

Trabajos Integrador

Programación I

Implementación de Algoritmos de Búsqueda y Ordenamiento

Alumnas:

Verónica Falconí - verónica.falconi@tupad.utn.edu.ar

Yohanna Díaz Monroy - yohanna.diaz@tupad.utn.edu.ar

• Materia: Programación I

• Profesor: Ariel Enferrel

• Fecha de Entrega: (09-05-2025)

1. Introducción	3
Objetivos y Aplicabilidad Práctica	3
2. Marco Teórico	4
Definiciones	4
Ordenamiento en programación	4
Tipos de Algoritmos de Ordenamiento	5
Algoritmo de la burbuja (Bubble Sort):	5
Algoritmo de Selección (Selection Sort):	6
Algoritmo de Inserción (Insertion Sort):	6
Ordenación por mezcla (Merge Sort):	7
Ordenamiento rápido (Quick Sort):	7
Algoritmo de intercambio	7
Que es la búsqueda en programación	8
Clasificaciones	8
Tipos de Búsqueda:	8
Búsqueda lineal	8
Búsqueda binaria	10
Búsqueda de interpolación	12
3. Caso Práctico	19
Problema a resolver	19
Código fuente comentado	20
Explicación de decisiones de diseño	32
Validación del funcionamiento	33
4. Metodología Utilizada	35
Revisión bibliográfica y marco teórico	35
Pruebas experimentales y análisis de rendimiento	35
Etapas de diseño y prueba del código	36
5. Resultados Obtenidos	36
Casos de prueba realizados	36
Para el caso del ordenamiento	37
Para el caso de la búsqueda	37
6. Conclusiones	38
7. Bibliografía	39
8. Anexos	40

1. Introducción

La búsqueda y el ordenamiento representan dos conceptos esenciales y complementarios. La capacidad de localizar un elemento específico dentro de un vasto conjunto de información resulta tan crucial como la habilidad de organizar dichos datos bajo un criterio lógico y predefinido. Esta interrelación no solo agiliza la recuperación de elementos individuales, sino que también revela patrones y relaciones intrínsecas que, de otro modo, permanecerían ocultos, transformando vastas colecciones de datos en conjuntos de información valiosa y verdaderamente accesible.

La relevancia de estas operaciones se magnifican exponencialmente en el campo de la programación. Aquí, los algoritmos de búsqueda y ordenamiento se erigen como pilares indispensables. Su implementación precisa no solo optimiza la manera en que se encuentra información específica, sino que también minimiza el consumo de recursos computacionales esenciales, como la memoria y la capacidad del procesador. Esta eficiencia inherente resulta crítica en entornos donde los recursos son limitados o en sistemas que deben gestionar un volumen masivo de solicitudes, garantizando así un rendimiento óptimo y una experiencia de usuario impecable. Es por esta razón que las organizaciones recurren a ellos, entendiendo que el buen manejo de la información es el desencadenante principal para la eficacia de cualquier programa.

Objetivos y Aplicabilidad Práctica

El presente trabajo tiene como objetivo principal la comprensión de los fundamentos de los algoritmos de búsqueda y ordenamiento, familiarizándose con sus principios básicos a través de una investigación teórica.

Más allá de la teoría, este estudio también busca demostrar la aplicabilidad real de lo aprendido. Se implementarán los algoritmos en un caso práctico, integrándolos en un pequeño proyecto que simule una situación real. Esta aproximación permitirá no solo consolidar el conocimiento teórico, sino también desarrollar habilidades de codificación prácticas y esenciales.

2. Marco Teórico

Definiciones

Ordenamiento en programación:

Ordenar significa reagrupar o reorganizar un conjunto de datos u objetos en una secuencia específica. Los algoritmos de ordenamiento son un conjunto de instrucciones que toman una lista o un arreglo de elementos como entrada y los reorganizan en una secuencia específica (generalmente numérica o alfabética) basándose en una clave de ordenamiento. El objetivo es que los elementos queden dispuestos de menor a mayor (orden ascendente) o de mayor a menor (orden descendente).

Los procesos de ordenación y búsqueda son frecuentes en nuestra vida diaria, en un mundo desarrollado y acelerado en el que la información es de vital importancia. Y la operación de búsqueda de información generalmente se hace sobre elementos ordenados. (b)

El ordenamiento o clasificación de datos (sort, en inglés) consiste en disponer un conjunto, estructura de datos en algún determinado orden con respecto a uno de los campos de elementos del conjunto.

Los datos de un vector, una lista o un archivo están ordenados cuando cada elemento ocupa el lugar que le corresponde según su valor, un dato clave o un criterio. Si cada elemento es diferente del primero es mayor que los anteriores, se dice que está ordenado ascendentemente, mientras que si cada elemento diferente del primero es menor que los anteriores, está ordenado de forma descendente. En un pequeño conjunto de datos el orden en que estos se presente puede ser irrelevante, pero cuando la cantidad aumenta el orden toma importancia. (e)

Una colección de datos (estructura) puede ser almacenada en un archivo, un array (vector o matriz), una lista enlazada o un árbol. Cuando los datos están almacenados en un array, una lista enlazada o un árbol, se denomina ordenación interna. Si los datos están almacenados en un archivo, el proceso de ordenación se llama ordenación externa.

El ordenamiento es una operación fundamental en la informática y la ciencia de datos por varias razones:

Facilita la búsqueda: Una vez que los datos están ordenados, la búsqueda de elementos se vuelve mucho más eficiente.

Preparación para otras operaciones: Muchas tareas y algoritmos (como la fusión de conjuntos de datos o la detección de duplicados) requieren que los datos estén previamente ordenados para funcionar correctamente o de manera eficiente.

Análisis y visualización de datos: Los datos ordenados son más fáciles de analizar, comparar y visualizar, lo que permite extraer información y patrones de forma más efectiva.

