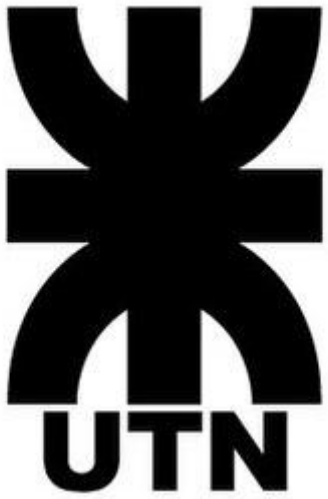

Trabajo Práctico Integrador Programación I

Alumnos:

- Alvarez, Lucas Gabriel.
- Atencio, Jorge.

Materia: Programación I.

Tema: Algoritmos de búsqueda y ordenamiento

Fecha de entrega: 20/06/2025

Índice

1. Introducción

En el mundo de la informática, los algoritmos representan el corazón de la programación. Se trata de secuencias finitas de instrucciones bien definidas que permiten resolver problemas de forma lógica, estructurada y eficiente. Entre las operaciones fundamentales en programación se destacan dos que son esenciales para la manipulación y análisis de datos: la búsqueda y el ordenamiento.
En el presente trabajo exploramos los principales algoritmos de búsqueda y ordenamiento utilizados en programación, implementados en el lenguaje Python. A través de un análisis teórico y la puesta en práctica de estos algoritmos, buscamos comprender su funcionamiento, su eficiencia y su aplicabilidad en distintos contextos.
Además, abordamos la relación que existe entre la búsqueda y el ordenamiento de datos, resaltando su relevancia en el desarrollo de software y su influencia en el rendimiento de aplicaciones.

2. Marco Teórico

2.1 ¿Qué es un algoritmo?

Un algoritmo es un conjunto de instrucciones definidas, finitas y ordenadas que permiten resolver un problema o realizar una tarea específica. Son la base del pensamiento computacional y del diseño de programas. En programación, los algoritmos determinan cómo se procesan los datos y cómo se resuelven los distintos problemas.
Un algoritmo es un procedimiento computacional bien definido que toma algún valor como entrada y produce un valor de salida.

2.2 ¿Qué es la búsqueda?

La búsqueda es una operación que permite localizar un elemento dentro de una colección de datos. En programación, este proceso se aplica sobre estructuras como listas, arreglos o bases de datos, y puede realizarse mediante distintos métodos, cada uno con características, ventajas y desventajas particulares.

2.3 Tipos de algoritmos de búsqueda

Búsqueda lineal (secuencial)

Es el método más simple. Consiste en recorrer uno por uno los elementos de la lista hasta encontrar el valor deseado. No requiere que los datos estén ordenados. Su complejidad es O(n), siendo n el tamaño de la lista.

Ventajas: fácil de implementar.

Desventajas: ineficiente en listas grandes.

Búsqueda binaria

Funciona sobre listas ordenadas. Compara el valor buscado con el elemento central, y reduce la búsqueda a la mitad correspondiente (menor o mayor). Su complejidad es O(log n).

Ventajas: mucho más rápida que la lineal en listas grandes.

Desventajas: requiere ordenamiento previo.

Búsqueda por interpolación

Mejora la búsqueda binaria en listas numéricas distribuidas uniformemente, estimando la posición del valor buscado mediante una fórmula. Su rendimiento puede llegar a O(log log n).

Búsqueda con hash

Utiliza una función hash para calcular la posición del dato directamente. Es muy eficiente (casi O(1)), pero requiere estructuras especiales (tablas hash).

2.4 ¿Qué es el ordenamiento?

El ordenamiento es el proceso mediante el cual se reorganizan los elementos de una lista según un criterio (ascendente o descendente). Es fundamental para facilitar búsquedas rápidas y mejorar la eficiencia de muchos algoritmos.

2.5 Tipos de algoritmos de ordenamiento

Bubble Sort (burbuja)
Compara elementos adyacentes e intercambia si están en el orden incorrecto. Repite el proceso hasta que la lista esté ordenada.

- Complejidad: O(n^2).
- Muy simple, pero ineficiente para listas grandes.
- Ejemplo: ordenar los precios de una lista de productos antes de mostrarla en una tienda online.

Selection Sort (selección)
Encuentra el menor (o mayor) elemento en la lista no ordenada y lo intercambia con el primero. Repite para el resto.

- Complejidad: O(n^2).
- Fácil de implementar, pero lento en grandes volúmenes.

Insertion Sort (inserción)
Inserta cada nuevo elemento en su posición correcta dentro de una sublista ordenada.

- Complejidad: O(n^2) en el peor caso, pero muy eficiente en listas casi ordenadas.
- Ideal para listas pequeñas o parcialmente ordenadas

Merge Sort (intercalación)
Divide la lista en mitades, ordena cada una recursivamente y luego las combina.

