Universidad Nacional de Entre Ríos

Facultad de Ingeniería

Algoritmos y Estructuras de Datos

Informe General del Trabajo Práctico N°1

Aplicaciones de los tipos abstractos de datos

Rodriguez Esteban

Trevisán Sergio

Romero Sheizza

2025

Problema 1

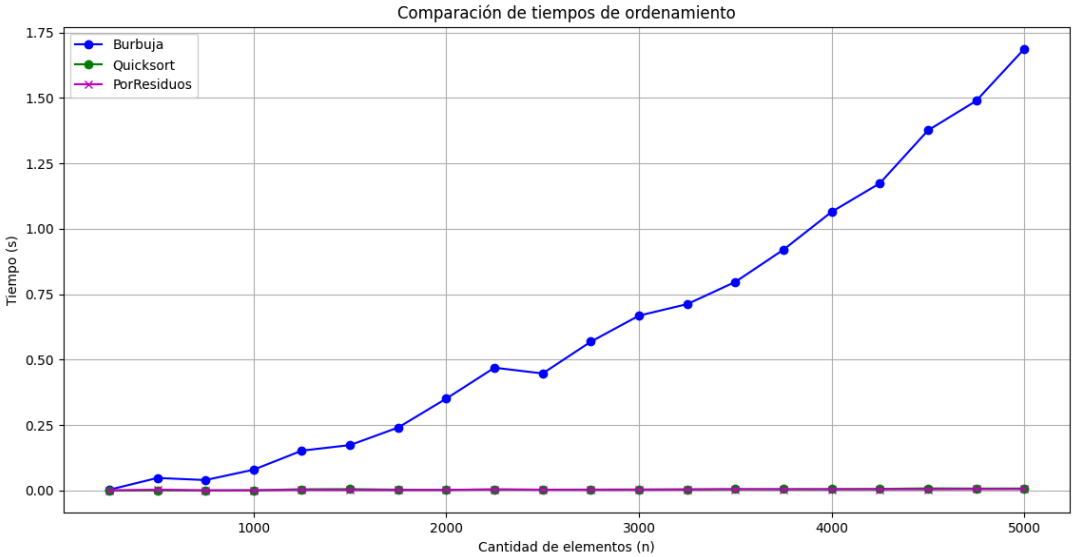
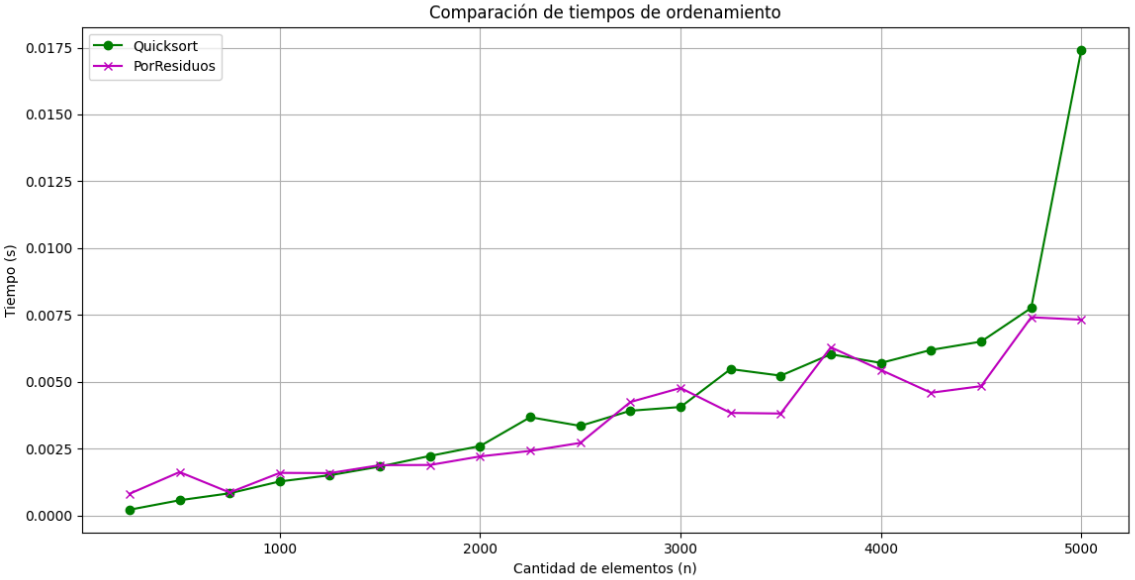


Gráfico 1: Comparación general de tiempos de ordenamiento.

Descripción:
En esta gráfica se muestra la comparación de tiempos de ejecución entre los algoritmos Burbuja, Quicksort y PorResiduos. Debido al crecimiento cuadrático de Burbuja, sus tiempos son notablemente mayores, lo que resalta aún más la eficiencia de Quicksort y PorResiduos.



Gráfica 2: Comparación de tiempos de ordenamiento entre Quicksort y PorResiduos para mejor visualización de las diferencias.

