Algoritmos de Búsqueda y Ordenamiento

El manejo eficiente de datos es crucial en el desarrollo de aplicaciones. Los algoritmos de búsqueda y ordenamiento optimizan operaciones fundamentales. Su correcta selección influye en la eficiencia, consumo de recursos y experiencia del usuario.

Objetivos del Trabajo

1 **Describir y Analizar**

Funcionamiento de cinco algoritmos fundamentales.

2 **Implementar**

Algoritmos utilizando el lenguaje Python.

3 **Comparar Desempeño**

En diferentes contextos y estructuras de datos.

4 **Proporcionar Recomendaciones**

Prácticas sobre su uso.

5 **Concientizar**

Importancia de decisiones algorítmicas eficientes.

Estructura de los Objetivos del Trabajo

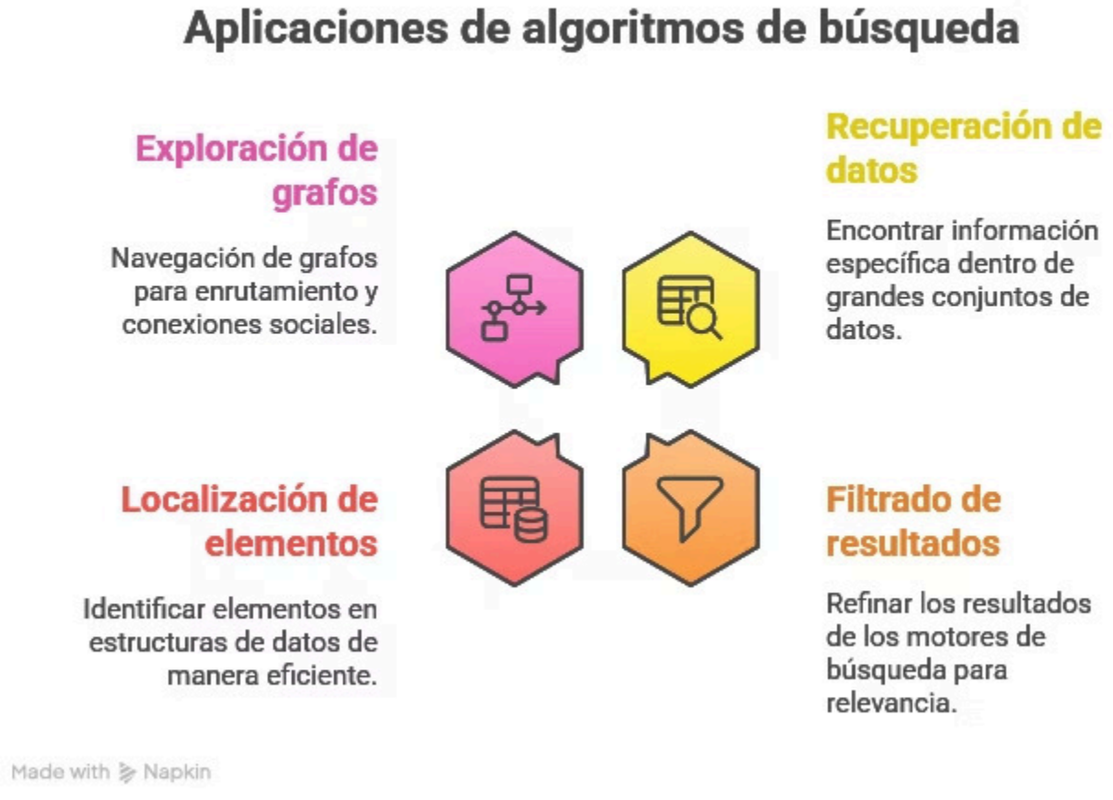