Optimización de recursos: Un algoritmo de ordenamiento eficiente puede reducir significativamente el tiempo de ejecución y el uso de memoria en aplicaciones que manejan grandes volúmenes de datos.

Tipos de Algoritmos de Ordenamiento :

Los algoritmos de ordenamiento se pueden clasificar de diversas maneras, pero una de las más comunes es por su complejidad temporal (cómo el tiempo de ejecución crece con el tamaño de los datos, denotado con la notación Big O, $O(n)$) y por la técnica que emplean:

Algoritmo de la burbuja (Bubble Sort):

Por la forma como se hacen los recorridos y las comparaciones éste es uno de los algoritmos de ordenamiento más fáciles de comprender y de programar, pero como explica Joyanes (2000: 252) no es recomendable su utilización en el desarrollo de software, por ser el de menor eficiencia.

La técnica de ordenamiento conocida como burbuja o burbujeo consiste en comparar el primer elemento con el segundo y si no cumplen el criterio de orden que se está aplicando se intercambian, acto seguido se pasa a comparar el segundo elemento con el tercero y si no están ordenados se intercambian también, luego se

compara el tercero con el cuarto y después el cuarto con el quinto y así sucesivamente hasta comparar los elementos

Complejidad: $O(n^2)$ en el peor y caso promedio. Sencillo de entender, pero muy ineficiente para grandes conjuntos de datos.

Algoritmo de Selección (Selection Sort):

Este método es similar al anterior en cuanto a los recorridos del vector y las comparaciones, pero con un número menor de intercambios. En el método de intercambio cada vez que se comparan dos posiciones y éstas no están ordenadas se hace el intercambio de los datos, de manera que en un mismo recorrido puede haber varios intercambios antes de que el número ocupe el lugar que le corresponde en el arreglo ordenado. En el método de selección, en cada recorrido se identifica el elemento que pertenece a una posición y se hace un solo intercambio.

Concepto: Divide la lista en una parte ordenada y otra desordenada. En cada iteración, encuentra el elemento mínimo (o máximo) de la parte desordenada y lo coloca al final de la parte ordenada.

Complejidad: $O(n^2)$ en todos los casos. Realiza menos intercambios que el Bubble Sort, pero las comparaciones siguen siendo muchas

Algoritmo de Inserción (Insertion Sort):

Este método consiste en ordenar los elementos del arreglo progresivamente, comenzando por los primeros y avanzando hasta ordenarlos todos. Dado el elemento de una posición i (i comienza en 2 y va hasta el fin del arreglo) se guarda en una variable auxiliar y ésta se compara con el elemento que está a la izquierda; si no están ordenados, el elemento de la izquierda se desplaza hacia la derecha. Se vuelve a comparar la variable auxiliar con el elemento que sigue por la izquierda. Todo elemento que no esté ordenado se desplaza una posición a la derecha, hasta que se encuentre uno que sí esté en orden o se llegue al inicio del vector. Al terminar este recorrido se deposita, en el espacio que quedó libre, el elemento contenido en la variable auxiliar. (e)

Complejidad: $O(n^2)$ en el peor y caso promedio, $O(n)$ en el mejor caso (cuando la lista ya está casi ordenada). Es eficiente para conjuntos de datos pequeños o casi ordenados.

Ordenación por mezcla (Merge Sort):

Este algoritmo consiste básicamente en dividir en partes iguales la lista de números y luego mezclarlos comparándolos, dejándolos ordenados.

Si se piensa en este algoritmo recursivamente, podemos imaginar que dividirá la lista hasta tener un elemento en cada lista, luego lo compara con el que está a su lado y según corresponda, lo sitúa donde corresponde. (d)

Complejidad: $O(n \log n)$ en todos los casos. Es un algoritmo estable (mantiene el orden relativo de elementos iguales) y eficiente para grandes conjuntos de datos, pero requiere espacio adicional.

Ordenamiento rápido (Quick Sort):

La idea del algoritmo es simple, se basa en la división en particiones de la lista a ordenar, por lo que se puede considerar que aplica la técnica divide y vencerás. El método es, posiblemente, el más pequeño de código, más rápido, más elegante, más interesante y eficiente de los algoritmos de ordenación conocidos.

- Es mucho más rápido que el Bubble Sort en la mayoría de los casos.

Complejidad: $O(n \log n)$ en el caso promedio, $O(n^2)$ en el peor caso (aunque esto es raro con una buena selección de pivote). Generalmente más rápido en la práctica que Merge Sort para conjuntos de datos grandes, ya que a menudo se puede hacer in-place (sin mucho espacio extra).

Algoritmo de intercambio

Este algoritmo no es el más eficiente, pero es muy didáctico. Consiste en tomar cada elemento y compararlo con los que están a su derecha, cada vez que se

identifica un par de elementos que no cumplen el criterio de ordenamiento que se está aplicando se intercambian. En cada iteración se encuentra el elemento que corresponde a una posición.

Que es la búsqueda en programación:

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arreglos y registros, y por ello será necesario determinar si un arreglo contiene un valor que coincida con un cierto valor clave.

El proceso de encontrar un elemento específico de un arreglo se denomina búsqueda.

Clasificaciones.

Tipos de Búsqueda:

El algoritmo consta de más de una etapa en el momento de la búsqueda se puede recurrir a diferentes algoritmos de búsqueda, estos poseen características que los hacen únicos para utilizarlos en situaciones diferentes.

