Trabajo Integrador: Programación – Algoritmos de Busqueda y Ordenamiento.

Alumnos:

Christian Bustamante – clb.cristian@gmail.com

Hernán Casalderrey – hjcasalderrey@outlook.com.ar

Materia: Programación 1

Comisión: 6

Profesor: Prof. Cinthia Rigoni

Tutor: Prof. Oscar Londero

Fecha de Entrega: 9 de junio de 2025

Índice

Introducción

En el presente trabajo práctico de la asignatura Programación 1 se abordará el estudio y la implementación de algoritmos fundamentales de búsqueda y ordenamiento utilizando el lenguaje de programación Python. Estos algoritmos constituyen la base de numerosas aplicaciones informáticas y son esenciales para el procesamiento eficiente de datos.

El objetivo principal del trabajo es analizar y comparar el comportamiento de dos algoritmos de búsqueda —búsqueda lineal y búsqueda binaria— y cuatro algoritmos de ordenamiento —Bubble Sort, Insertion Sort, Selection Sort y Quick Sort— en un contexto práctico. Para ello, se desarrollará un caso de estudio en el que se generará un diccionario de clientes con identificadores únicos (ID) y nombres, ambos generados aleatoriamente.

A partir de este conjunto de datos, se extraerán los ID de los clientes para ser ordenados y posteriormente utilizados en operaciones de búsqueda. La elección del algoritmo de ordenamiento dependerá de la cantidad de registros generados: si el número de elementos es menor o igual a 100, se aplicará el algoritmo Bubble Sort por su simplicidad y facilidad de implementación; en cambio, si el número de elementos supera los 100, se utilizará Quick Sort, dada su mayor eficiencia en grandes volúmenes de datos.

Este enfoque permitirá no solo poner en práctica los conceptos teóricos aprendidos en clase, sino también observar empíricamente las diferencias de rendimiento y aplicabilidad de cada algoritmo en función del tamaño de los datos y del tipo de operación a realizar.

Marco Teórico

1. Algoritmos de Búsqueda

Los algoritmos de búsqueda permiten localizar un elemento dentro de una estructura de datos. En este trabajo se analizarán dos tipos:

- **Búsqueda Lineal**: Es el método más simple. Consiste en recorrer secuencialmente la estructura de datos comparando cada elemento con el valor buscado. Su complejidad temporal es $O(n)$, lo que la hace poco eficiente para grandes volúmenes de datos, pero útil en listas no ordenadas.
- **Búsqueda Binaria**: Requiere que los datos estén previamente ordenados. Divide repetidamente el conjunto en mitades, comparando el valor buscado con el elemento central. Su complejidad es $O(\log n)$, lo que la convierte en una opción mucho más eficiente que la búsqueda lineal en listas ordenadas.

2. Algoritmos de Ordenamiento

El ordenamiento de datos es una operación clave en informática, ya que permite optimizar búsquedas y mejorar la organización de la información. En este trabajo se implementarán cuatro algoritmos:

- **Bubble Sort**: Compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Es fácil de implementar pero ineficiente para grandes volúmenes de datos, con una complejidad de $O(n2)$.
- **Insertion Sort**: Construye la lista ordenada de a un elemento por vez, insertando cada nuevo elemento en su posición correcta. También tiene una complejidad de $O(n2)$, aunque puede ser más eficiente que Bubble Sort en listas parcialmente ordenadas.
- **Selection Sort**: Selecciona el elemento más pequeño del conjunto no ordenado y lo intercambia con el primero, repitiendo el proceso para el resto de la lista. Su complejidad también es $O(n2)$.
- **Quick Sort**: Es un algoritmo de ordenamiento eficiente que utiliza el enfoque de divide y vencerás. Selecciona un pivote y particiona la lista en dos sublistas, una con elementos menores y otra con elementos mayores al pivote. Su complejidad promedio es $O(n \log n)$, aunque en el peor caso puede ser $O(n2)$.