Marco Teórico: Búsqueda en Programación

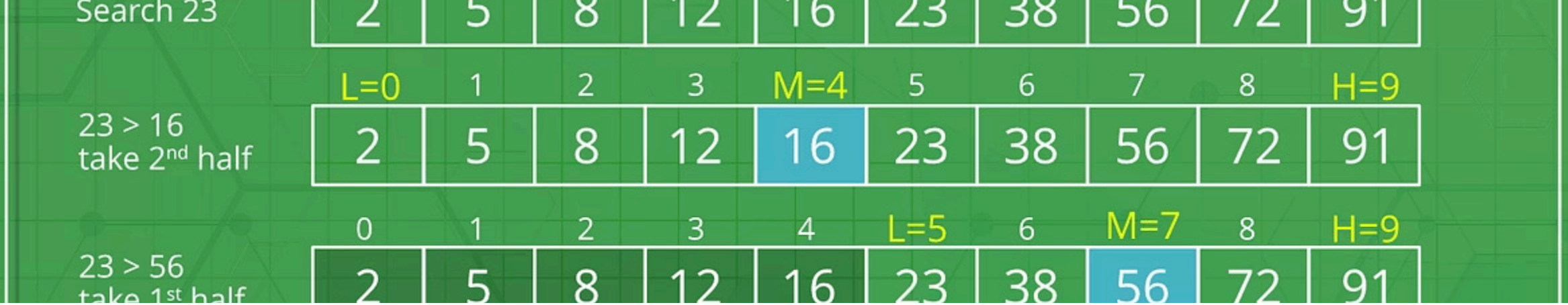
Concepto Clave

La búsqueda encuentra un elemento en una estructura de datos. Su eficiencia varía según el algoritmo. Conceptos como "complejidad computacional" son clave.

Aplicaciones Comunes

- Optimización de consultas en bases de datos.
- Procesamiento de datos para análisis estadístico.
- Sistemas de recomendación.
- Renderizado de listas en aplicaciones.





Algoritmos de Búsqueda Comunes

Búsqueda Lineal

Recorre secuencialmente cada elemento. Es simple de implementar y no requiere datos ordenados. Ineficiente para grandes conjuntos de datos.

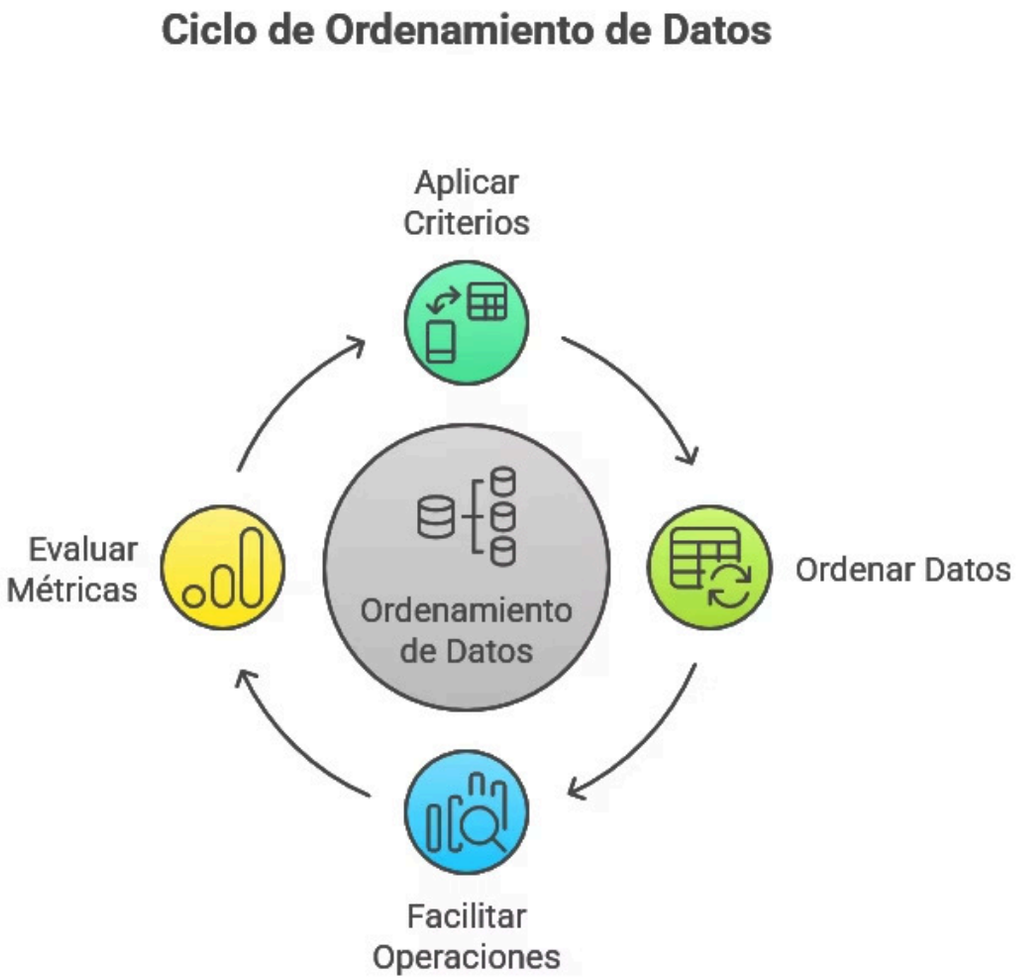
Complejidad: O(n).