Búsqueda lineal:

La búsqueda lineal, también conocida como búsqueda secuencial, va de elemento en elemento de la matriz, comparándolo con el elemento objetivo hasta que encuentra una coincidencia. Si la función de búsqueda llega al final de la matriz sin encontrar ninguna coincidencia, indica que el elemento objetivo no está presente en el conjunto de datos. El elemento buscado puede ser un número, una cadena, un objeto o cualquier otro tipo de dato que pueda almacenarse en una matriz. Esta flexibilidad hace de la búsqueda lineal un algoritmo versátil que puede aplicarse a una amplia gama de tipos de datos. El tiempo de ejecución de un algoritmo de búsqueda lineal es proporcional al tamaño de la matriz. La complejidad computacional de la búsqueda lineal es $O(n)$, donde n es el número de elementos de la matriz.

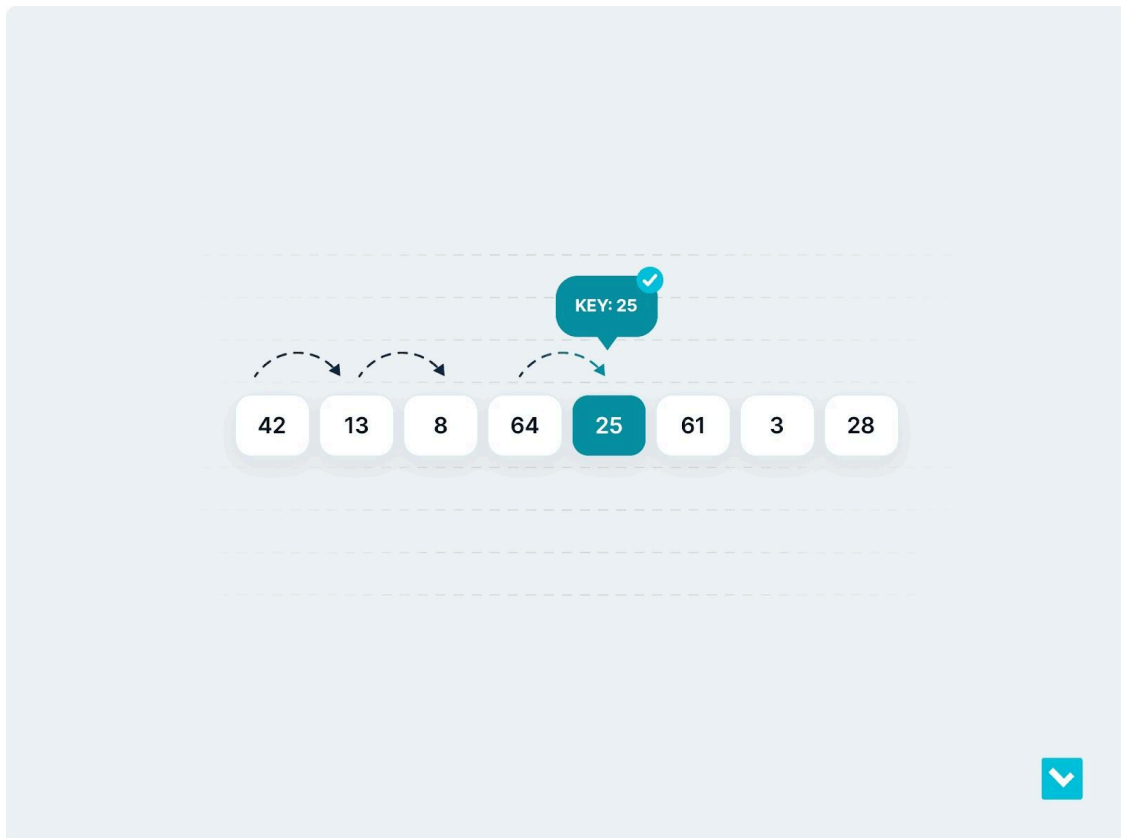


Figura 1. Comparación de diferentes algoritmos de búsqueda. **Nota.** Tomado de 6 Tipos de Algoritmos de Búsqueda Que Debes Conocer (2023), por Luigi's Box. <https://www.luigisbox.es/blog/tipos-de-algoritmos-de-busqueda/>

Aplicaciones de la búsqueda lineal

- Tareas de búsqueda sencillas: La búsqueda lineal suele utilizarse para un amplio espectro de tareas de búsqueda básicas en las que el conjunto de datos es pequeño o en las que no se justifica la complejidad adicional de otros algoritmos.
- Estructuras de datos lineales: La búsqueda lineal es fundamental para muchas estructuras de datos lineales, como matrices, listas y pilas.
- Matrices desordenadas: Dado que la búsqueda lineal comprueba cada elemento secuencialmente, puede localizar elementos independientemente de su posición en la matriz.

Pros

- Sencillo y fácil de implementar.
- Funciona en matrices sin ordenar.
- Adecuado para conjuntos de datos pequeños.

Contras

- No es adecuado para grandes conjuntos de datos debido al largo tiempo de ejecución.
- Tiene una complejidad temporal $O(n)$ en el peor de los casos.

Búsqueda binaria:

Es un algoritmo de búsqueda eficiente que funciona en conjuntos de datos ordenados. Divide el conjunto de datos en dos mitades y busca el elemento deseado en la mitad correspondiente. Repite este proceso hasta encontrar el elemento o determinar que no está en el conjunto de datos. Después, comprueba el valor objetivo con el elemento central y procede hacia la parte derecha o izquierda de la matriz. Este proceso continúa hasta que se encuentra el valor objetivo. El intervalo de búsqueda aparece vacío cuando el objetivo no está presente en la matriz.

El tiempo de ejecución de un algoritmo de búsqueda binaria es $O(\log n)$, donde n es el número de elementos de la matriz.

