UNER Facultad de Ingeniería

Licenciatura en Bioinformática

Bioingeniería

TRABAJO PRÁCTICO N°1

ALGORITMOS Y ESTRUCTURAS DE DATOS

Docentes: Rizzato, Juan - Diaz Zamboni, Javier

Alumnos:

Isaac, Priscila Rocio

Jacobo, Nahir

Nista, Nicolás

Fecha de entrega: 02/05

Año lectivo: 2025

Problema 1

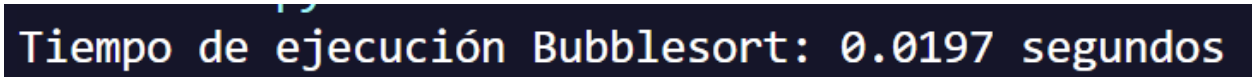
En el desarrollo de lo propuesto en el ejercicio 1, se definió en “modules” por separado los tres algoritmos de ordenamiento pedidos. Para compararlos, por cada vez que se ejecute el código completo, se hace una lista con los tiempos de ejecución y se los pasa a una función que grafica los tiempos de ejecución de cada algoritmo en función de la cantidad de elementos de la lista.

a) Ordenamiento burbuja

El ordenamiento burbuja es un algoritmo definido como una función que toma una lista y la devuelve con sus elementos ordenados. Su mecanismo para ordenar los elementos es comparar los pares adyacentes (ej: elemento 1 y elemento 2, elemento 2 y elemento 3), luego toma una decisión respecto a si los elementos deben ser intercambiados de lugar o mantenerse igual.

Nivel de complejidad: $O(n^2)$
$T(n) = 2 + 3n^2$

Tiempo de ejecución:



b) Ordenamiento quicksort

Es un algoritmo de ordenamiento que toma como argumento a una lista y la devuelve con sus elementos ordenados. Su funcionamiento se basa en tomar un pivote y dividir la lista en sublistas (ej: sublista con elementos menores al pivote, sublista con elementos iguales al pivote y sublista con elementos mayores al pivote). Esta división es aplicada de manera recursiva a todas las sublistas hasta que tengan uno o ningún elemento, logrando el ordenamiento.

Nivel de complejidad: $O(n)$
$T(n) = n + 5$

Tiempo de ejecución:

c) Ordenamiento por residuos (radix sort)

Radix Sort es un algoritmo de ordenamiento no comparativo que ordena enteros procesando sus dígitos de forma individual. Funciona distribuyendo los números en “contenedores” basados en cada dígito, desde el menos significativo hasta el más significativo.

Nivel de complejidad de counting_sort_por_digito: $O(n)$
$T(n) = 8n + 2$

Nivel de complejidad de radixsort: O(n)
T(n) = 3n + n

Nivel de complejidad de la suma de ambos: O(n)
T(n) = 11n + 5

Tiempo de ejecución algoritmo de ordenamiento por residuos (Radix Sort):

```
Tiempo de ejecución Radixsort: 0.0008 segundos
```

d) Sorted

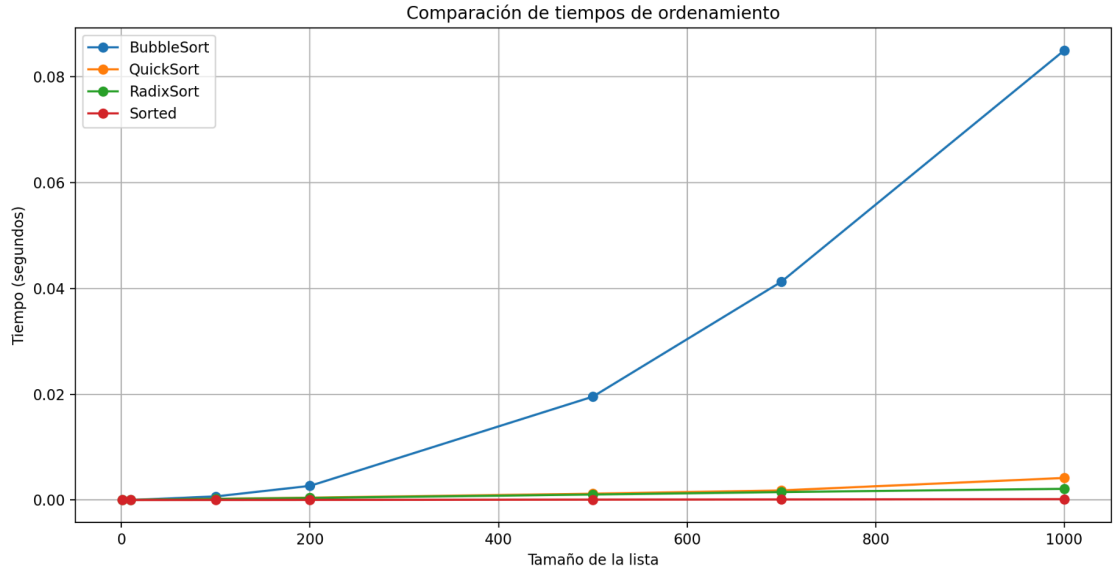
Es una función de Python que ordena los elementos de una lista (u otra colección) y devuelve una nueva lista ordenada, sin modificar la original. Internamente usa un algoritmo optimizado que detecta patrones en los datos para ordenar de forma rápida y eficiente.

Tiempo de ejecución:

```
Tiempo de ejecución Sorted: 0.0000 segundos
```

A priori se esperaría que el algoritmo de ordenamiento quicksort sea el más rápido ya que tiene un orden de complejidad menor.