Búsqueda Binaria

Aplica "divide y vencerás" en datos ordenados. Muy eficiente para grandes volúmenes. Requiere que los datos estén previamente ordenados.

Complejidad: O(log n).

Ordenamiento en Programación



El ordenamiento organiza datos para optimizar búsquedas y mejorar el rendimiento. La elección del algoritmo depende del tamaño de la colección y la memoria disponible.

Uso de Memoria

Algunos algoritmos necesitan más espacio para ejecutarse.

Eficiencia

La velocidad varía según el algoritmo y el tamaño de los datos.

Estabilidad

Si conserva el orden relativo de elementos iguales.

Algoritmos de Ordenamiento Comunes

1

Bubble Sort

Compara e intercambia adyacentes. Simple, pero $O(n^2)$ ineficiente.

2

Quick Sort

Usa pivote para dividir. O(n log n) en promedio, muy rápido.

3

Selection Sort

Busca el mínimo y lo coloca. $O(n^2)$, poco eficiente.

4

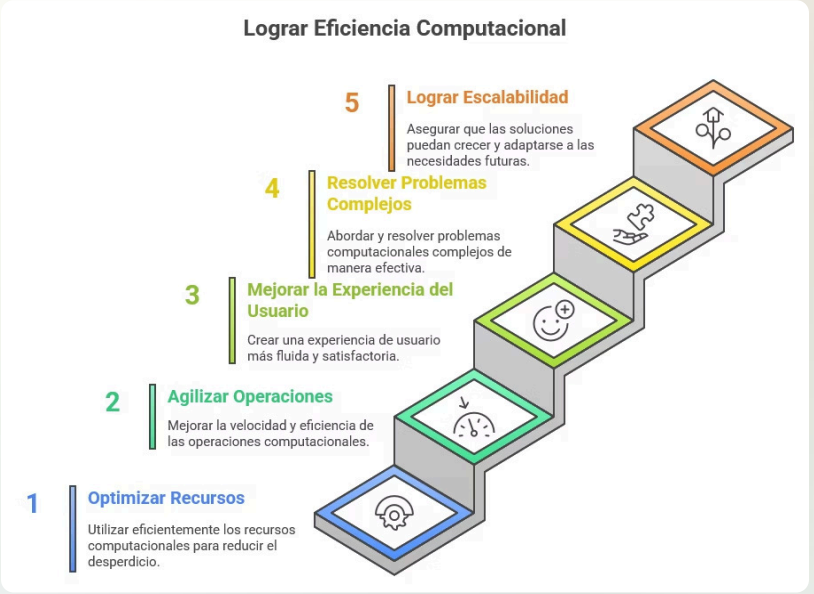
Insertion Sort

Inserta elementos uno a uno. $O(n^2)$ pero eficiente para listas pequeñas.