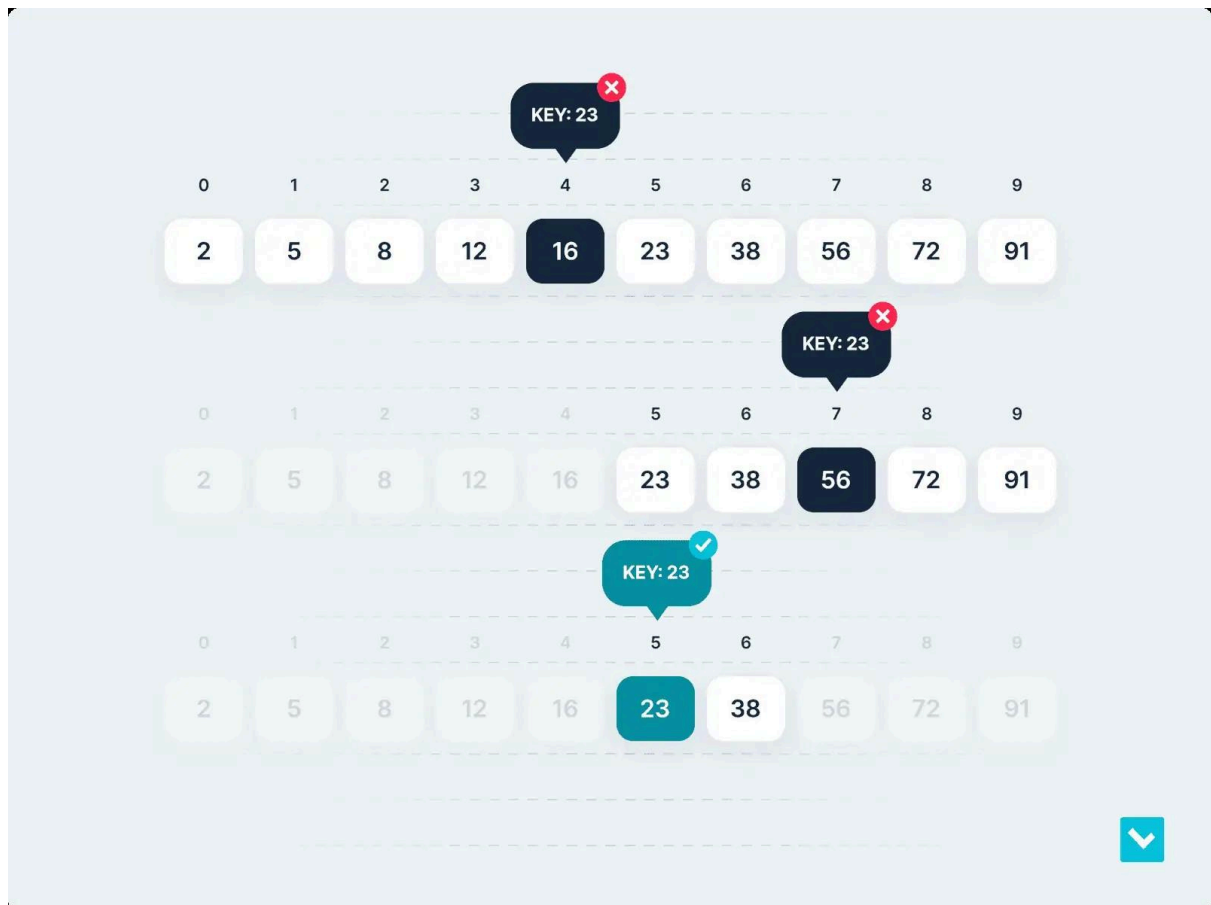


Figura 2. Comparación de diferentes algoritmos de búsqueda. **Nota.** Tomado de 6 *Tipos de Algoritmos de Búsqueda Que Debes Conocer* (2023), por Luigi's Box. <https://www.luigisbox.es/blog/tipos-de-algoritmos-de-busqueda/>

Aplicaciones de la búsqueda binaria

- Búsqueda en bases de datos: Es un algoritmo eficaz para la búsqueda en bases de datos para localizar rápidamente registros basados en campos indexados, como ID o claves.
- Búsqueda de elementos en matrices ordenadas: La búsqueda binaria localiza eficazmente elementos en matrices ordenadas, ejemplo la búsqueda de una palabra en un diccionario o la localización de un elemento en una lista ordenada.
- Base algorítmica: Es un componente fundamental en el desarrollo de sofisticados algoritmos de aprendizaje automático.

- Optimización de algoritmos: La búsqueda binaria se utiliza en algoritmos, incluidas las estrategias de divide y vencerás, para resolver eficazmente problemas como la búsqueda de raíces cuadradas o la localización de elementos máximos en matrices.
- Búsqueda en árboles: Los árboles binarios de búsqueda (BST) utilizan este algoritmo para buscar, insertar y eliminar elementos de forma eficaz manteniendo el orden del árbol.

Pros

- Un algoritmo muy eficaz para grandes colecciones de datos ordenados.
- Este algoritmo es mucho más rápido en matrices ordenadas que la búsqueda lineal.
- En comparación con muchos otros algoritmos, la búsqueda binaria es bastante sencilla.

Contras

- Necesita una matriz ordenada antes de buscar, lo que puede suponer una sobrecarga adicional.
- Requiere un acceso aleatorio a los elementos de la matriz,

Búsqueda de interpolación:

Es un algoritmo de búsqueda que mejora la búsqueda binaria al estimar la posición del elemento deseado en función de su valor. Puede ser más eficiente que la búsqueda binaria para conjuntos de datos grandes con una distribución uniforme de valores, ofrece una ventaja sobre la búsqueda binaria. La interpolación consiste en crear nuevos puntos de datos dentro del espacio de búsqueda conocido.

Mientras que la búsqueda binaria se centra siempre en el elemento central, la búsqueda por interpolación ajusta su posición de búsqueda en función del valor clave buscado. Por ejemplo, si la clave de búsqueda está más cerca del extremo, la búsqueda por interpolación tiende a empezar a buscar desde ese extremo.

