Facultad de Ingeniería
UNER

UNER

Facultad de Ingeniería

Algoritmo y Estructura de Datos

TP N°1

Aplicación de TADs

Fecha de Presentación: 02/05/25

Integrantes: Frederich,Rocio
Merlo, María Fernanda
Sivila, Matias

IMPLEMENTACIÓN DE ALGORITMOS DE ORDENAMIENTOS.

➢ **ORDENAMIENTO BURBUJA.**

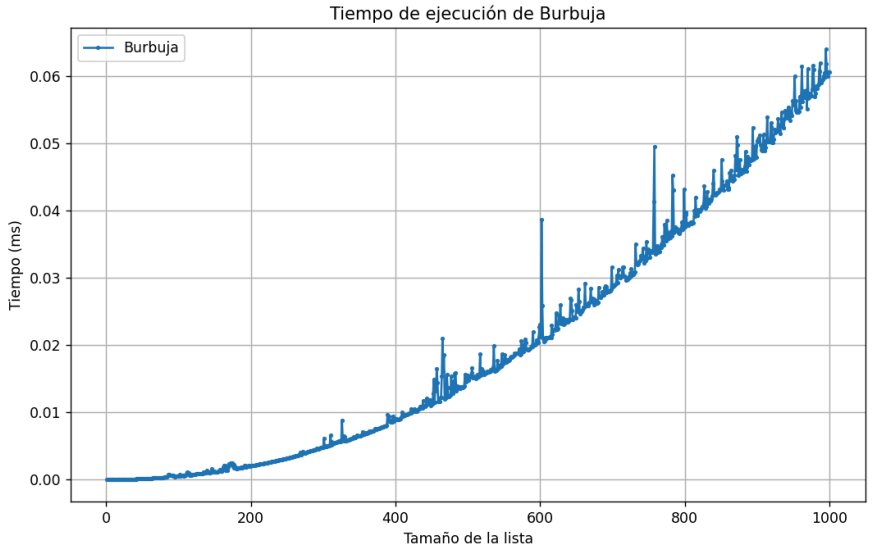
Este ordenamiento consiste en recorrer la lista desde el elemento **j = 0** y compararlo con el siguiente **(j + 1)**. Si este último es **menor** que el primero, se intercambian. Luego se incrementa el índice **j** y se repite el proceso. Si no se cumple la condición, simplemente se avanza el índice. Dicho ordenamiento tiene una complejidad promedio de **O(n^2)**, mientras que en el mejor de los casos (cuando la lista ya está ordenada) tiene una complejidad de **O(n)**.

```python
#METODO BURBUJA

def metodo_burbuja(lista):
    for num_pasadas in range(len(lista)-1):
        for j in range(num_pasadas):
            if lista[j] > lista[j+1]:
                lista[j],lista[j+1] = lista[j+1],lista[j]
    return lista

```

En los resultados que obtuvimos al aplicar el algoritmo y obtener los tiempos de éste, llegamos a la conclusión de que la gráfica claramente sigue la tendencia de una función cuadrática, por lo tanto podemos afirmar que el ordenamiento sigue correctamente la notación **O(n^2).**