5

Merge Sort

Divide, ordena y fusiona. Siempre O(n log n), estable.



Aplicaciones del Ordenamiento

Bases de Datos

Optimización de consultas mediante índices.

Análisis de Datos

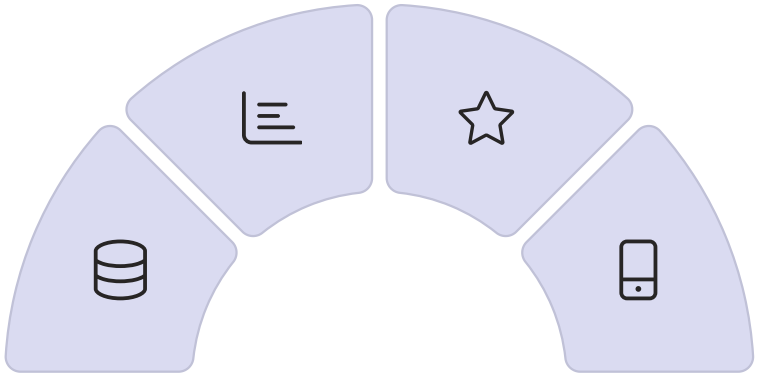
Facilita la visualización y estadísticas.

Sistemas de Recomendación

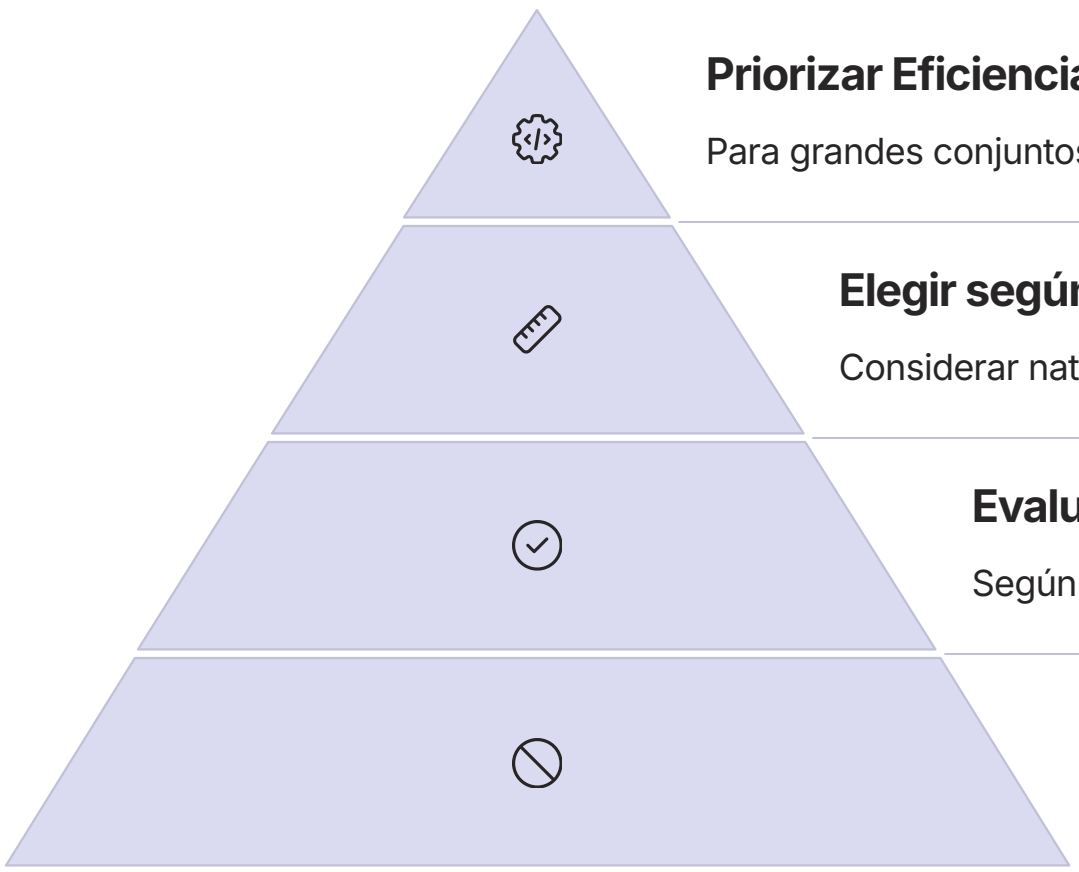
Ordena productos por relevancia.