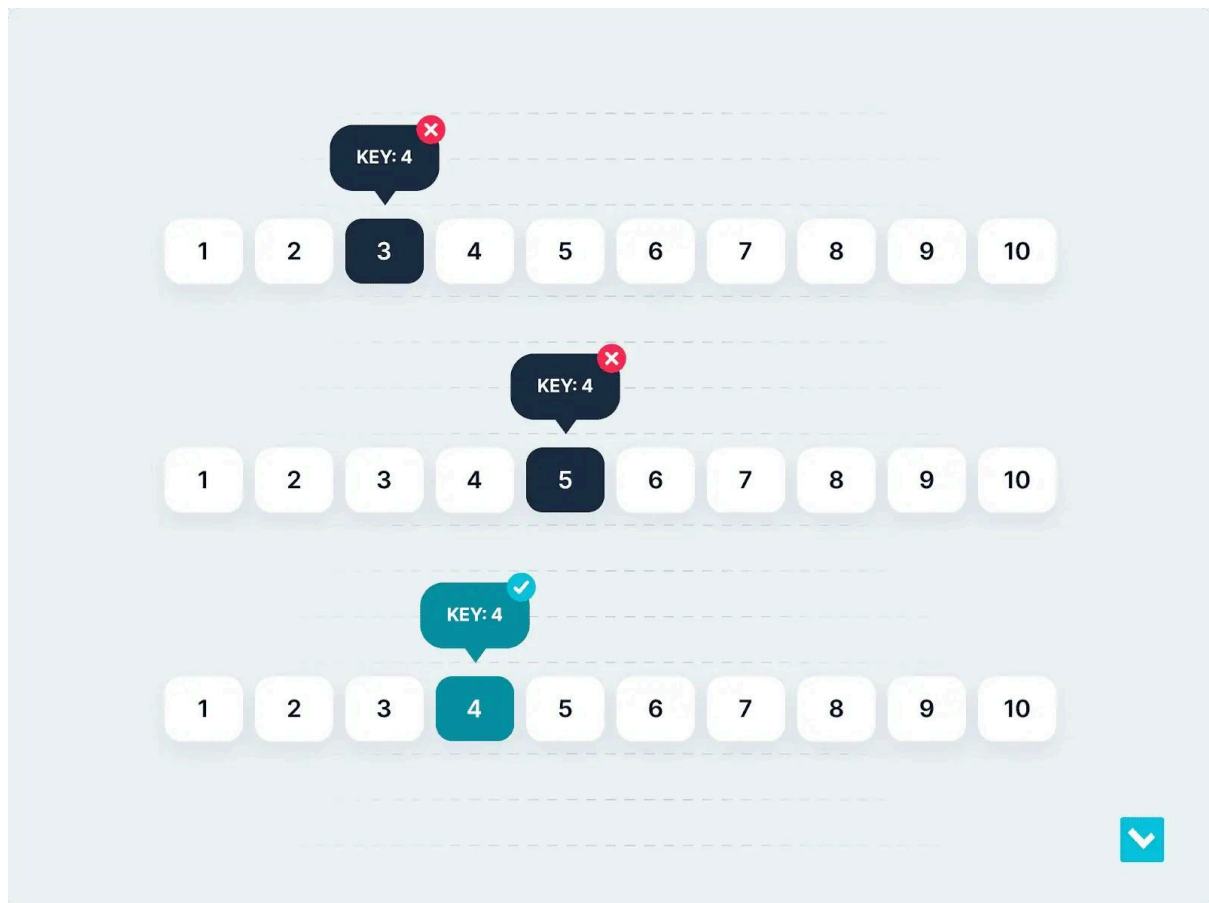


Figura 3. Comparación de diferentes algoritmos de búsqueda. **Nota.** Tomado de 6 Tipos de Algoritmos de Búsqueda Que Debes Conocer (2023), por Luigi's Box. <https://www.luigisbox.es/blog/tipos-de-algoritmos-de-busqueda/>

Aplicaciones de la búsqueda por interpolación

Búsqueda en bases de datos: La búsqueda por interpolación se utiliza habitualmente en bases de datos para buscar registros en función de algún elemento clave.

Búsqueda en matrices grandes: Este algoritmo de búsqueda facilita una búsqueda más eficiente en grandes matrices ordenadas, especialmente cuando la distribución de valores es uniforme.

Recuperación de datos de tablas de símbolos: Esta puede utilizarse en tablas de símbolos, donde las claves están asociadas a valores. Por ejemplo, en los lenguajes de programación, las tablas de símbolos se utilizan para almacenar variables,

funciones y otros identificadores. La búsqueda por interpolación puede localizar rápidamente la información asociada a un identificador específico.

Aplicaciones sensibles al tiempo: Debido a su capacidad para recuperar datos rápidamente, esta es adecuada para aplicaciones sensibles al tiempo.

Pros

La búsqueda por interpolación es eficaz para matrices ordenadas de gran tamaño.

Este algoritmo de búsqueda tiene la complejidad temporal más eficiente($\log n$).

La búsqueda por interpolación puede identificar antes si el elemento objetivo no está en la matriz, lo que permite resoluciones de búsqueda más rápidas.

Contras

Para realizar la búsqueda de interpolación, hay que ordenar la matriz.

Posible disminución de la eficacia si la matriz no está distribuida uniformemente.

Algoritmo de Salto Búsqueda por saltos

La búsqueda por saltos, también conocida como búsqueda por bloques, está diseñada específicamente para matrices ordenadas. A diferencia de la búsqueda lineal, que examina cada elemento secuencialmente, la búsqueda por saltos salta en incrementos de tamaño fijo, omitiendo múltiples elementos para realizar búsquedas más rápidas.

Si el elemento objetivo se encuentra dentro de un rango especificado, el algoritmo emplea una búsqueda lineal dentro de ese bloque. Este enfoque reduce sustancialmente las operaciones, mejorando la eficacia de la búsqueda.