3. Estructuras de Datos Utilizadas

Para este trabajo se utilizará un **diccionario de Python**, donde cada entrada representará un cliente con un ID (clave) y un nombre (valor). Esta estructura permite un acceso eficiente a los datos y facilita la extracción de los ID para su posterior ordenamiento y búsqueda.

4. Criterio de Selección de Algoritmos

Se establecerá una lógica condicional para seleccionar el algoritmo de ordenamiento según la cantidad de registros generados:

- Si hay **100 o menos elementos**, se utilizará **Bubble Sort** por su simplicidad.
- Si hay **más de 100 elementos**, se aplicará **Quick Sort** por su eficiencia en grandes volúmenes de datos.

Explicación del Caso Práctico

Este caso práctico tiene como objetivo aplicar algoritmos de búsqueda y ordenamiento en un contexto simulado de gestión de clientes, utilizando Python. A continuación se describen los pasos principales del código:

1. Generación de Datos

Se utiliza la función **generar_clientes(99)** o **generar_clientes(101)** para crear un diccionario de clientes con 99 o 101 entradas (para simular el tamaño del diccionario). Cada entrada contiene un nombre aleatorio como clave y un ID numérico aleatorio como valor. Esto simula una pequeña base de datos de clientes.

2. Extracción de IDs

Se extraen únicamente los **IDs** de los clientes (los valores del diccionario) y se almacenan en una lista llamada ids.

3. Ordenamiento Condicional

Dependiendo de la cantidad de elementos en la lista de IDs:

- Si hay **100 o menos elementos**, se utiliza el algoritmo **Bubble Sort**.
- Si hay **más de 100 elementos**, se utiliza **Quick Sort**.

Esto permite aplicar un criterio de eficiencia: Bubble Sort es simple pero lento, mientras que Quick Sort es más eficiente para listas grandes.

4. Búsqueda Binaria

Una vez ordenada la lista de IDs, se selecciona un ID específico (el que está en la posición **25**) y se realiza una **búsqueda binaria** para encontrar su posición en la lista ordenada.

5. Resultado de la Búsqueda

Si el ID es encontrado, se muestra su índice en la lista ordenada. Luego, se recorre el diccionario original para encontrar el **nombre del cliente** correspondiente a ese ID.

Objetivo del Caso Práctico

Este ejercicio permite:

- Aplicar algoritmos de ordenamiento y búsqueda en un entorno simulado.
- Comparar el rendimiento de distintos métodos según el tamaño de los datos.
- Integrar estructuras de datos como diccionarios y listas.
- Desarrollar lógica condicional para seleccionar algoritmos apropiados.

Metodología Utilizada

Para el desarrollo del trabajo práctico se adoptó una metodología modular y didáctica, orientada tanto a la reutilización del código como a la claridad en la presentación de los conceptos. El proceso se dividió en tres componentes principales:

1. Estructura del Proyecto

Se organizaron los archivos en tres módulos independientes, con el objetivo de mantener una arquitectura limpia y reutilizable:

- **Archivo de Métodos (Metodos.py)**: Contiene la implementación de los algoritmos de ordenamiento (Bubble Sort, Quick Sort) y búsqueda (Búsqueda Binaria). Esta separación permite reutilizar los algoritmos en otros proyectos sin necesidad de reescribir el código.
- **Archivo de Generación de Datos (Clientes.py)**: Incluye la función generar_clientes, que simula una base de datos generando un diccionario con nombres e IDs aleatorios. Esta función permite crear distintos escenarios de prueba de forma dinámica.
- **Archivo Principal (TP.py)**: Es el script que integra los módulos anteriores. Aquí se genera el conjunto de datos, se extraen los IDs, se ordenan según la cantidad de registros, y se realiza una búsqueda binaria sobre la lista ordenada. También se imprime el resultado de la búsqueda y el cliente correspondiente.

