Ordenamiento y Búsqueda en Estructuras de Datos

Curso

Tecnicatura en Programación

Alumnos

Barroso, Nikolas - nhbouharriet@gmail.com

Huarcaya, Ivan. - Hf.rivan@gmail.com

Materia

programación I

Docentes

Prof. Ariel Enferrel – Tutor Maximiliano Sar Fernández

Fecha de entrega

09/06/2025

Contenido

1. Introducción

1.1. Contextualización del Problema

El procesamiento y manejo eficiente de datos es un pilar central en el desarrollo de aplicaciones informáticas. En contextos como el Big Data, la Inteligencia Artificial o los sistemas de información, la eficiencia algorítmica representa un factor decisivo para el éxito de los proyectos. En este sentido, los algoritmos de búsqueda y ordenamiento permiten optimizar operaciones fundamentales como la recuperación, clasificación y gestión de grandes volúmenes de información.

Su aplicación va desde la localización de registros en bases de datos, hasta el renderizado de listas en aplicaciones web, pasando por tareas de inteligencia computacional. La correcta selección de estos algoritmos influye no solo en la eficiencia del código, sino también en el consumo de recursos y la experiencia del usuario final.

Según un estudio publicado por McKinsey Global Institute (2021), las organizaciones que implementan algoritmos eficientes para el manejo de datos pueden mejorar su productividad hasta en un 40%.

1.2. Objetivos del Trabajo

Describir y analizar el funcionamiento de cinco algoritmos fundamentales (tres de ordenamiento y dos de búsqueda).

Implementar dichos algoritmos utilizando el lenguaje de programación Python.

Comparar su desempeño en diferentes contextos y estructuras de datos.

Proporcionar recomendaciones prácticas sobre su uso según el tipo de problema.

Concientizar sobre la importancia de las decisiones algorítmicas en el desarrollo de software eficiente.

2. Marco Teórico

2.1. Búsqueda en Programación

La búsqueda es una operación que permite encontrar un elemento determinado dentro de una estructura de datos. Esta acción puede parecer simple, pero su eficiencia puede variar enormemente según el algoritmo utilizado. Conceptos clave como "complejidad computacional", "eficiencia algorítmica" y "estructura de datos" están intrínsecamente ligados a esta problemática.

2.2. Aplicaciones de la Búsqueda

- Describir y analizar el funcionamiento de cinco algoritmos fundamentales (tres de ordenamiento y dos de búsqueda).
- Implementar dichos algoritmos utilizando el lenguaje de programación Python.
- Comparar su desempeño en diferentes contextos y estructuras de datos.
- Proporcionar recomendaciones prácticas sobre su uso según el tipo de problema.
- Concientizar sobre la importancia de las decisiones algorítmicas en el desarrollo de software eficiente.

2.3. Algoritmos de Búsqueda Comunes

- **Búsqueda Lineal (Linear Search)**

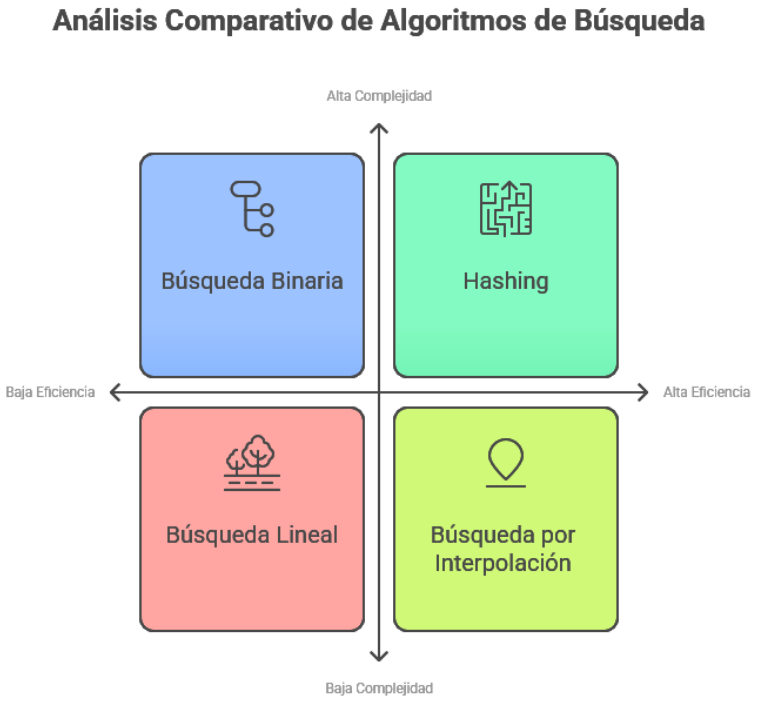
La búsqueda lineal es el algoritmo más básico, el cual recorre secuencialmente cada elemento hasta encontrar el objetivo.