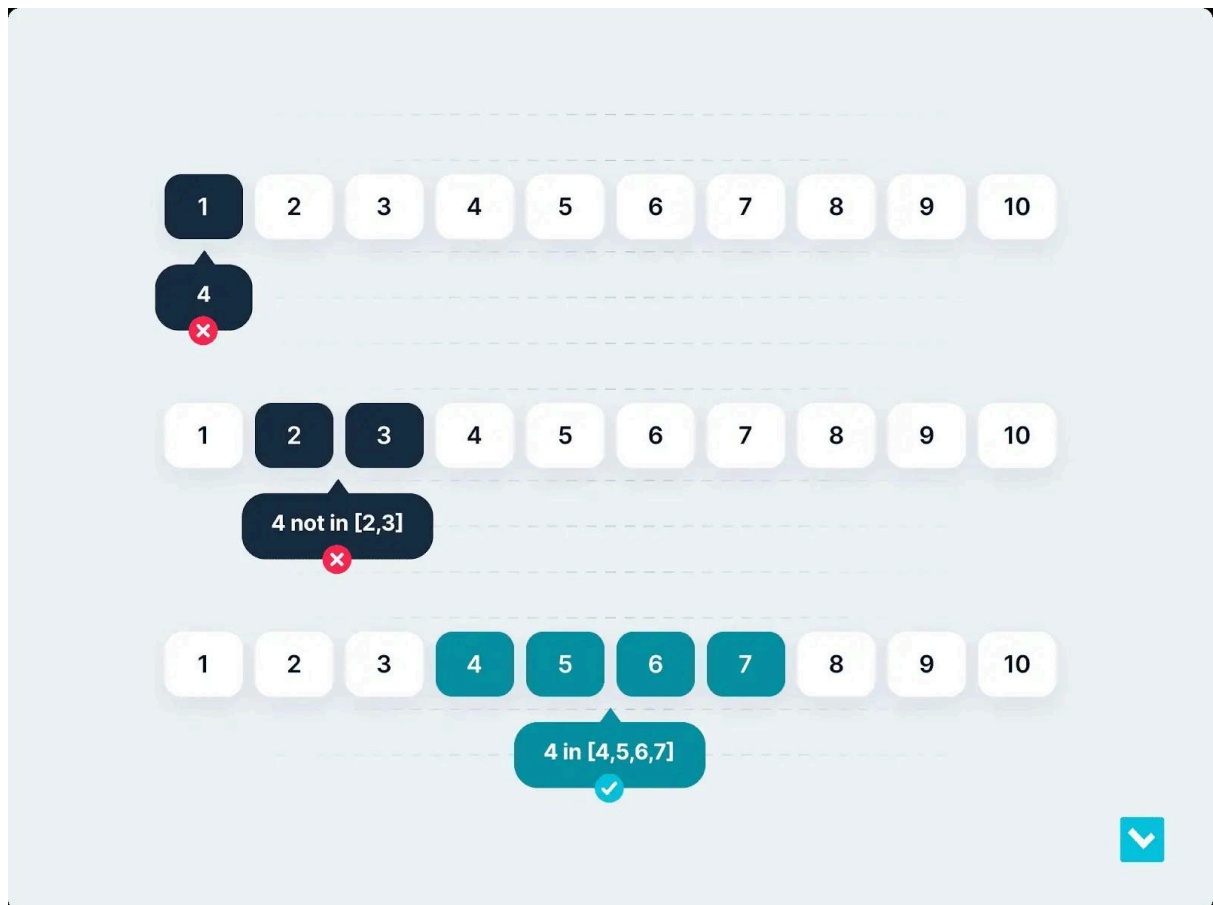


Figura 4. Comparación de diferentes algoritmos de búsqueda. **Nota.** Tomado de 6 *Tipos de Algoritmos de Búsqueda Que Debes Conocer* (2023), por Luigi's Box. <https://www.luigisbox.es/blog/tipos-de-algoritmos-de-busqueda/>

Aplicaciones de la búsqueda por saltos

Aplicaciones JavaScript: La búsqueda por saltos optimiza el proceso de búsqueda en aplicaciones JavaScript, mejorando la eficacia y la capacidad de respuesta.

Búsqueda en sistemas de bases de datos: Este algoritmo busca registros en función de algún elemento clave en grandes bases de datos.

Recuperación de datos de medios visuales: La búsqueda por saltos localiza con eficacia un fotograma específico dentro de un archivo de vídeo de gran tamaño avanzando un número fijo de fotogramas cada vez.

Pros

Es mejor que una búsqueda lineal para matrices grandes y uniformemente distribuidas.

La búsqueda por saltos tiene una complejidad temporal menor que la búsqueda lineal.

La implementación es menos complicada en comparación con algoritmos como la búsqueda binaria o la búsqueda ternaria.

Contras

La búsqueda por saltos requiere que la matriz de elementos esté ordenada, lo que no siempre es posible o práctico.

Posible rendimiento subóptimo con datos distribuidos de forma desigual.(g)

- **Búsqueda de hash**

Es un algoritmo de búsqueda que utiliza una función hash para asignar cada elemento a una ubicación única en una tabla hash. Esto permite acceder a los elementos en tiempo constante, lo que lo hace muy eficiente para conjuntos de datos grandes.

Complejidad logarítmica

La complejidad de un algoritmo es una medida de cuán eficiente es el algoritmo para resolver el problema. En otras palabras, es una medida de cuánto tiempo y espacio (memoria) requiere el algoritmo para producir una solución. Las complejidades de tiempo y espacio de un algoritmo se pueden expresar utilizando la notación O grande. Esta notación proporciona una forma de comparar la complejidad de diferentes algoritmos. Por ejemplo, un algoritmo con una complejidad temporal de $O(n)$ se considera más eficiente que un algoritmo con una complejidad temporal de $O(n^2)$, porque crece a un ritmo más lento a medida que aumenta el tamaño de entrada. El objetivo del diseño de algoritmos es encontrar algoritmos que sean eficientes y fáciles de implementar.

