Facultad de Ingeniería
UNER

Informe sobre la implementación y análisis de algoritmos de ordenamiento

Algoritmos y Estructura de Datos

Integrantes:

- Zapata Mariana Gabriela
- Weimer Valentin
- Kerbs Javier

2° Cuatrimestre, 2025

Problema 3: Algoritmos de Ordenamiento.

1. Introducción.

El objetivo de la actividad es aplicar los conceptos de algoritmos de ordenamiento y de tipos abstractos de datos (TAD) en Python. Para ello se implementan distintos métodos de ordenamiento, se analizan sus características y se comparan sus rendimientos en términos de tiempo de ejecución. Asimismo, se incorporan validaciones mediante manejo de excepciones y se utilizan pruebas unitarias para comprobar el correcto funcionamiento de las implementaciones. Finalmente, se contrasta el rendimiento de los algoritmos implementados con la función integrada sorted() de Python.

2. Algoritmos a implementar

Se desarrollan tres algoritmos clásicos de ordenamiento:

1. **Ordenamiento Burbuja (Bubble Sort).**
   Consiste en recorrer la lista comparando pares de elementos adyacentes y realizando intercambios cuando están en orden incorrecto. Repite este proceso hasta que toda la lista queda ordenada. Es simple de implementar, pero muy ineficiente en listas grandes.
   Complejidad temporal:
   - Mejor caso: O(n)
   - Promedio y peor caso: O(n^2)

```python
def ordenamiento_burbuja_corto(lista):
    intercambiado = True
    num_pasadas = len(lista)-1
    while num_pasadas > 0 and intercambiado:
        intercambiado = False
        for j in range(num_pasadas):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
                intercambiado = True
        num_pasadas -= 1
    return lista
```

intercambiado = True se usa como **bandera** para detectar si en la pasada hubo algún intercambio.

Si en una pasada completa no hubo intercambios (intercambiado = False), significa que la lista ya está ordenada, y se puede cortar antes de seguir gastando tiempo.
Esto puede ahorrar muchas iteraciones cuando la lista está **casi ordenada**.

2. **Ordenamiento Quicksort.**

   Utiliza la estrategia de “divide y vencerás”. Elige un elemento pivote y divide la lista en dos sublistas: una con los elementos menores y otra con los mayores al pivote. El proceso se repite recursivamente sobre cada sublista.
   Complejidad temporal:

   - Promedio: O(n log n)
   - Peor caso: O(n^2), cuando la elección del pivote es desfavorable.

3. **Ordenamiento por residuos (Radix Sort).**

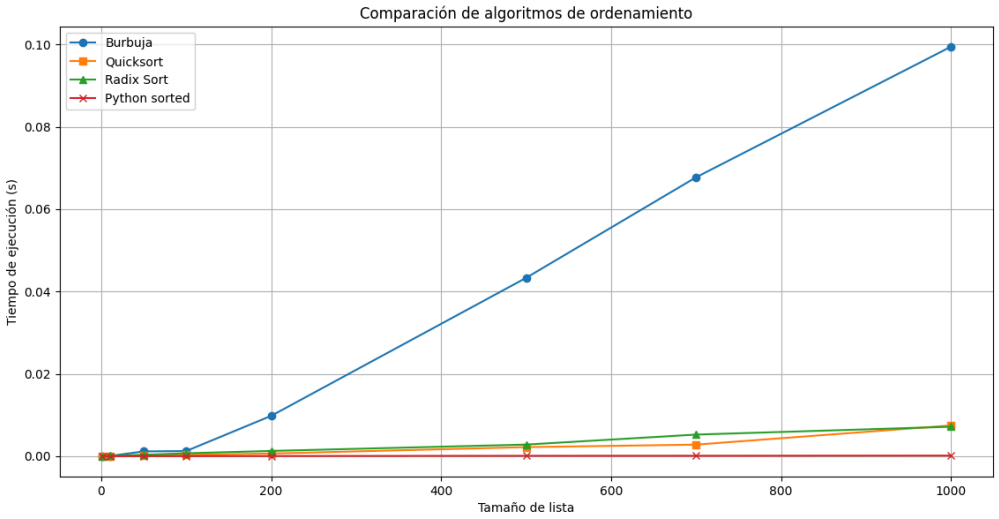
   Ordena los números en función de sus dígitos, comenzando por el menos significativo (unidades) hasta el más significativo. Para ello utiliza un proceso de distribución en cubetas según el valor del dígito actual.
   Este algoritmo no realiza comparaciones directas entre números, lo que lo hace muy eficiente para listas de enteros.
   Complejidad temporal: O(n·k), donde k es la cantidad de dígitos de los números. Para números de 5 dígitos, k es constante, por lo que el comportamiento es cercano a O(n)

3. Experimentos con listas aleatorias.

Se generan listas de números enteros de cinco dígitos, con una longitud mínima de 500 elementos, empleando la función random.randint().
Sobre estas listas se aplican los tres algoritmos y se verifica la exactitud de los resultados.

Posteriormente, se mide el tiempo de ejecución de los algoritmos para listas cuyo tamaño varía entre 1 y 1000 elementos. Para ello se utiliza la biblioteca time de Python, registrando la duración de cada ejecución.

4. Resultados.



5. Conclusión.

- Para listas grandes y datos genéricos, la función `sorted()` es la opción más eficiente, aprovechando *Timsort*.
- Para datos numéricos con pocos dígitos, *Radix Sort* es muy eficiente.
- *Bubble Sort* es útil solo en casos muy específicos, debido a su mala complejidad de tiempo en grandes volúmenes de datos.
- *Quicksort* ofrece un buen equilibrio entre eficiencia y complejidad en promedio, pero puede sufrir en el peor caso debido a la elección del pivote.