Pero, al medir los tiempos de ejecución y graficarlos en una misma gráfica, se observa que el método BubbleSort es notablemente más lento que los demás, que QuickSort y RadixSort son muy similares y que sorted es el más veloz de todos.



La función sorted() de Python es más eficiente que otros algoritmos como bubblesort, quicksort o radixsort porque está basada en un algoritmo optimizado para funcionar bien con datos reales. Este algoritmo detecta patrones en la lista, como partes ya ordenadas, y los aprovecha para ordenar más rápido. También funciona con muchos tipos de datos y conserva el orden de los elementos iguales, lo que es útil cuando se trabaja con objetos. Por eso, sorted() es una opción más rápida y versátil para ordenar en Python.

Problema 2

Lista doblemente enlazada: Es una estructura de datos que consiste en un conjunto de nodos enlazados secuencialmente, los cuales suelen ser normalmente registros y que tienen un tamaño fijo. Suelen llamarse estructuras dinámicas porque pueden crearse y destruirse según se vayan necesitando.

Gráfica de los tiempos de ejecución de los métodos len, copiar e invertir aplicados en la Lista Doblemente Enlazada. Los métodos van a ser testeados con listas de tamaños 500, con muertos desde 10.000 hasta 99.999.

Tiempo de ejecución de invertir():

```
invertir() - Tamaño 1000: 0.000073 segundos
```

Tiempo de ejecución de copia():

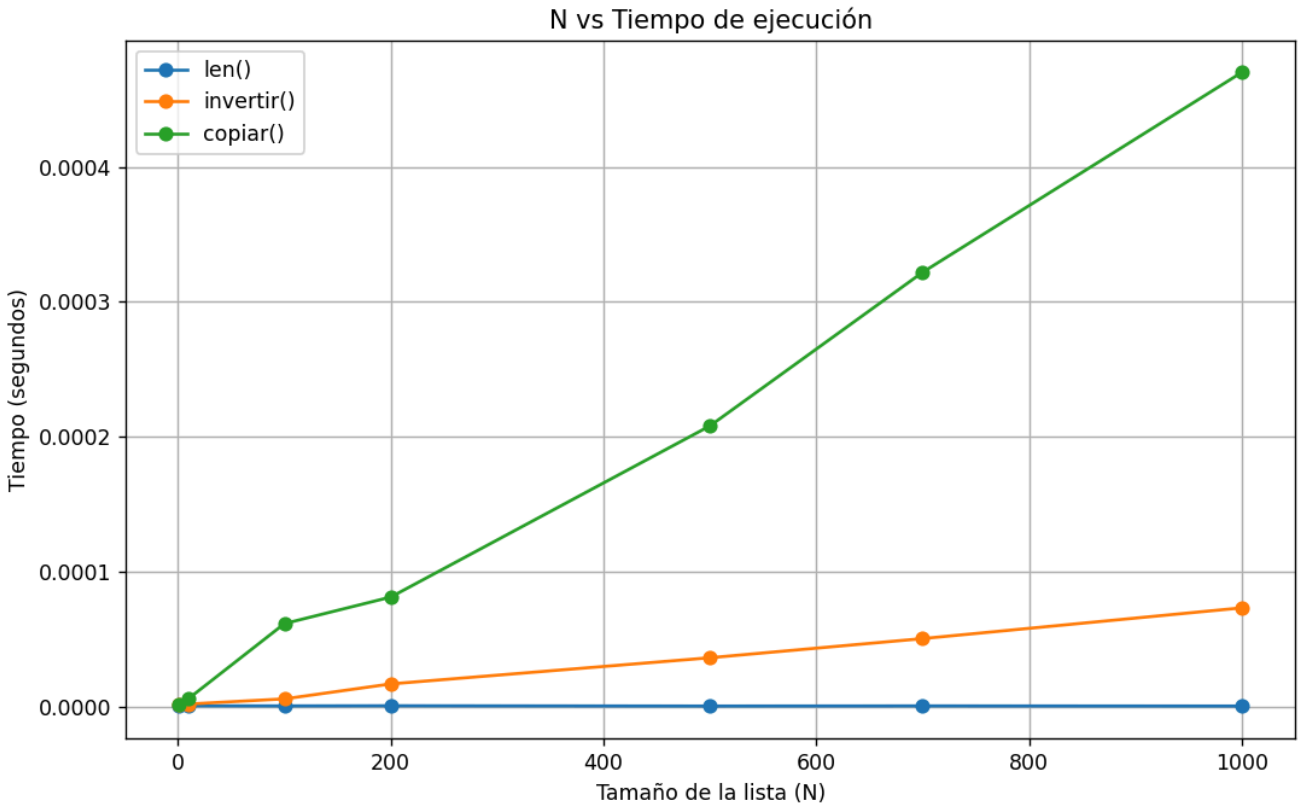
```
copiar() - Tamaño 700: 0.000322 segundos
```

Tiempo de ejecución de la función len():

```
len() - Tamaño 1000: 0.000000 segundos
```

A partir de la gráfica que compara los tiempos de ejecución de los comandos se observa que copiar() e invertir() se comportan de manera similar en lo que respecta a velocidad. La función len() es más rápida en comparación a los anteriores.

Se puede suponer que las funciones copiar() e invertir() tienen un orden O(n) ya que recorren la lista n veces y len() un orden O(1) ya que recorre la lista una sola vez.



Problema 3

Para el desarrollo de lo propuesto en el problema 3 hicimos uso de la clase lista doblemente enlazada creada en el problema 2. Las funciones definidas en dicha clase son especialmente útiles para crear la clase Mazo por las características propias que tendría un mazo, como agregar y extraer.