➢ **ORDENAMIENTO QUICKSORT**

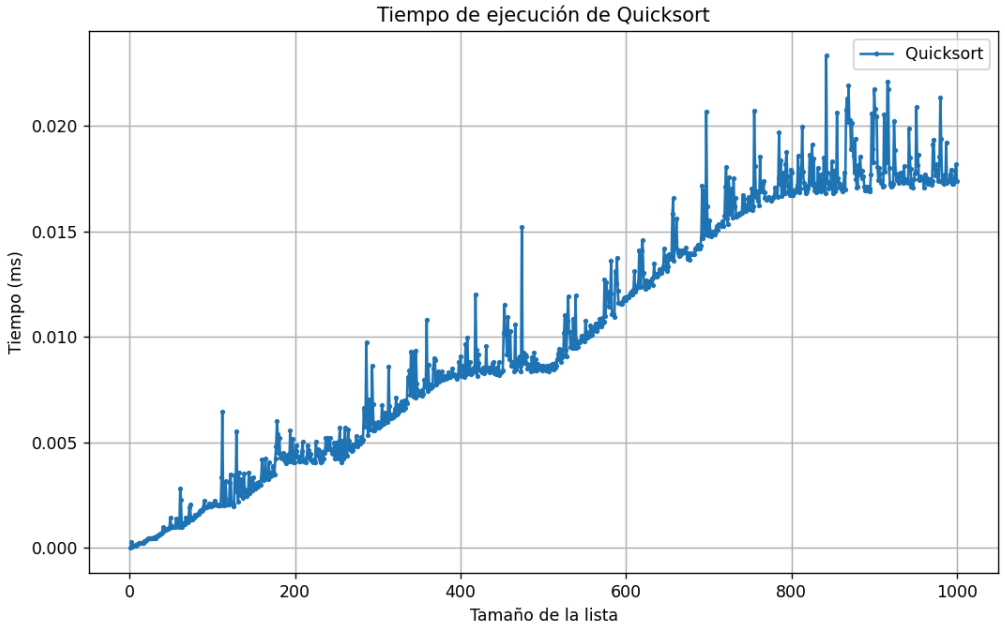
Este ordenamiento consiste en seleccionar un pivote, que puede ser el primer elemento o un elemento estratégico de la lista. Se recorren los elementos usando dos punteros: uno desde el final hacia el comienzo y otro desde el inicio hacia el final (después del pivote). Ambos punteros comparan los elementos con el pivote, intercambiando valores cuando corresponde, de modo que los elementos menores al pivote queden a su izquierda y los mayores a su derecha. Cuando los punteros se cruzan, el pivote se coloca en su posición final. Para elegir un pivote óptimo en este caso se utilizó la función **numpy.median()** de la librería NumPy, calculando la mediana de la lista para encontrar un pivote representativo y balancear las particiones. Esto se debe a que como los números son aleatorios, cada vez que ejecutemos el código, estos van a variar, por ende el pivote tiene que ser un número de acuerdo a los valores que tenga la lista.

Ambas particiones se separan en listas menores al pivote original y mayores al pivote, realizándose de manera recursiva este algoritmo.

```
lista_menores = quicksort(lista[:punt_der])
lista_mayores = quicksort(lista[punt_der + 1:])

return lista_menores + [pivote] + lista_mayores
```

Este ordenamiento tiene en promedio una complejidad **O(nlogn)** y en el peor de los casos, como puede ser cuando se elija como pivote al mayor número de la lista, tendría una complejidad **O(n^2).**



En nuestros resultados aunque la medición muestra variaciones y picos debido a las condiciones de los datos y el entorno de ejecución, la tendencia general sigue el comportamiento esperado de **O(nlogn)** en el caso promedio de quicksort.

