TP N° 1: Problema 1.

Alumnas: Nuñez Juliana, Morello Milagros.

Facultad de Ingeniería.

Universidad Nacional de Entre Ríos.

Lic. en Bioinformática.

Algoritmos y Estructuras de datos.

Desarrollo:

En este documento se procederá a realizar una breve explicación y análisis de resultados del Problema 1, del trabajo práctico n°1.

Como primera instancia se realizó una investigación acerca de los algoritmos de ordenamiento que la cátedra pidió una implementación, Radix sort, Quick sort y Bubble sort, de los cuales posteriormente se busco ejemplos de funcionamiento y se analizó el cómo mueven los datos con los simuladores proporcionados en el campus virtual.

Se hizo la implementación de los algoritmos dentro de un modelo de funciones que posteiomente es llamado en el archivo main.py, para lo cual también se requirió la creación de un segundo módulo el cual genera una lista de números (mínimo 500, de 5 cifras), para pode sa la misma con los algoritmos a mencionados (no la lista original, cada algoritmo funcionó con una copia de la lista).

Luego de realizar la implementación de los mismos se pidió el análisis del orden de complejidad que cada uno presenta:

A. **Bubble sort:** compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Complejidad: O(n2) en el peor, O(n) en el mejor y promedio de los casos; En el peor de los casos (cuando la lista está en orden inverso), se requieren n−1 pasadas completas sobre la lista y se realizan n−i comparaciones para cada pasada, donde i es el número de la pasada. En consecuencia, se produce una complejidad cuadrática, O(n2).
B. **Quick sort:** Particione la lista en sublistas de elementos menores y mayores utilizando el enfoque de "divide y vencerás". En los mejores (los casos típicos), el pivote divide la lista en dos sublistas de tamaño similar, lo que da una profundidad de recursión de logn, y se realizan n operaciones para dividir la lista. En el peor de los casos, las particiones son desequilibradas si el pivote es siempre el elemento más chico o más grande, lo que resulta en una complejidad de O(n2).
C. **Radix sort:** Ordena los números dígito a dígito (iniciando desde el menos significativo), para cada dígito, el algoritmo agrupa los elementos en "buckets" (cubetas) según el valor del dígito actual y luego los combina en una lista ordenada basada en ese dígito. El número de dígitos del número más grande (d, en este caso 5) y el número de elementos (n) determinan la complejidad. Se realiza una pasada completa por la lista por cada dígito, agrupando los elementos en los cubos. Por lo tanto, el tiempo que lleva agrupar y combinar los elementos es proporcional a d veces el tiempo total, lo que da como resultado O(d×(n+k)). Funciona bien cuando el número de dígitos (d) es menor que el número de elementos (n).

Un avez eso fue realizado, se hizo una investigación de una función incluida en la biblioteca de python, **sorted,** la cual realiza la misma actividad que los algoritmos anteriores, odena las listas, pero funciona de otra manera:

- Se utiliza Timsort para la función sorted: El algoritmo Timsort es un híbrido que utiliza Merge Sort e Insertion Sort. Primero, encuentra sublistas ("runs") que ya están ordenadas o casi ordenadas

y luego las combina utilizando un método eficiente de combinación de clasificación. Está diseñado para funcionar con listas parcialmente ordenadas.