Aplicaciones Web/Móviles

Renderizado eficiente de listas.



Recomendaciones Prácticas



Priorizar Eficiencia

Para grandes conjuntos de datos.

Elegir según Datos

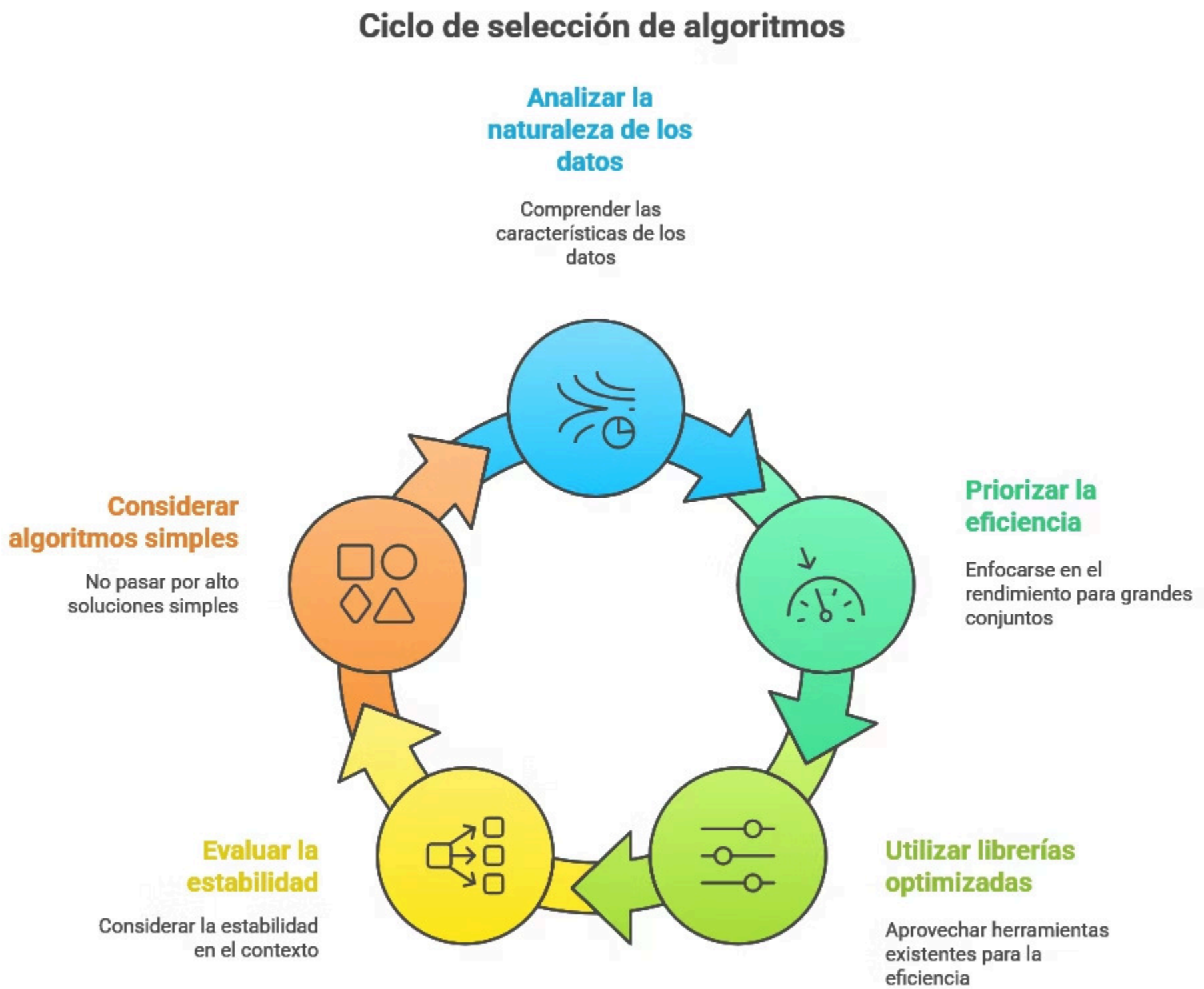
Considerar naturaleza, volumen y distribución.