- **Funcionamiento**: Recorre cada elemento secuencialmente hasta encontrar el objetivo.
- **Complejidad**: **O(n)** (lineal).
- **Peor caso:** *O(n)*
- **Mejor caso:** *O(1)* (si el elemento está en la primera posición)
- **Ventajas**: Simple de implementar, no requiere datos ordenados.
- **Desventajas**: Ineficiente para grandes conjuntos de datos.
- **Aplicaciones:** Búsqueda en listas pequeñas. Verificación de existencia en estructuras no indexadas.

- **Búsqueda Binaria (Binary Search)**

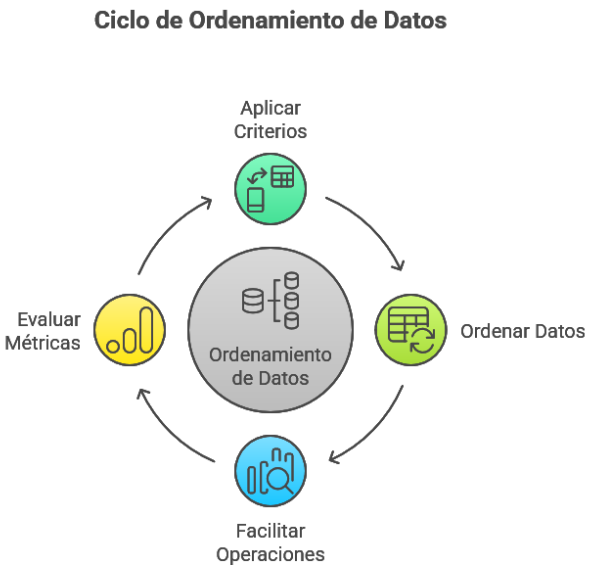
Este algoritmo aplica la estrategia **"divide y vencerás"**, requiriendo que los datos estén ordenados. Divide repetidamente el conjunto en mitades, descartando la mitad no relevante en cada iteración.

- **Funcionamiento**: Divide repetidamente el conjunto de datos a la mitad, aprovechando que está ordenado.
- **Complejidad**: **O(log n)** (logarítmica).
- **Ventajas**: Muy eficiente para datos ordenados.
- **Desventajas**: Requiere que los datos estén previamente ordenados.
- **Aplicaciones:** Búsqueda en bases de datos indexadas, Implementación en librerías estándar.



2.4. Ordenamiento en Programación

El ordenamiento organiza los datos siguiendo un criterio (numérico, alfabético, etc.), es una operación clave para optimizar búsquedas y mejorar el rendimiento en el procesamiento de datos.



2.5. Compensaciones de los algoritmos de ordenación

¿Cuán grande es la colección que se ordena? ¿Cuánta memoria hay disponible? ¿La colección necesita crecer?. Las respuestas a estas preguntas pueden determinar qué algoritmo funcionará mejor para cada situación.

Algunos algoritmos como la ordenación por combinación pueden necesitar mucho espacio o memoria para ejecutarse, mientras que la ordenación por inserción no siempre es la más rápida, pero no requiere muchos recursos para ejecutarse

Al seleccionar un algoritmo de ordenamiento, se deben considerar factores como:

- Estabilidad (si conserva el orden relativo de elementos iguales).
- Uso de memoria adicional.
- Tipo y distribución de los datos.
- Complejidad temporal en peor, mejor y promedio caso.

2.6. Clasificación de un algoritmo de ordenación

Los algoritmos de ordenación se pueden clasificar en función de los siguientes parámetros:

- La cantidad de intercambios o inversiones requeridas:
- El número de comparaciones:
- Ya sea que usen recursividad o no:
- Ya sean estables o inestables:
- La cantidad de espacio adicional requerido

2.7. Algoritmos de Ordenamiento Comunes

- Bubble Sort (Ordenamiento por Burbuja)

Permite que los valores más bajos o más altos aparezcan en la parte superior. Recorre la lista varias veces intercambiando elementos adyacentes desordenados.