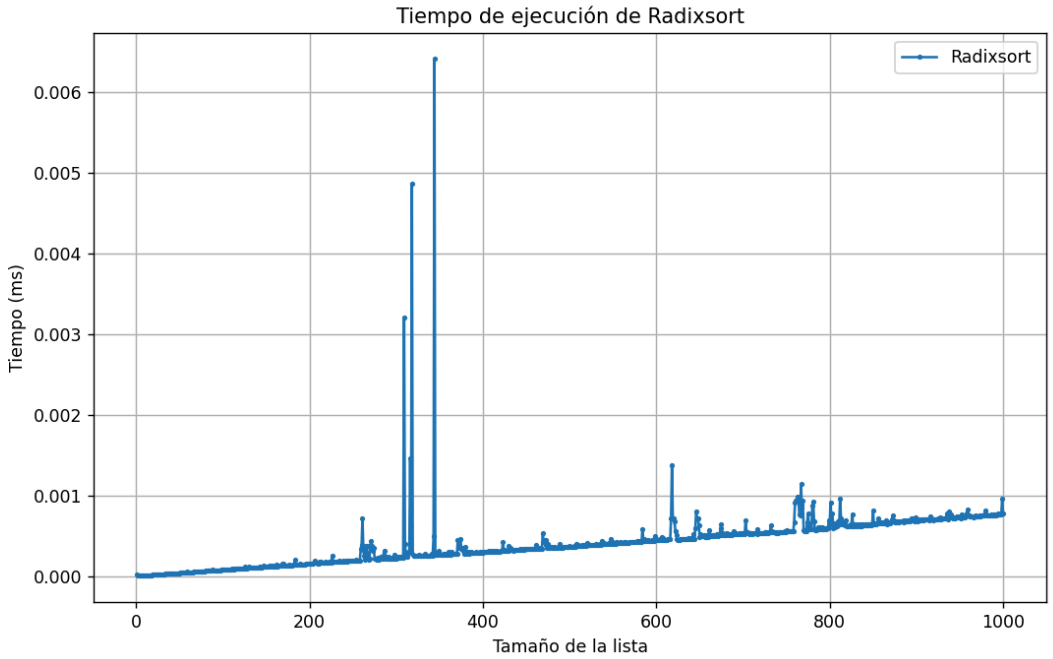
➢ **ORDENAMIENTO RADIX SORT**

Radix Sort es un algoritmo de ordenamiento que procesa los números según sus dígitos, comenzando por el dígito de menor importancia (LSD, *Least Significant Digit*). En este caso, se utilizó el método de **buckets**, que consiste en una lista de sublistas (índices del 0 al 9) donde se almacenan temporalmente los elementos de acuerdo al valor del dígito actual. Luego de procesar cada dígito, los elementos se reconstruyen en una lista ordenada parcialmente, y el proceso se repite para cada posición de dígito hasta completar el ordenamiento.

```
    buckets = [[] for _ in range(10)]  # Crear 10 buckets para cada dígito (0-9)

    for numero in lista:  # Distribuir los números en los buckets según el dígito actual
        digito = (numero // exp) % 10  # Obtener el dígito en la posición actual
        buckets[digito].append(numero)
```

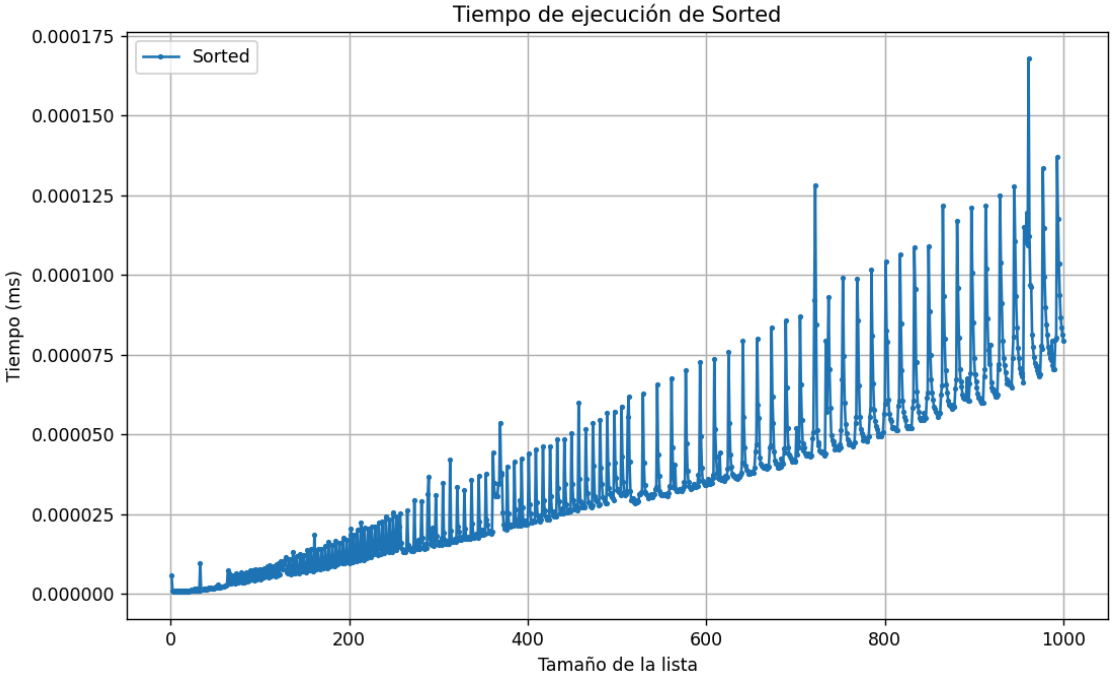
La complejidad de Radix Sort es **O(nk)** tanto en el caso promedio como en el peor caso, siendo **n** el número de elementos y k el número de dígitos. Si **k** es pequeño y constante, Radix Sort puede considerarse un algoritmo de ordenamiento **O(n).**