El estudio de la complejidad de los algoritmos es una parte importante tanto de las matemáticas como de la informática. Nos ayuda a comprender las limitaciones de los algoritmos y las compensaciones involucradas en la resolución de problemas complejos.

Principales órdenes de complejidad

Orden	Nombre
$O(1)$	constante
$O(\log n)$	logarítmica
$O(n)$	lineal
$O(n \log n)$	casi lineal
$O(n^2)$	cuadrática
$O(n^3)$	cúbica
$O(a^n)$	exponencial

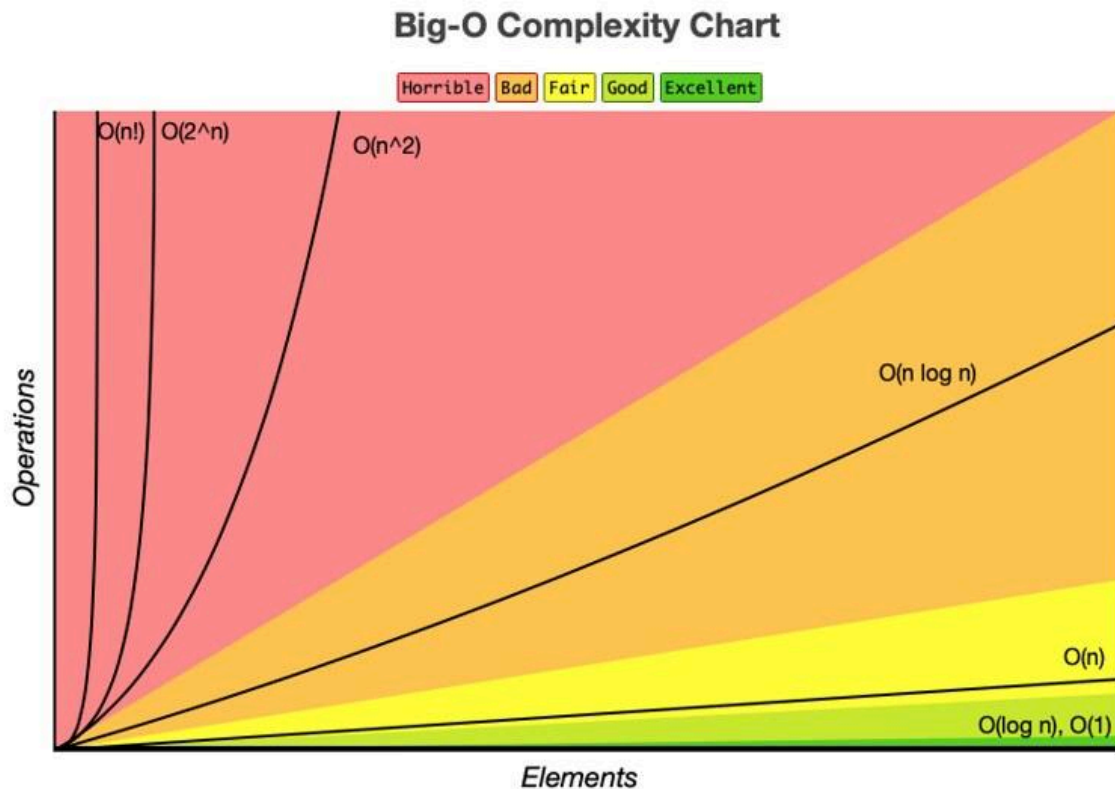


Figura 5. Ejemplo de la complejidad de un algoritmo. **Nota.** Tomado de *La complejidad de los algoritmos* (2023), por European Valley.
<https://europeanvalley.es/noticias/la-complejidad-de-los-algoritmos/>

Comparación entre algoritmos de ordenamiento según su complejidad

Algoritmo	Mejor Caso	Peor Caso	Caso Promedio	Eficiencia
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Baja
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Baja
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Media
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	Alta

Cuadro extraído de la bibliografía de la Clase de Programación: Búsquedas y Ordenamiento en Python

Importancia de la búsqueda y el ordenamiento.

- **Eficiencia:** Mejoran el tiempo de ejecución de programas que manejan grandes cantidades de datos.
- **Organización:** Permiten trabajar con datos de forma más clara. Facilitan la estructuración y presentación de estos de manera clara,coherente, simplificando su análisis y comprensión.
- **Eficacia:** Al optimizar el acceso y la manipulación de datos, estos algoritmos mejoran significativamente el tiempo de ejecución de programas, especialmente aquellos que manejan grandes volúmenes de información.
- **Escalabilidad:** Su diseño permite manejar conjuntos de datos de diferentes tamaños, adaptándose a las necesidades cambiantes de un programa.
- **Precisión:** Garantizan la recuperación de resultados exactos y relevantes, evitando errores y ambigüedades en la búsqueda de información.
- **Versatilidad:** Se aplican en una amplia gama de contextos y dominios, desde bases de datos y sistemas de archivos hasta aplicaciones web y motores de búsqueda.(f)

3. Caso Práctico

Problema a resolver

Se requiere desarrollar un programa en Python para una librería digital ya existente y cargada con datos de los 1000 libros más elegidos, con sus títulos en español. El objetivo principal es implementar y comparar la eficiencia de diferentes algoritmos de búsqueda (por título, autor o año) y ordenamiento (por puntuación) sobre esta colección de libros. El sistema deberá permitir al usuario interactuar con la

funcionalidad de búsqueda y visualización de resultados ordenados, al mismo tiempo que ofrece una comparación empírica del rendimiento de los algoritmos utilizados para cada operación.