- Complejidad: O(n log n), estable y eficiente.
- Muy eficiente y estable, útil para grandes volúmenes de datos.
- Desventajas: requiere memoria adicional para combinar las listas.

Quick Sort (rápido)
Selecciona un pivote, divide la lista en menores y mayores al pivote, y aplica recursión.

- Complejidad: O(n log n) en promedio.
- Ventajas: muy rápido en la práctica.
- Desventajas: rendimiento pobre en listas ya ordenadas si no se elige bien el pivote.

Comparación de algoritmos:

- Los algoritmos O(n^2) (burbuja, selección, inserción) son adecuados para listas pequeñas.

- Los algoritmos O(n log n) (merge, quick) son preferibles para listas grandes.
- La elección depende del tamaño, la naturaleza de los datos, y si se requiere estabilidad o bajo consumo de memoria.

2.6 Relación entre búsqueda y ordenamiento

En muchos casos, para aplicar una búsqueda eficiente (como la binaria), es necesario que los datos estén previamente ordenados. Así, el ordenamiento no solo mejora la legibilidad o presentación de los datos, sino que habilita técnicas de búsqueda más rápidas.
Además, algoritmos como las búsquedas con hash pueden beneficiarse de estructuras ordenadas para detectar colisiones o aplicar estrategias híbridas.

2.7 Importancia en la programación actual

Tanto la búsqueda como el ordenamiento son pilares en el desarrollo de software, desde tareas simples hasta aplicaciones complejas como motores de búsqueda, sistemas de recomendación, bases de datos o inteligencia artificial. Elegir el algoritmo adecuado puede representar una diferencia significativa en la eficiencia y escalabilidad de una solución.
Comprender su funcionamiento es clave para optimizar recursos, reducir tiempos de espera y brindar mejores experiencias a los usuarios. Por ello, son temas centrales en la formación de programadores y científicos de datos.

3. Algoritmo (Caso práctico)

Para ejemplificar la aplicación práctica de los algoritmos de búsqueda y ordenamiento, desarrollamos un programa en Python que permite:

- Generar listas de números aleatorios del tamaño que el usuario desee (Hasta 10.000).

- Ordenar estas listas utilizando tres algoritmos clásicos: Bubble Sort, Insertion Sort y Selection Sort.

- Buscar un elemento dentro de las listas utilizando dos métodos: búsqueda lineal y búsqueda binaria.

- Medir y comparar los tiempos de ejecución de cada algoritmo sobre listas de diferentes tamaños, permitiendo evaluar su rendimiento y eficiencia.

Este experimento nos permitió observar de forma concreta cómo se comportan los algoritmos al aplicarse sobre conjuntos de datos aleatorios, simulando situaciones reales. Además, la comparación entre los métodos de búsqueda y ordenamiento ofreció una perspectiva clara sobre las ventajas y limitaciones de cada técnica.

4. Metodología Empleada

La elaboración de este trabajo práctico integrador se desarrolló en las siguientes etapas:

- **Recolección de información teórica:** Se consultaron fuentes confiables, incluyendo el material provisto por la cursada, bibliografía recomendada y documentación externa, con el objetivo de comprender a fondo los algoritmos de búsqueda, ordenamiento y análisis de eficiencia.

- **Diseño e implementación de algoritmos en Python:** A partir de los conceptos teóricos, se seleccionaron e implementaron diversos algoritmos clásicos, priorizando una estructura modular y comprensible. El desarrollo se realizó de forma colaborativa, permitiendo la revisión y mejora constante del código.

- **Pruebas funcionales con diferentes conjuntos de datos:** Se aplicaron los algoritmos sobre distintos tamaños y tipos de listas para analizar su comportamiento, eficiencia y tiempos de ejecución. Esto permitió obtener una base objetiva para la comparación entre métodos.

- **Registro y análisis de resultados:** Se documentaron los resultados obtenidos en cada prueba y se realizó un análisis comparativo considerando eficiencia, claridad del algoritmo y adecuación al problema planteado.

- **Elaboración del informe y producción del video de presentación:** Se organizó la información en un informe estructurado que incluye marco teórico, desarrollo práctico y conclusiones. Asimismo, se produjo un video explicativo en el que se

muestra el funcionamiento del programa y se reflexiona sobre el trabajo realizado en equipo.

5. Resultados Obtenidos

A partir del desarrollo e implementación del programa en Python, se obtuvieron los siguientes resultados:

- **Funcionamiento correcto del menú interactivo:**
  Se logró una interfaz en consola clara y funcional, que permite al usuario definir el tamaño de la lista a procesar y repetir pruebas si lo desea.

- **Ordenamiento exitoso con tres algoritmos clásicos:**
  Se implementaron correctamente los métodos de ordenamiento **Bubble Sort**, **Insertion Sort** y **Selection Sort**. Todos ellos ordenaron las listas de números generadas aleatoriamente sin errores. Además, se midió el tiempo de ejecución de cada uno, lo que permitió observar diferencias de rendimiento según el tamaño de la lista ingresada.