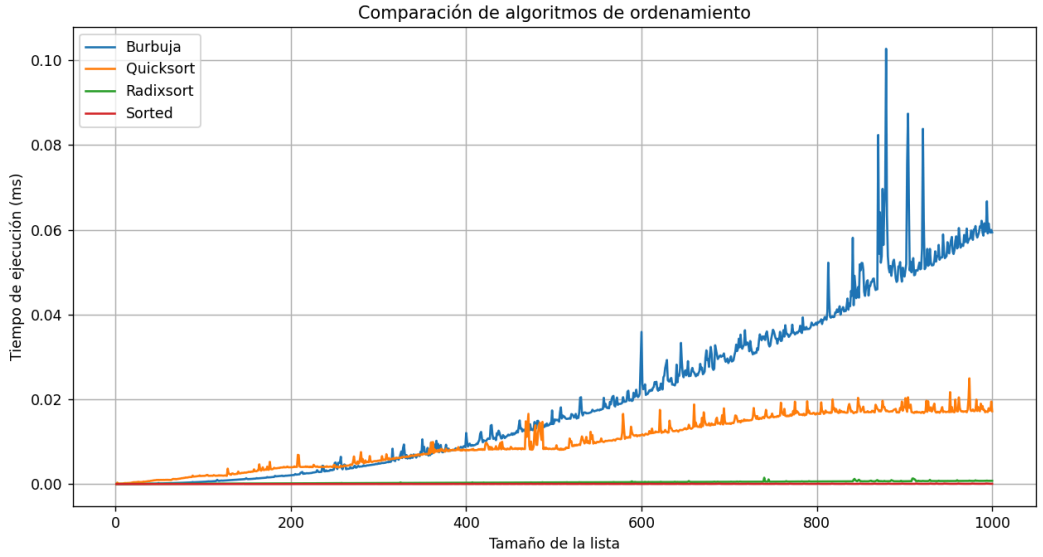
En los resultados que se obtuvieron de igual manera se puede comprobar que el ordenamiento sigue una tendencia lineal o de complejidad **O(n).**

FUNCIÓN SORTED():

En Python, el método **sorted()** utiliza el algoritmo Timsort, un híbrido entre Merge Sort e Insertion Sort. Tiene complejidad **O(nlogn)** en el caso promedio y en el peor caso, y **O(n)** en el mejor caso cuando los datos están casi ordenados.



Al comparar el método **sorted()** con los ordenamientos **burbuja**, **quicksort** y **radixsort**, claramente la función de python sorted() está muy optimizada y se ve en la gráfica su complejidad respecto del resto.



IMPLEMENTACIÓN DE LA LISTA DOBLEMENTE ENLAZADA(LDE).

En este problema lo que nos toca implementar es el TAD Lista doblemente enlazada, el cual permita almacenar elementos de cualquier tipo que sean comparables, como por ejemplo: enteros, flotantes y string. El TAD lista doblemente enlazada es una estructura de datos lineal, la cual se compone de nodos. Cada nodo posee un dato almacenado y dos referencias, una al siguiente nodo y otra al nodo previo.

La implementación en este problema debe respetar cierta especificación lógica.

Primero, creamos la clase Nodo, la cual representa un elemento (dato) y referencia al nodo anterior y al nodo siguiente. Luego creamos la clase **ListaDobleEnlazada** la cuál representa la lista completa con sus respectivos nodos, define el inicializador que crea una lista vacía y luego inicializa los atributos principales de la clase a los cuales guarda con una referencia, cabeza para el primer nodo, cola para el último nodo y **tamanio** para la cantidad de elementos de la lista. Si se pasa un iterable que no está vacío y no es none, se agrega al final de la lista, sino se salta el bloque.

Para agregar elementos al inicio creamos un nuevo nodo que apunta a la referencia anterior (cabeza), actualizamos la referencia con la del nuevo nodo y aumentamos el tamaño. Lo mismo ocurre si lo queremos agregar al final.

Ahora con el método insertar, el cual permite agregar un elemento a una posición específica de la lista doblemente enlazada, si la posición es mayor al tamaño de la lista o es none, agrega el elemento directamente al final de la lista, si es menor a cero larga error, si es cero lo agrega al principio y si es cualquier otra posición recorro la lista desde la “cabeza” hasta llegar a la posición deseada, en donde se crea un nuevo nodo con dicho ítem y se enlaza y se termina aumentando el tamaño de la lista.

Definimos el método extraer, en donde si no le pasamos una posición determinada, elimina el último elemento. Devuelve el elemento extraído.

Definimos el método copiar, donde creamos una nueva lista llamada copia. Vamos a recorrer la lista original desde el primer nodo mientras haya nodos en la lista, y agregamos el elemento del nodo actual al final de la lista copia y retornamos la nueva lista copiada.