Facultad de Ingeniería
UNER

Descripción:
Esta gráfica compara el rendimiento de los algoritmos Quicksort y PorResiduos (RadixSort) para listas cuyo tamaño varía desde 250 a 5000 elementos.
Se realizó una visualización separada para observar mejor las diferencias sutiles de comportamiento entre estos métodos de ordenamiento eficientes.
Hasta n≈4000 approx, RadixSort y Quicksort son muy competitivos. RadixSort es más rápido (casi lineal), mientras que Quicksort puede sufrir "picos" en su tiempo de ejecución.
Cabe destacar que, cuando se trabaja con listas de hasta 5000 elementos compuestas por números de solo dos o tres cifras, la complejidad O(n·k) de RadixSort (con k constante) lo vuelve asintóticamente más eficiente que Quicksort, cuya complejidad O(n log n) crece más rápidamente.

Ordenamiento Burbuja

Complejidad: O(n^2)

- Compara pares de elementos adyacentes y los intercambia si están en orden incorrecto.
- En el peor caso, requiere recorrer el arreglo n veces y en cada pasada compara hasta n elementos.
- En el gráfico se observa que el tiempo de ejecución de Burbuja crece de manera no lineal y rápidamente a medida que n aumenta, lo cual concuerda con una complejidad cuadrática.

Quicksort

Complejidad promedio: O(n log (n))

Peor caso: si la lista ya está ordenada o semi-ordenada, el algoritmo puede presentar una complejidad O(n^2), debido al desbalance que quedaría en las listas.

- Quicksort utiliza el enfoque de “divide y vencerás”.
- Divide el arreglo en dos subarreglos y luego ordena recursivamente.
- En el gráfico, la línea de Quicksort es prácticamente plana, lo que indica que su tiempo de ejecución es muy bajo en comparación con la de burbuja, y crece mucho más lentamente, como se espera de un algoritmo eficiente.

Por residuos

Complejidad estimada:O(n·k), donde *k* es el número de dígitos analizados.

- Aunque no es un algoritmo de ordenamiento típico, puede estar clasificando elementos con base en una operación rápida (como el módulo).
- En el gráfico, su tiempo también se mantiene constante y muy bajo, incluso más bajo que Quicksort, lo cual sugiere que se trata de un algoritmo lineal o casi constante en este rango de valores.

La función sorted() de Python se utiliza para ordenar elementos de cualquier iterable (como listas, tuplas, diccionarios, etc.) y devuelve una nueva lista ordenada, sin modificar el iterable original. Esta divide una lista en sublistas de tamaño óptimo y las ordena con un algoritmo de inserción, para luego fusionarlas, lo que logra un orden de complejidad de O(n log n) donde n es el número de elementos en el iterable que se está ordenando.

```python
def ordenamientoPorResiduos (unaLista):
    posiciones = [4,3,2,1,0]
    for p in posiciones:
        auxiliar = {i: [ ] for i in range (10)}
        for x in unaLista:
            cadena = str (x)
            digito = int (cadena [p])
            auxiliar [digito].append(x)
        unaLista = []
        for i in range(10):
            unaLista.extend (auxiliar[i])
```

Explicación del código para el Ordenamiento por residuos (radix sort).

Ordena por el último dígito (el menos significativo). Creamos un diccionario de listas vacío, uno para cada dígito del 0 al 9.

Para cada número x en la lista:

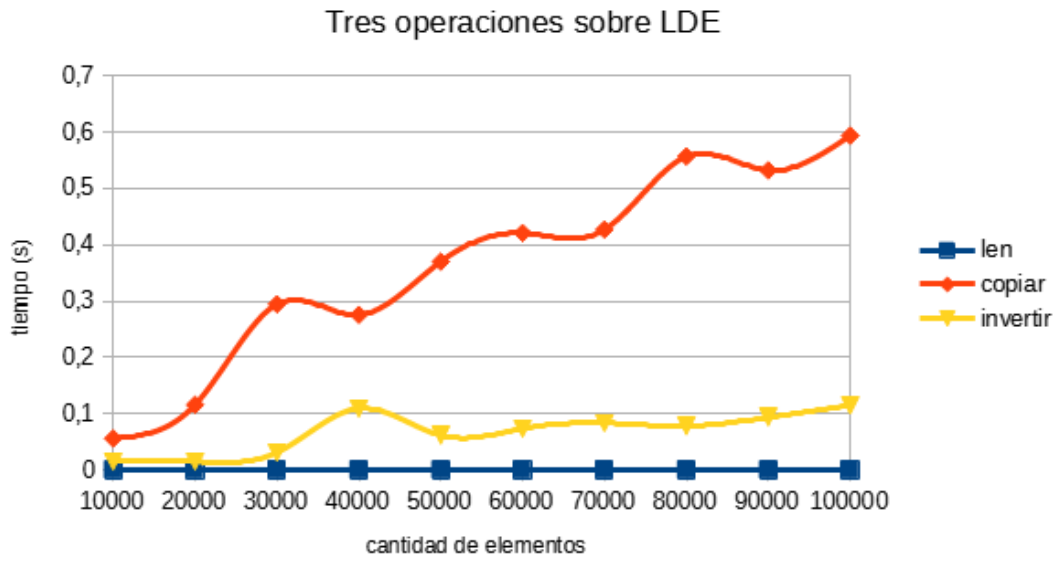
- Lo convierte a cadena de texto (str(x)), para poder acceder a sus dígitos.
- Toma el dígito en la posición p.
- Usa ese dígito para meter el número en la correspondiente lista en auxiliar.