- **Comparación de eficiencia en búsquedas:**
  El programa aplicó **búsqueda binaria** sobre listas ordenadas y **búsqueda lineal** sobre las listas originales, registrando el tiempo que tardó cada una. En todos los casos, la búsqueda binaria fue más rápida, especialmente en listas grandes, validando en la práctica su **eficiencia logarítmica (O(log n))** frente a la **búsqueda lineal (O(n))**.

- **Comprensión aplicada de la teoría algorítmica:**
  Se evidenció cómo la eficiencia teórica de los algoritmos se refleja en su rendimiento práctico. Por ejemplo, **Bubble Sort** y **Selection Sort** mostraron tiempos significativamente mayores que Insertion Sort en algunos casos, lo cual es coherente con su estructura interna de comparación e intercambio.

- **Manejo de errores y validación de entradas:**
  El menú incluye manejo de excepciones para validar que el usuario ingrese un número entero positivo. Esta mejora refuerza la robustez del código y evita errores durante la ejecución.

- **Facilidad para pruebas sucesivas:**
  Al permitir repetir las pruebas con distintas cantidades de datos desde la misma ejecución del programa, se facilitó el análisis experimental del comportamiento algorítmico, promoviendo la exploración por parte del usuario.

6. Conclusión

El estudio y aplicación de algoritmos de búsqueda y ordenamiento constituye una base clave en la formación de todo programador. Cómo se desarrolló en el marco teórico, ordenar los datos antes de aplicar ciertos métodos de búsqueda, como la binaria, es una condición necesaria para lograr resultados eficientes y confiables. Esta relación directa entre ordenamiento previo y velocidad de búsqueda se comprobó en los resultados obtenidos, reafirmando la teoría con evidencia práctica.

La posibilidad de comparar distintos algoritmos frente a conjuntos de datos aleatorios de diferentes tamaños permitió visualizar cómo la elección de una estrategia algorítmica adecuada no solo soluciona un problema, sino que también optimiza el rendimiento general del sistema. En ese sentido, la eficiencia no es un detalle menor, sino una variable crítica en el diseño de soluciones informáticas.

En resumen, reforzamos la importancia de entender no solo cómo funcionan los algoritmos, sino cuándo conviene aplicar cada uno, teniendo en cuenta el tipo de datos, el contexto de uso y los recursos disponibles. El análisis crítico de los resultados —como los tiempos registrados en cada prueba— nos permitió validar los conceptos estudiados y adquirir herramientas concretas para el desarrollo de software más eficiente.

7. Bibliografía

- Python Software Foundation. (2024). Python 3.12 Documentation. https://docs.python.org/3/

- Apuntes y materiales de la materia Programación I, Tecnicatura Universitaria en Programación a Distancia, UTN.

8. Anexos

Video explicativo:

Repositorio GitHub:

Pruebas de verificación de funcionamiento:

```
PS C:\Users\lalvarez\Desktop\pdf_paginas> python tpintegrador.py
 ____________________________________________________________________________________
|                                                                                    |
|Bienvenido! Analizaremos los tiempos de procesamiento de distintas tipos de busquedas|
|____________________________________________________________________________________|
Cuántos números querés procesar? 250

_____________________________________________
Pruebas realizadas con 250 elementos
Bubble Sort: 0.002454 segundos
Insertion Sort: 0.000954 segundos
Selection Sort: 0.000649 segundos
Búsqueda Binaria: 0.000005 segundos
Búsqueda Lineal: 0.000007 segundos
Querés procesar otra lista? (s/n): s
Cuántos números querés procesar? 10000

_____________________________________________
Pruebas realizadas con 10000 elementos
Bubble Sort: 3.701716 segundos
Insertion Sort: 1.685955 segundos
Selection Sort: 1.710454 segundos
Búsqueda Binaria: 0.000008 segundos
Búsqueda Lineal: 0.000042 segundos
Querés procesar otra lista? (s/n): n
Programa finalizado.
```

En la imagen se puede observar el funcionamiento del menú y las pruebas realizadas de distintos tamaños y la diferencias de tiempo de procesamiento de cada ejecución.
También se puso en práctica la opción de elegir procesar otra lista o finalizar la ejecución.

```
PS C:\Users\lalvarez\Desktop\pdf_paginas> python tpintegrador.py
 ____________________________________________________________________________________
|                                                                                    |
|Bienvenido! Analizaremos los tiempos de procesamiento de distintas tipos de busquedas|
|____________________________________________________________________________________|
Cuántos números querés procesar? asoidja
Entrada no válida: Se permiten solamente números enteros positivos.
```

Realizamos la prueba de ingreso de caracteres no válidos y la captura del error.