2. Explicaciones Didácticas

Para facilitar la comprensión del trabajo, se realizaron tres explicaciones en video:

- **Explicación de los Métodos de Búsqueda**: Presentada mediante una presentación en PowerPoint, donde se detallan los principios teóricos y el funcionamiento de la búsqueda lineal y binaria.
- **Explicación de los Métodos de Ordenamiento**: También desarrollada en PowerPoint, se explican los algoritmos Bubble Sort, Insertion Sort, Selection Sort y Quick Sort, incluyendo sus ventajas, desventajas y complejidades computacionales.
- **Explicación del Código en Visual Studio Code**: Se grabó una demostración práctica del código en funcionamiento, explicando paso a paso cómo se integran los módulos y cómo se ejecutan las operaciones de ordenamiento y búsqueda.

Resultados Obtenidos

A partir de la ejecución del código desarrollado, se obtuvieron los siguientes resultados:

1. Generación de Datos

Se generó exitosamente un diccionario de 101 clientes, cada uno con un nombre aleatorio y un ID numérico único. Esto permitió simular una base de datos realista para aplicar los algoritmos de ordenamiento y búsqueda.

2. Ordenamiento de IDs

Dado que la cantidad de registros fue mayor a 100, se aplicó el algoritmo **Quick Sort** para ordenar los IDs de los clientes. El resultado fue una lista ordenada de forma ascendente, lo que permitió optimizar la posterior búsqueda binaria.

3. Búsqueda Binaria

Se seleccionó un ID específico (el que se encontraba en la posición 25 de la lista ordenada) y se realizó una búsqueda binaria sobre la lista. El algoritmo localizó correctamente el ID y devolvió su índice dentro de la lista ordenada.

4. Identificación del Cliente

Una vez encontrado el ID, se recorrió el diccionario original para identificar el nombre del cliente correspondiente. El sistema mostró correctamente el nombre asociado al ID buscado, validando así la integridad de los datos y la efectividad del proceso.

5. Validación Visual

Durante la ejecución del programa, se imprimieron en consola:

- El diccionario completo de clientes generados.
- La lista de IDs ordenados.
- El resultado de la búsqueda binaria.
- El nombre del cliente correspondiente al ID encontrado.

Esto permitió verificar visualmente que cada etapa del proceso se ejecutó correctamente y que los algoritmos aplicados funcionaron según lo esperado.

```
Clientes generados: {'Marta0': 71, 'Carlos1': 76, 'Ana2': 99, 'Juan3': 79, 'Maria4': 20, 'Juan5': 92, 'Marta6': 38, 'Ana7': 70, 'Marta8': 437, 'Juan9': 2, 'Pedro10': 43, 'Jose11': 75, 'Carlos12': 686, 'Carlos1
3': 51, 'Marta14': 1, 'Marta15': 5, 'Elena16': 48, 'Luis17': 61, 'Marta18': 53, 'Luis19': 42, 'Juan20': 97, 'Carlos21': 52, 'Luis22': 17, 'Jose23': 47, 'Elena24': 713, 'Elena25': 94, 'Elena26': 80, 'Marta27':
3, 'Laura28': 87, 'Ana29': 31, 'Elena30': 400, 'Jose31': 15, 'Elena32': 12, 'Jose33': 11, 'Luis34': 55, 'Carlos35': 37, 'Luis36': 81, 'Ana37': 69, 'Juan38': 298, 'Luis39': 205, 'Jose40': 57, 'Elena41': 19, 'Ca
rlos42': 13, 'Marta43': 66, 'Elena44': 521, 'Marta45': 187, 'Elena46': 973, 'Laura47': 64, 'Ana48': 34, 'Luis49': 23, 'Luis50': 982, 'Maria51': 16, 'Elena52': 90, 'Laura53': 757, 'Carlos54': 260, 'Marta55': 82
, 'Carlos56': 6, 'Maria57': 988, 'Ana58': 40, 'Luis59': 32, 'Carlos60': 799, 'Carlos61': 62, 'Pedro62': 619, 'Laura63': 339, 'Juan64': 9, 'Luis65': 88, 'Juan66': 271, 'Juan67': 35, 'Luis68': 893, 'Maria69': 84
6, 'Juan70': 461, 'Marta71': 45, 'Elena72': 717, 'Ana73': 447, 'Ana74': 10, 'Elena75': 275, 'Carlos76': 725, 'Laura77': 505, 'Carlos78': 641, 'Carlos79': 608, 'Juan80': 29, 'Carlos81': 24, 'Juan82': 245, 'Carl
os83': 573, 'Luis84': 44, 'Pedro85': 291, 'Jose86': 478, 'Jose87': 77, 'Luis88': 27, 'Ana89': 91, 'Juan90': 95, 'Jose91': 96, 'Laura92': 58, 'Maria93': 63, 'Carlos94': 155, 'Pedro95': 93, 'Laura96': 22, 'Pedro
97': 198, 'Luis98': 687, 'Maria99': 947, 'Laura100': 425}
----------------------------------
IDs ordenados: [1, 2, 3, 5, 6, 9, 10, 11, 12, 13, 15, 16, 17, 19, 20, 22, 23, 24, 27, 29, 31, 32, 34, 35, 37, 38, 40, 42, 43, 44, 45, 47, 48, 51, 52, 53, 55, 57, 58, 61, 62, 63, 64, 66, 69, 70, 71, 75, 76, 77,
 79, 80, 81, 82, 87, 88, 90, 91, 92, 93, 94, 95, 96, 97, 99, 155, 187, 198, 205, 245, 260, 271, 275, 291, 298, 339, 400, 425, 437, 447, 461, 478, 505, 521, 573, 608, 619, 641, 686, 687, 713, 717, 725, 757, 799
, 846, 893, 947, 973, 982, 988]
----------------------------------
El cliente con ID 38 fue encontrado en el índice 25 de la lista ordenada.
El cliente con ID 38 es: Marta6
```