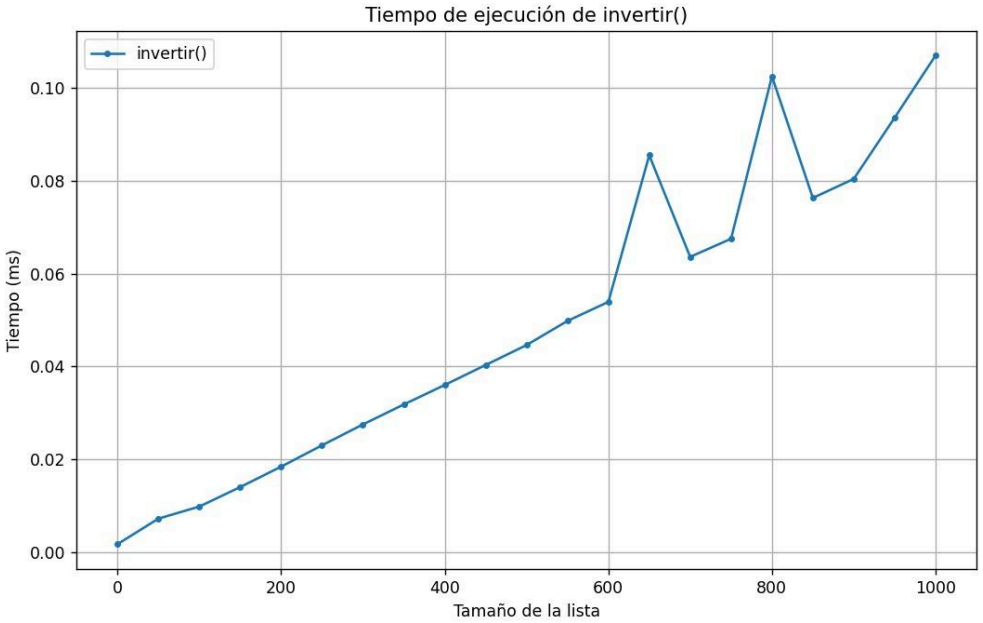
Para el método invertir, empezamos desde el primer nodo e intercambiamos los extremos de la lista, es decir, la cola con la cabeza, siempre y cuando haya nodos en la lista.

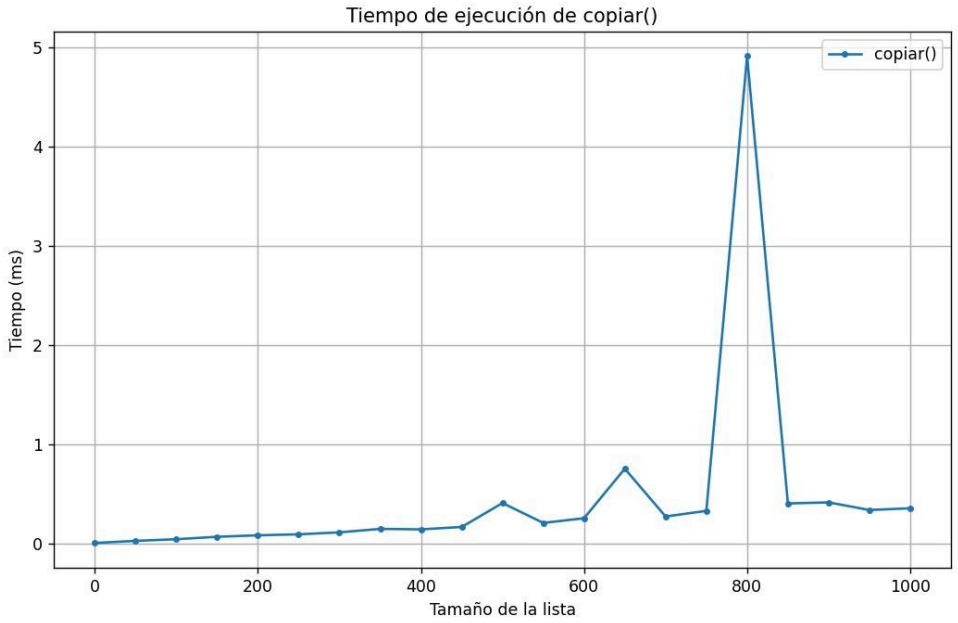
Para el método concatenar, en el cual unimos una lista al final de la otra, primero hacemos una copia de la lista que queremos concatenar, y mientras hay nodos en la lista, empezando desde el primer nodo,agregamos cada elemento al final de la lista y avanzamos. Para finalizar, retorna la lista ya concatenada.

