, definiendo el ganador de cada turno, y luego chequeando si hay algún ganador del juego en sí. Se muestra por consola el resultado de cada turno, junto con la guerra de cada jugador. Para esto se sobrecargó el método de str de la clase para mostrarse a sí misma.
- Carta: Es la clase para cada carta del mazo, recibe un palo y un valor, e inicialmente pone la carta boca abajo. Esta clase permite comparar sus valores con los de otras cartas, como también una representación en string de la carta dependiendo de si está boca arriba o boca abajo.
- ColaDoble: Se utiliza esta clase para los mazos, tanto como el inicial como el de los jugadores. Se optó por esta clase ya que al momento de repartir, los mazos se deben comportar como una pila, y, durante el juego, se deben comportar como una cola. La implementación de esta clase se realizó en un módulo aparte, utilizando la lista doblemente enlazada previamente implementada en el ejercicio 1 como estructura interna de la clase.

El loop de juego explicado en pseudocódigo (desde que se llama a iniciar_juego):

- Se genera la mesa de cada jugador (dos listas de Python, esta decisión fue para simplificar la inserción de cartas en cada mesa y la entrega del botín de guerra para cada jugador, donde cabe destacar que ambas mesas podrían haber sido una pila).
- Arrancamos el juego sin que haya guerra.
- Mientras que la cantidad de turnos jugados sea menor a 10000
    - Verificamos las cartas restantes de cada jugador. Si alguno no tiene cartas, debemos cortar el loop de juego y asignar el ganador.
    - Si no estamos en guerra, debemos aumentar la cantidad de turnos.
    - Si estamos en guerra, debemos verificar que la cantidad de cartas de cada jugador sea la suficiente para la jugada de la guerra, donde la cantidad mínima necesaria son 4 cartas. Si esto no sucede, asignamos el ganador y salimos del loop de juego, en caso contrario, jugamos las 3 cartas boca abajo.
    - Independientemente del estado, jugamos las cartas boca arriba para cada jugador.
    - Mostramos la mesa al usuario.
    - Decidimos el ganador del turno. En caso de empate vamos a la guerra y dejamos que el loop continúe, caso contrario le otorgamos el botín de guerra al ganador y limpiamos las mesas.
- Fin del loop de juego.
- Si la cantidad de turnos jugados es exactamente igual a 10000 y no hay un ganador, el resultado del juego es empate.
- Mostramos el resultado al usuario.

# Ejercicio 3

Para implementar el algoritmo de ordenamiento externo por mezcla natural o equilibrada decidimos dividirlo en tres funciones:

- Dividir la cual se encarga de realizar las particiones.
- Mezcla realiza la mezcla ordenada.
- ordenar por mezcla natural que funciona a modo de controladora para iterar las funciones anteriores las veces necesarias.

Para el algoritmo que verifica que el archivo está ordenado decidimos implementar una función que recorra el archivo leyendo de a dos líneas y comparándolas, donde retorna False si el valor de la línea anteriormente leída es mayor a la siguiente, y True si el loop termina (es decir, todas las líneas estaban en orden ascendente).
Así mismo nos pedían comparar el tamaño del archivo en bytes antes y después de realizado el ordenamiento, por lo que utilizamos el módulo os y su función path.getsize para obtener el tamaño del archivo en ambos estados (desordenado y ordenado). Se realiza la comprobación de los tamaños antes de ser ordenado y después, donde se muestra en consola si ha variado o no el tamaño del archivo.
Se realizaron pruebas con diferentes cantidades de elementos, corriendo por último la pedida por la cátedra, de 5 millones de elementos con 20 cifras. Al momento de la prueba, el equipo contaba con un microprocesador Ryzen 7 1700, 16gb de ram DDR4 y un disco SSD Kingston A400. El output del programa fue el siguiente:

```
Cantidad de valores a escribir: 5000000
t = 1000000 , N_restantes = 4000000
t = 1000000 , N_restantes = 3000000
t = 1000000 , N_restantes = 2000000
t = 1000000 , N_restantes = 1000000
t = 1000000 , N_restantes = 0
File Size: 110000000 bytes
Tiempo empleado: 302s
Esta ordenado
File Size: 110000000 bytes
El archivo no varió su tamaño.
```