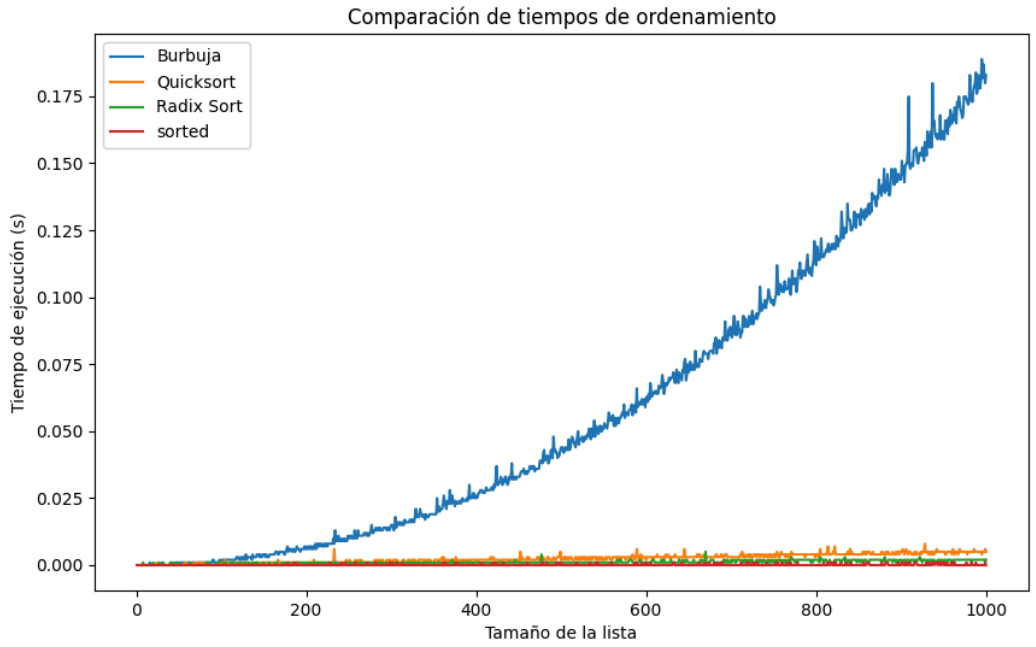
- La complejidad es O(nlogn) en los casos peores, promedios y mejores. Justificación: Merge Sort y Insertion Sort se combinan para una combinación eficiente.
- Beneficios:
  - La estabilidad es aquella que mantiene el orden relativo de los elementos iguales.
  - Eficiencia en Casos Reales: Cuando se trata de listas que ya están parcialmente ordenadas, lo que es común en datos reales, es extremadamente efectivo.

También se realizó una medición del tiempo de ejecución que cada uno tenía (tanto sorted como los algoritmos implementados) con el siguiente código:

```
import time
import random
from algoritmos import ord_burbuja, quicksort, radix_sort

# Funciones de ordenamiento ya definidas (ord_burbuja, quicksort, radix_sort)

# Función para medir el tiempo de ejecución de cada algoritmo
def medir_tiempos():
    tams = range(1, 1001)
    tiempos_burbuja = []
    tiempos_quicksort = []
    tiempos_radix = []
    tiempos_sorted = []

    for tam in tams:
        lista = [random.randint(10000, 99999) for _ in range(tam)]

        # Medir tiempo para burbuja
        inicio = time.time()
        ord_burbuja(lista.copy())
        tiempos_burbuja.append(time.time() - inicio)

        # Medir tiempo para quicksort
        inicio = time.time()
        quicksort(lista.copy())
        tiempos_quicksort.append(time.time() - inicio)

        # Medir tiempo para radix sort
        inicio = time.time()
        radix_sort(lista.copy())
        tiempos_radix.append(time.time() - inicio)
```

```
        # Medir tiempo para sorted
        inicio = time.time()
        sorted(lista.copy())
        tiempos_sorted.append(time.time() - inicio)
```

Y con esos datos se realizó la siguiente gráfica (utilizando matplotlib):



Conclusiones y resultados:

Tiempos de Ejecución:
Como era de esperar, el Ordenamiento Burbuja mostró un tiempo de ejecución considerablemente más largo para listas de mayor tamaño debido a su complejidad O(n2). Quicksort y sorted fueron significativamente más rápidos, y Radix Sort también funciona bien para listas de números de cinco dígitos.
Comparación de complejidad:
Sorted (Timsort) y el Quicksort tienen una complejidad promedio de O(nlogn), lo que los hace mucho más eficientes que el ordenamiento burbuja. Radix Sort funciona bien en ciertas situaciones, especialmente cuando el rango de números es limitado. Bubble sort no es tan eficiente debido a todas las veces que la lista debe ser recogida y al tiempo que ello toma.

Para listas pequeñas o casi ordenadas, sorted es la opción más eficiente. Para listas grandes con números de un rango limitado, Radix Sort puede ser más eficiente.