Después reconstruimos la lista:

- Empezando vacía.
- Para cada dígito de 0 a 9, saca los números (en orden) y los agrega a unaLista.

Facultad de UNER Ingeniería

Problema 2



Se puede observar que:
Len tiene un orden de complejidad O(1), es una línea horizontal en la gráfica, lo cual es lógico ya que solo debe revisarse el atributo self.tamanio de la lista doblemente enlazada.
Copiar e invertir, ambas son de complejidad lineal.
Invertir crece con la cantidad de elementos, pero menos empinadamente que copiar, esto es porque no crea nuevos nodos, simplemente cambia los punteros anterior y siguiente de cada nodo.
Copiar crece casi linealmente con la cantidad de elementos, aunque con algunas pequeñas oscilaciones. Esto se debe a que implica recorrer todos los nodos y crear uno nuevo por cada uno, lo que conlleva a más operaciones de memoria, lo que justifica que sea más costoso que invertir.

Las listas doblemente enlazadas se basan en el uso de nodos, definidos previamente mediante una clase personalizada. En esta implementación, cada nodo tiene una carga útil (el valor a almacenar) y dos referencias (punteros): una al nodo anterior y otra al nodo siguiente. Ambos punteros pueden tener el valor None, lo que indica que no existe nodo anterior o siguiente.
Una lista doblemente enlazada se inicializa vacía. Se le pueden agregar elementos mediante diferentes métodos que crean objetos Nodo y los enlazan correctamente. La clase ListaDobleEnlazada es la encargada de gestionar los nodos y contiene atributos clave para su funcionamiento:
__cabeza: referencia al primer nodo de la lista. Su atributo anterior siempre es None.

__cola: referencia al último nodo. Su atributo siguiente siempre es None.

__cantidad_elementos: contador interno que lleva el número total de nodos en la lista. Se inicializa en cero.

Las funciones implementadas fueron:

- **__str__(self)**: genera una representación en cadena de la lista, mostrando todos los elementos separados por " - ". Si está vacía, devuelve "lista sin elementos".

- **esta_vacia(self)**: devuelve True si la lista no contiene elementos (__cantidad_elementos == 0), de lo contrario False.

- **__len__(self)**: permite usar len(lista) para obtener la cantidad de nodos actuales.

- **agregar_al_inicio(self, item)**: inserta un nuevo nodo al principio de la lista. Si la lista está vacía, el nuevo nodo se convierte tanto en la cabeza como en la cola.

- **agregar_al_final(self, item)**: agrega un nodo al final de la lista. También maneja el caso en que la lista esté vacía.

- **insertar(self, item, posicion=-1)**: permite insertar un nodo en cualquier posición válida. Si posicion == 0, se usa agregar_al_inicio(). Si la posición es negativa o igual al largo de la lista, se inserta al final. En posiciones intermedias, se ajustan los punteros de los nodos adyacentes.

- **extraer(self, posicion=-1)**: elimina un nodo en una posición determinada y devuelve su valor. Si no se especifica posición, se elimina el último nodo (__cola). Si se elimina la cabeza o la cola, los atributos __cabeza y __cola se actualizan adecuadamente. También valida errores de tipo o posición inválida.

- **copiar(self)**: crea una nueva lista con los mismos elementos que la original, conservando el orden. Cada elemento se inserta usando insertar.

- **invertir(self)**: invierte el orden de los nodos de la lista. Para ello, intercambia los punteros anterior y siguiente de cada nodo. Finalmente, se actualizan los atributos __cabeza y __cola.

- **concatenar(self, p_lista)**: agrega todos los elementos de otra lista doblemente enlazada al final de la lista actual, sin modificar la original. Se recorre nodo por nodo y se insertan los elementos con agregar_al_final.

- **__add__(self, p_lista)**: redefine el operador + para sumar dos listas. Crea una nueva lista vacía, concatena la lista actual (self) y luego la lista recibida (p_lista). Usa internamente el método concatenar.

Problema 3

Para la resolución de este problema se consideró al código brindado por la cátedra y posteriormente implementamos a la clase mazo, la cual representa una baraja de objetos Carta mediante una lista doblemente enlazada -que se implementó en el problema anterior. La estructura de lista doblemente enlazada permite una manipulación eficiente de las

cartas, facilitando operaciones como agregar, quitar y recorrer las cartas tanto desde el principio como desde el final del mazo.
Como resultado, el código informará si el ganador fue el jugador 1, jugador 2 o bien si hubo algún empate. La lógica para determinar el ganador se basa en la comparación de las cartas jugadas por cada jugador en cada ronda.