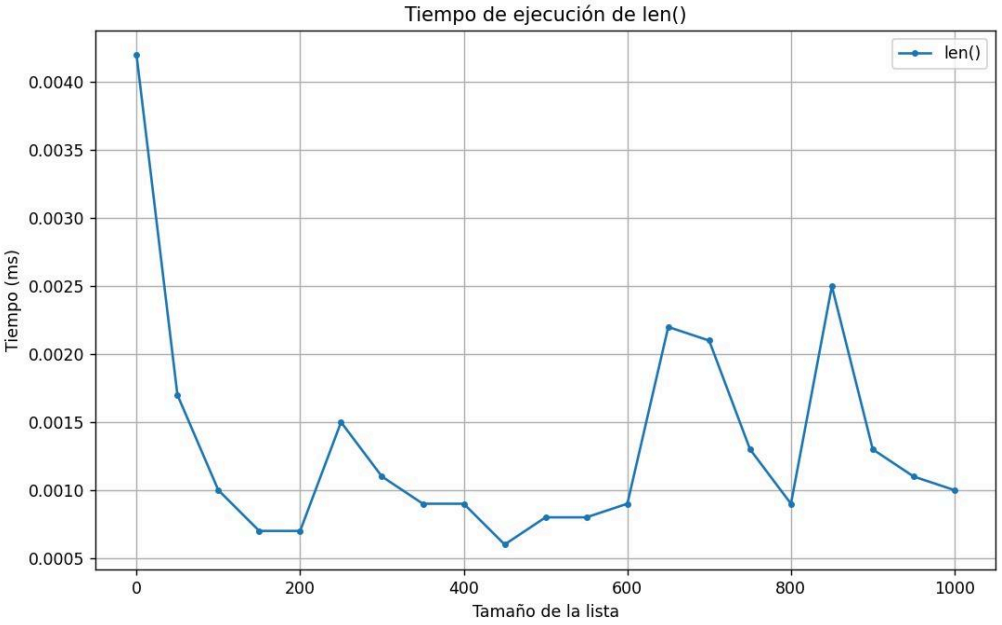
El método __add__ crea una copia de la lista actual, a dicha copia le agrega todos los elementos de la lista que pasamos y al final devuelve la nueva lista combinada. Y para el método __iter__ permite iterar sobre la lista. Lo que hace es comenzar desde el primer nodo de la lista, devuelve el valor del nodo pero la función en vez de terminar, pasa al siguiente nodo

Respecto al Test de Lista Doble Enlazada verificó nuestro código, que su funcionamiento es correcto.

A continuación veremos la complejidad de una ListaDoblementeEnlazada con los métodos **invertir()** que invierte el orden la lista asignando la cabeza a la cola y la cola a la cabeza**, copiar()** devuelve una lista igual **,** y el método **len()** que devuelve la cantidad de elementos de la lista.







IMPLEMENTACIÓN DE LA CLASE MAZO COMO LDE.

Para este problema debemos implementar el juego de cartas **"Guerra"** que es un juego de azar donde el objetivo es ganar todas las cartas.

Para empezar, la cátedra nos brindó el código con el algoritmo del juego guerra, lo que tuvimos que hacer es el código e implementación de la clase Mazo, el cuál hace uso de una ListaDobleEnlazada, clase que importamos desde un archivo llamado LDE.py. Esta clase se usa internamente para almacenar objetos de tipo Carta y realizar las operaciones que se le soliciten.

Para la clase Mazo, se crea una nueva instancia de ListaDobleEnlazada, la cual actuará como la estructura interna para almacenar las cartas de dicho mazo. Primero obtenemos la cantidad de cartas en el Mazo, y después definimos métodos en los cuales, va a agregar una carta al inicio del Mazo, otro en el que va a agregar una carta abajo, es decir, al final del Mazo, en otro extraemos la carta de la parte superior del Mazo, es decir, posición cero. Otro método, sacar carta de abajo, en el que extraemos la última carta del Mazo. Estos dos últimos métodos emiten la excepción DequeEmptyError,
si el mazo está vacío.

Luego, con __str__ devolvemos una representación en cadena del Mazo, en el que unimos cada carta con espacios.

Con los test para la clase de mazo y el test para el juego guerra verificamos que está correctamente implementado nuestro código.

También implementamos la clase carta, en la cual vamos a representar lo que sería una carta individual del mazo, con su valor y su respectivo palo. En este código podemos comparar dos cartas según su valor numérico y también observar si está boca arriba (visible=True) o boca abajo (no visible=False).Para que dichas comparaciones se puedan hacer, las cartas J,Q,K y A toman un valor número.