Evaluar Estabilidad

Según el contexto de la aplicación.

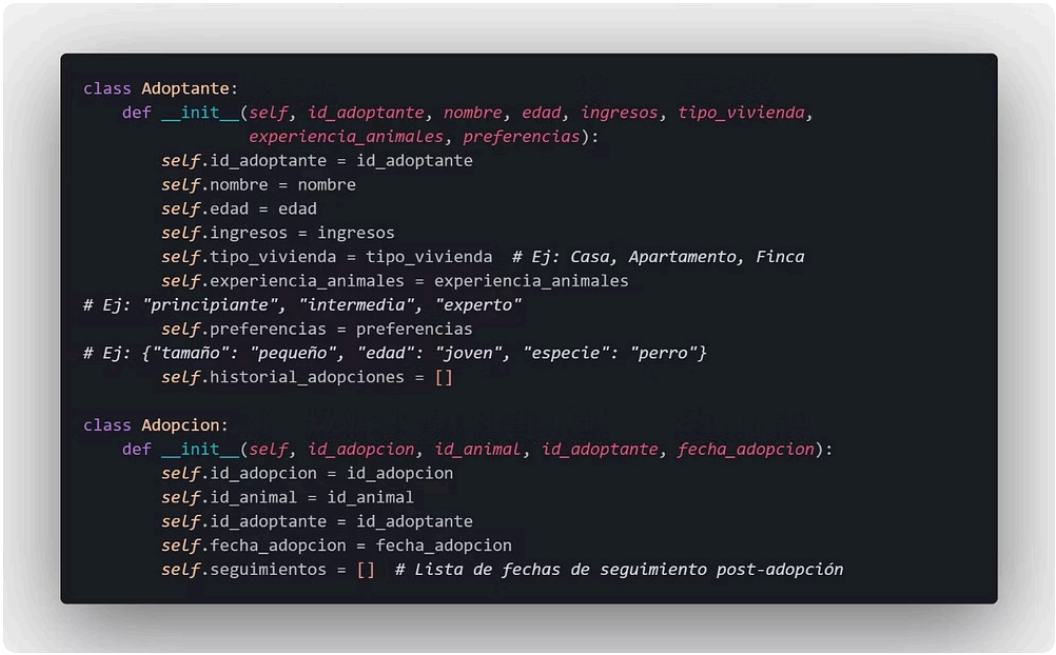
Evitar Bubble Sort

Excepto por extrema simplicidad.



Casos Prácticos

```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 = []
```



```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
```

Caso 1:

Sistema de Gestión para Refugio de Animales

Optimizar la administración y acceso a la información de todos los animales del refugio, asegurando una gestión eficiente y rápida.

Caso 2:

Sistema de Adopciones y Seguimiento

Administrar de forma efectiva el proceso de adopción y el seguimiento posterior de los animales, asegurando el mejor hogar para cada uno

Conclusión

La elección del algoritmo depende del tipo de datos, contexto y recursos. Comprender la complejidad algorítmica es fundamental para software robusto.

O(n log n)

MergeSort

Garantiza eficiencia en todos los casos.

O(n log n)

QuickSort

Preferido por su eficiencia promedio.

O(log n)

Búsqueda Binaria

Óptima para datos ordenados.