- **Funcionamiento**: Compara elementos adyacentes y los intercambia si están en el orden incorrecto.
- **Complejidad**: **$O(n^2)$** (ineficiente para grandes conjuntos).
- **Ventajas**: Simple de entender e implementar.
- **Desventajas**: Lento para datos grandes.

- **Quick Sort (Ordenamiento Rápido)**

Emplea un **pivote** para particionar la lista en subconjuntos (elementos menores y mayores al pivote), aplicando recursión para ordenar cada partición.

  - **Funcionamiento**: Usa un **pivote** para dividir el conjunto en subgrupos (menores y mayores).
  - **Complejidad**: **O(n log n)** en promedio (muy eficiente).
  - **Caso promedio:** *O(n log n)*
  - **Peor caso:** $O(n^2)$ (si el pivote es mal seleccionado)
  - **Ventajas**: Uno de los más rápidos en la práctica.
  - **Desventajas**: **$O(n^2)$** en el peor caso (si el pivote es mal elegido).
  - **Aplicaciones:** Librerías estándar

- **Selection Sort (Ordenamiento por Selección)**

Itera a través del arreglo: Selecciona el elemento más pequeño actual y lo cambia de lugar.

  - **Funcionamiento**: Busca el mínimo en cada iteración y lo coloca en su posición correcta.
  - **Complejidad**: **$O(n^2)$**.
  - **Ventajas**: Más eficiente que Bubble Sort en algunos casos.
  - **Desventajas**: Poco eficiente para datos grandes.

- **Insertion Sort (Ordenamiento por Inserción)**

Construye una secuencia ordenada insertando cada elemento en su posición correcta dentro de la sublista ya ordenada.

  - **Funcionamiento**: Construye una lista ordenada insertando elementos uno por uno.
  - **Complejidad**: **$O(n^2)$**.
  - **Peor caso:** $O(n^2)$
  - **Mejor caso:** *O(n)* (cuando la lista ya está ordenada)
  - **Ventajas**: Eficiente para listas pequeñas o casi ordenadas.
  - **Desventajas**: Lento para conjuntos grandes.
  - **Aplicaciones:** Algoritmo base en optimizaciones como **Timsort**.

- **Merge Sort (Ordenamiento por Mezcla)**

Aplica **divide y vencerás** dividiendo la lista en mitades, ordenándolas recursivamente y luego fusionándolas.

- **Funcionamiento**: Divide el conjunto en mitades, las ordena y luego las fusiona.
- **Complejidad**: **O(n log n)** (siempre).
- **Ventajas**: Estable y eficiente.
- **Desventajas**: Requiere memoria adicional para la fusión.
- **Aplicaciones:** Ordenamiento externo (datos en disco).

2.8. Aplicaciones del Ordenamiento

- **Optimización de consultas en bases de datos** (índices).
- **Procesamiento de datos** (análisis estadístico).
- **Sistemas de recomendación** (ordenar productos por relevancia).
- **Renderizado de listas** en aplicaciones web/móviles.
- **Importancia de la Búsqueda y el Ordenamiento**

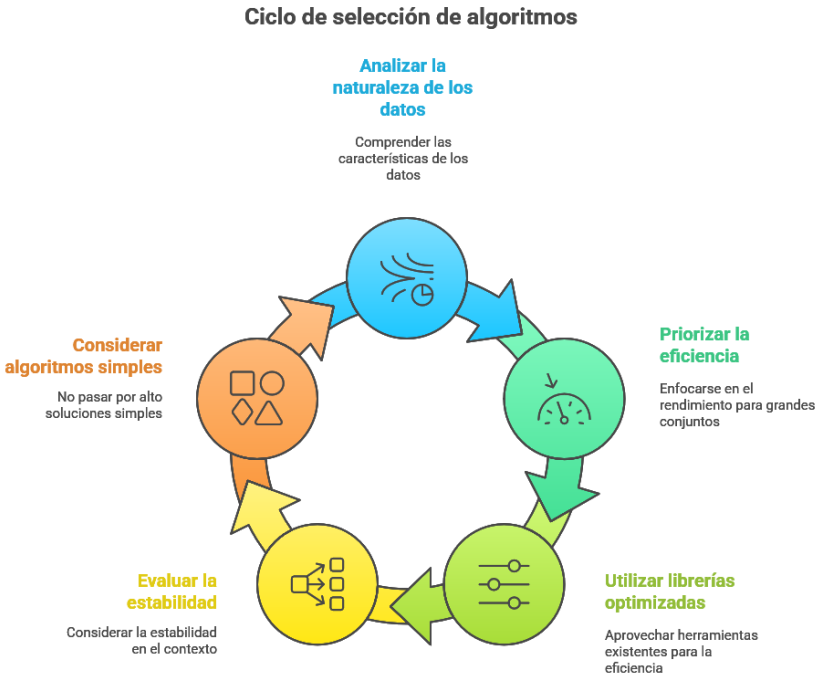
Estos algoritmos son esenciales porque:

- **Mejoran la eficiencia** de los programas.
- **Permiten búsquedas rápidas** (especialmente con datos ordenados).
- **Facilitan el análisis de datos** (visualización, estadísticas).
- **Son fundamentales en estructuras de datos** (árboles, grafos, tablas hash).
- **Optimizan el rendimiento** en aplicaciones críticas (bases de datos, motores de búsqueda).

Recomendaciones Prácticas

A la hora de seleccionar un algoritmo de ordenamiento o búsqueda, se recomienda:

- Utilizar QuickSort cuando se necesite alto rendimiento con listas grandes y se cuente con suficiente memoria.
- Elegir algoritmos según la naturaleza de los datos (ordenados, distribuidos, volumen).
- Priorizar la eficiencia cuando se trata de grandes conjuntos.
- Evaluar la necesidad de estabilidad según el contexto.
- Preferir Inserción si se trata de listas pequeñas o parcialmente ordenadas, donde su rendimiento es aceptable y su implementación es sencilla.
- Evitar Burbuja, excepto en situaciones donde la simplicidad extrema sea prioritaria.
- Para búsquedas, aplicar búsqueda binaria solo en listas ordenadas; de lo contrario, utilizar búsqueda lineal.
- No subestimar algoritmos simples como Insertion Sort en contextos específicos.
- En sistemas con restricciones de memoria, optar por algoritmos in-place.
- Considerar el tipo de datos y su distribución antes de elegir el algoritmo.



3. Caso practico

Caso 1

Algoritmos Clave Aplicados - Búsqueda Binaria:

¿Para qué?

Localizar animales rápidamente usando su ID de chip o número de registro.

¿Cómo funciona?

Requiere que los datos estén ordenados. Divide la lista a la mitad repetidamente hasta encontrar el animal deseado, siendo muy eficiente para grandes volúmenes de datos.

Ordenamiento Rápido (Quick Sort):

¿Para qué?

Organizar las listas de animales por criterios importantes como fecha de ingreso, edad o estado de salud.

¿Cómo funciona?

Es un algoritmo de ordenamiento muy eficiente que organiza los datos de forma rápida, facilitando búsquedas y reportes posteriores.

Búsqueda por Interpolación:

¿Para qué?

Buscar animales por características numéricas como peso o edad, donde los datos suelen tener una distribución uniforme.

¿Cómo funciona?

Es una mejora de la búsqueda binaria que estima la posición del elemento, acelerando la búsqueda en datos bien distribuidos.

Estructura de Datos Base

Para una gestión efectiva, la información de cada animal se almacenaría en una estructura de datos clara, como una clase Animal en Python, que contendría atributos como ID, nombre, especie, edad, peso, estado de salud y fecha de ingreso.

```python
class Animal:
    def __init__(self, id_chip, nombre, especie, raza, edad, peso,
                 fecha_ingreso, estado_salud, adoptado=False):
        self.id_chip = id_chip
        self.nombre = nombre
        self.especie = especie  # Perro, Gato, Conejo, etc.
        self.raza = raza
        self.edad = edad  # en meses
        self.peso = peso  # en kg
        self.fecha_ingreso = fecha_ingreso
        self.estado_salud = estado_salud
# Excelente, Bueno, Regular, Crítico
        self.adoptado = adoptado
        self.historial_medico = []
```

Caso 2

Algoritmos aplicados:

Búsqueda de interpolación: Para buscar adoptantes por rango de edad o ingresos

Ordenamiento rápido: Para generar listas de compatibilidad adopción-animal

Búsqueda lineal con filtros múltiples: Para matching adopción basado en múltiples criterios

Estructura de datos:

class Adoptante:

def _init_(self, id_adoptante, nombre, edad, ingresos, tipo_vivienda,

experiencia_animales, preferencias):

self.id_adoptante = id_adoptante

self.nombre = nombre

self.edad = edad

self.ingresos = ingresos

self.tipo_vivienda = tipo_vivienda # Casa, Apartamento, Finca

```
        self.experiencia_animales = experiencia_animales
        self.preferencias = preferencias  # Tamaño, edad, especie preferida
        self.historial_adopciones = []

class Adopcion:
    def _init_(self, id_adopcion, id_animal, id_adoptante, fecha_adopcion):
        self.id_adopcion = id_adopcion
        self.id_animal = id_animal
        self.id_adoptante = id_adoptante
        self.fecha_adopcion = fecha_adopcion
        self.seguimientos = []  # Lista de seguimientos post-adopción
```

Casos de uso reales:

Matching de compatibilidad: Algoritmo de puntuación ordenado de mayor a menor compatibilidad

Lista de espera por tipo de animal: Cola ordenada por fecha de solicitud

Seguimiento post-adopción: Búsqueda por fechas de seguimiento requeridas

Estadísticas de éxito: Análisis de adopciones exitosas vs devueltas

Métricas de matching:

Criterios de compatibilidad (puntaje 1-10):

- Tamaño del animal vs espacio disponible
- Experiencia del adoptante vs necesidades del animal
- Tiempo disponible vs requerimientos de ejercicio
- Presupuesto vs costos estimados de mantenimiento

```python
class Adoptante:
    def __init__(self, id_adoptante, nombre, edad, ingresos, tipo_vivienda,
                 experiencia_animales, preferencias):
        self.id_adoptante = id_adoptante
        self.nombre = nombre
        self.edad = edad
        self.ingresos = ingresos
        self.tipo_vivienda = tipo_vivienda  # Ej: Casa, Apartamento, Finca
        self.experiencia_animales = experiencia_animales
# Ej: "principiante", "intermedia", "experto"
        self.preferencias = preferencias
# Ej: {"tamaño": "pequeño", "edad": "joven", "especie": "perro"}
        self.historial_adopciones = []

class Adopcion:
    def __init__(self, id_adopcion, id_animal, id_adoptante, fecha_adopcion):
        self.id_adopcion = id_adopcion
        self.id_animal = id_animal
        self.id_adoptante = id_adoptante
        self.fecha_adopcion = fecha_adopcion
        self.seguimientos = []  # Lista de fechas de seguimiento post-adopción
```

4. Conclusión

La programación eficiente requiere un conocimiento profundo de los algoritmos de búsqueda y ordenamiento. A través de este trabajo, se evidenció que la elección adecuada de dichos algoritmos depende del tipo de datos, el contexto de aplicación y los recursos disponibles.

La eleccion debe basarse en:

- Tamaño del conjunto de datos.
- Requisitos de estabilidad y memoria.
- Frecuencia de actualización.
- Futuras líneas de investigación:
- Análisis de algoritmos híbridos (e.g., Timsort).
- Optimización de búsqueda en estructuras no lineales (árboles, grafos).

Comprender conceptos como complejidad algorítmica, recursividad, estabilidad y eficiencia permite desarrollar soluciones más robustas y eficaces.

Este estudio refuerza la importancia de comprender estos algoritmos para el desarrollo de software escalable y de alto rendimiento.

La búsqueda permite encontrar información rápidamente, con algoritmos como **binaria** (para datos ordenados) o **hash** (para acceso instantáneo).

El ordenamiento organiza los datos para hacer búsquedas y procesamientos más eficientes, con algoritmos como **Quick Sort** y **Merge Sort**.

La elección del algoritmo depende del contexto: tamaño de los datos, frecuencia de actualización y requisitos de rendimiento.

La **búsqueda binaria** es óptima para datos ordenados, mientras que el **hashing** ofrece el mejor rendimiento en estructuras bien diseñadas.

MergeSort garantiza *O(n log n)* en todos los casos, siendo ideal para aplicaciones que requieren estabilidad.

QuickSort es preferido en implementaciones prácticas debido a su eficiencia promedio.