Conclusiones

A lo largo de este trabajo práctico se logró cumplir con los objetivos propuestos, aplicando de manera efectiva algoritmos de búsqueda y ordenamiento en un entorno simulado de gestión de datos. Las principales conclusiones son:

- **Modularidad y reutilización**: La separación del código en módulos independientes (métodos, generación de datos y ejecución principal) facilitó la organización, comprensión y reutilización del código.
- **Aplicación práctica de algoritmos**: Se implementaron y compararon distintos algoritmos de ordenamiento (Bubble Sort y Quick Sort) y búsqueda (búsqueda binaria), observando sus diferencias de rendimiento según el tamaño de los datos.
- **Criterio de selección eficiente**: La elección condicional del algoritmo de ordenamiento en función del tamaño de la lista permitió aplicar un enfoque eficiente y adaptativo, simulando decisiones reales en programación.
- **Visualización y validación**: La impresión de resultados en consola y la explicación en video permitieron validar el correcto funcionamiento del sistema y facilitaron la comprensión del proceso por parte de terceros.
- **Aprendizaje significativo**: El trabajo integró teoría y práctica, fortaleciendo el entendimiento de estructuras de datos, algoritmos y buenas prácticas de programación en Python.

Bibliografía Académica

1. UTN TUPaD 2025 Teoria de Busqueda y Ordenamiento: https://tup.sied.utn.edu.ar/mod/resource/view.php?id=3434.
2. **Avalos, M. S.** (2025). *Apunte No. 10: Búsqueda y Ordenamiento*. Taller de Programación. Disponible en: https://tute-avalos.com/static/python/10%20-%20Algoritmos%20de%20b%C3%BAsqueda%20y%20ordenamiento.pdf.
3. **Algar Díaz, M. J. & Fernández de Sevilla Vellón, M.** (2019). *Introducción práctica a la programación con Python*. Universidad de Alcalá. Disponible en: https://elibro.net/es/lc/unapec/titulos/124259.
4. **4Geeks Academy**. *Algoritmos de Ordenamiento y Búsqueda en Python: Optimizando la gestión de datos*. Disponible en: https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python.

Anexo:

Repositorio: https://github.com/hcasalderrey/UTN-TUPaD-P1-TPIntegrador