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

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.

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(lista):
    for num_pasadas in range(len(lista)-1, 0, -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
```

for num_pasadas in range(len(lista)-1, 0, -1): se hace un número decreciente de pasadas.

- En la **1ª pasada** se comparan todos los elementos hasta el final.
- En la **2ª pasada** ya no hace falta revisar el último, porque ya quedó en su lugar.

for j in range(num_pasadas): recorre la lista comparando pares de elementos **adyacentes**.

if lista[j] > lista[j+1]: si el elemento actual es mayor que el siguiente, **se intercambian**.

```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)

Pruebas unitarias

Con el fin de comprobar el correcto funcionamiento de las implementaciones, se emplean pruebas unitarias utilizando el módulo unittest. Estas pruebas verifican que los algoritmos devuelvan listas efectivamente ordenadas y que se comporten de manera consistente ante casos especiales, como listas vacías o con un solo elemento.

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.

Resultados y análisis de complejidad

Problema 3: Algoritmos de Ordenamiento.

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.

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(lista):
    for num_pasadas in range(len(lista)-1, 0, -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
```

for num_pasadas in range(len(lista)-1, 0, -1): se hace un número decreciente de pasadas.

- En la **1ª pasada** se comparan todos los elementos hasta el final.
- En la **2ª pasada** ya no hace falta revisar el último, porque ya quedó en su lugar.
- Y así sucesivamente.

for j in range(num_pasadas): recorre la lista comparando pares de elementos **adyacentes**.

if lista[j] > lista[j+1]: si el elemento actual es mayor que el siguiente, **se intercambian**.

```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)

Tipos Abstractos de Datos y manejo de excepciones.

Los algoritmos se encapsulan en funciones y estructuras que permiten manipular listas de manera controlada.

Se utilizan mecanismos de manejo de excepciones (try/except) para validar los datos de entrada, asegurando que las listas contengan únicamente elementos comparables. De esta forma se evitan errores de ejecución al intentar ordenar datos incompatibles.

Pruebas unitarias.

Con el fin de comprobar el correcto funcionamiento de las implementaciones, se emplean pruebas unitarias utilizando el módulo unittest. Estas pruebas verifican que los algoritmos devuelvan listas efectivamente ordenadas y que se comporten de manera consistente ante casos especiales, como listas vacías o con un solo elemento.

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.

Resultados y análisis de complejidad.

Complejidad.

De manera teórica:

- **Ordenamiento Burbuja** → O(n2) complejidad cuadrática (se realizan n−1 pasadas, y en cada pasada se hacen hasta n−i comparaciones).

- **Ordenamiento Quicksort** → Promedio O(nlogn)
  → Peor caso O(n2) (cuando la lista ya está ordenada y se elige siempre el peor pivote).

- **Ordenamiento Radix Sort** → O(d · (n+k)) (d es la cantidad de dígitos y k el rango de valores posibles por dígito).

- **Ordenamiento Sorted** → Promedio O(nlogn)
  → Mejor caso O(n) (si la lista ya está parcialmente ordenada).

Los resultados obtenidos fueron (...),

Sorted().

La funcion sorted() ordena cualquier iterable devolviendo una nueva lista ordenada, sin modificar la original. Su funcionamiento se basa en el algoritmo Timsort, que es una combinación de merge sort y insertion sort, donde primero se hace un deteccion de runs (sublistas ya ordenadas) lo que permite ahorrar comparaciones por el ordenamiento parcial existente, luego se ordenan runs cortos usando insertion sort y por último se fusionan los runs.

Conclusiones.

Conclusión

En este informe, se implementaron y analizaron tres algoritmos clásicos de ordenamiento: *Bubble Sort*, *Quicksort* y *Radix Sort*, junto con la función integrada `sorted()` de Python, para comparar su rendimiento en términos de eficiencia y tiempo de ejecución.

1. **Bubble Sort**:
    - Es un algoritmo sencillo pero ineficiente, especialmente para listas grandes. Su complejidad O(n²) lo hace poco práctico cuando se trabaja con grandes volúmenes de datos, aunque puede ser útil en casos donde los datos ya están casi ordenados.
2. **Quicksort**:
    - Uno de los algoritmos más eficientes en promedio, con una complejidad O(n log n). Sin embargo, su rendimiento puede degradarse a O(n²) en el peor de los casos, especialmente si la elección del pivote es mala. A pesar de esto, su rendimiento en listas grandes es generalmente superior a otros algoritmos de ordenamiento.
3. **Radix Sort**:
    - Este algoritmo es altamente eficiente para listas de enteros con un número limitado de dígitos (como en el caso de los números de 5 dígitos en este experimento). Su complejidad O(n·k) lo hace especialmente útil cuando el rango de valores posibles es limitado, pero no es tan efectivo para datos más generales o cuando los valores tienen un número variable de dígitos.
4. **Función `sorted()` de Python**:
    - La función integrada `sorted()` de Python, basada en el algoritmo *Timsort*, se destacó por su eficiencia y rendimiento general. Timsort combina las

ventajas de *Merge Sort* e *Insertion Sort*, logrando un comportamiento de O(n log n) en la mayoría de los casos, y optimizando los casos en los que la lista ya está parcialmente ordenada.

Conclusión general:

- 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.