Código fuente comentado

Se realiza el programa con tres archivos para una mejor organización y lectura del código.

El archivo 'algoritmo.py' contiene las funciones de carga de libros, ordenamiento y búsqueda.

```
import csv

def cargar_libros(archivo):
    """
    Realiza la carga un archivo CSV con datos de libros.

    Parámetros: archivo (str): Ruta del archivo CSV.

    Retorna: list: Lista de diccionarios con claves: 'titulo', 'autor' y 'puntuacion'.
    """
    # Abrir el archivo CSV en modo lectura con codificación UTF-8

    libros = []

    with open(archivo, newline='', encoding='utf-8') as archivo_csv:
        lector = csv.DictReader(archivo_csv) # Leer filas como diccionarios

        # Convertir cada fila en diccionario con puntuación numérica

        for fila in lector:
            fila["puntuacion"] = float(fila["puntuacion"]) # Convertir a float
```

```

        libros.append(fila)

    return libros

def bubble_sort(lista, clave):
    """
        Ordena una lista de diccionarios en orden descendente según el valor
        asociado a una clave específica.

        Implementa el algoritmo de ordenamiento Bubble Sort (ordenamiento
        burbuja).

        Parámetros:

        lista --> lista de diccionarios a ordenar

        clave --> clave del diccionario por la cual se realizará el ordenamiento

        Retorna:

        La lista ordenada en orden descendente (modificada en el lugar)
    """
    n = len(lista) # Se obtiene la cantidad de elementos de la lista

    # Bucle externo que recorre la lista
    for i in range(n):

        # Bucle interno para comparar elementos seguidos
        for j in range(0, n - i - 1):

            # Si el elemento actual es mayor que el siguiente, se intercambian
            if lista[j][clave] < lista[j + 1][clave]:

```

```

        lista[j], lista[j + 1] = lista[j + 1], lista[j]

    return lista # Se devuelve la lista ordenada

def selection_sort_desc(lista, clave):
    """
        Ordena una lista de diccionarios en orden descendente según el valor
        asociado a una clave específica.

        Utiliza el algoritmo de ordenamiento por selección (Selection Sort).

        Parámetros:

        lista --> lista de diccionarios a ordenar

        clave --> clave del diccionario por la cual se realizará el ordenamiento

        Retorna:

        La lista ordenada en orden descendente (modificada en el lugar)

    """
    n = len(lista) # Se obtiene la longitud de la lista

    # Recorremos todos los elementos de la lista
    for i in range(n):

        # Encontrar el índice del elemento máximo

        max_index = i

        # Comparamos este elemento con todos los demás que siguen en la lista
        for j in range(i + 1, n):

```

```

        # Si encontramos un elemento con mayor valor, actualizamos el
índice

        if lista[j][clave] > lista[max_index][clave]:

            max_index = j

        # Intercambiamos el elemento actual con el elemento de mayor valor
encontrado

        lista[i], lista[max_index] = lista[max_index], lista[i]

    return lista # Se devuelve la lista ordenada

def insertion_sort_desc(lista, clave):
    """
        Ordena una lista de diccionarios en orden descendente según el valor
asociado a una clave específica.

        Utiliza el algoritmo de ordenamiento por inserción (Insertion Sort).

        Parámetros:

        lista: lista de diccionarios a ordenar

        clave: clave del diccionario por la cual se realizará el ordenamiento

        Retorna:

        La lista ordenada en orden descendente (modificada en el lugar)
    """
    # Recorremos la lista desde el segundo elemento hasta el final

```

```

for i in range(1, len(lista)):

    actual = lista[i] # Elemento que se va a insertar

    j = i - 1

    # Mover los elementos que sean menores que el actual a posiciones
mayores

    while j >= 0 and lista[j][clave] < actual[clave]:

        lista[j + 1] = lista[j]

        j -= 1

    # Insertamos el elemento en su posición correcta

    lista[j + 1] = actual

return lista # Se devuelve la lista ordenada en orden descendente

def quicksort(lista, clave):

    """

    Ordena una lista de diccionarios en orden descendente según el valor de
una clave específica.

    Implementa el algoritmo de ordenamiento Quick Sort.

    Parámetros:

    lista --> lista de diccionarios a ordenar (por ejemplo, libros)

    clave --> clave del diccionario por la cual se realizará el ordenamiento
(por ejemplo, "puntuacion")

```


Retorna:

Una nueva lista ordenada en orden descendente

```
"""
```

```
# Caso base: si la lista tiene 0 o 1 elemento, ya está ordenada
```

```
if len(lista) <= 1:
```

```
    return lista
```

```
else:
```

```
    # Elegimos el primer elemento como pivote
```

```
    pivot = lista[0]
```

```
    # Elementos con valor igual o mayor al pivote (para orden descendente)
```

```
    greater = [x for x in lista[1:] if x[clave] >= pivot[clave]]
```

```
    # Elementos con valor menor al pivote
```

```
    less = [x for x in lista[1:] if x[clave] < pivot[clave]]
```

```
    # Llamada recursiva: ordenamos las sublistas y unimos todo
```

```
    return quicksort(greater, clave) + [pivot] + quicksort(less, clave)
```

```
def busqueda_lineal(lista, valor):
```

```
    """
```

```
    Realiza una búsqueda lineal de libros con una puntuación específica.
```

```

Parámetros:

lista (list): Lista de libros.

valor (float): Puntuación a buscar.


Retorna:

list: Lista de libros que coinciden con la puntuación.

"""

    resultados = [] # Lista vacía que guarda los libros que cumplan con la
condición

    # Recorremos cada libro en la lista
    for libro in lista:

        if libro["puntuacion"] == valor:

            resultados.append(libro) # Lo agregamos a la lista de resultados

    return resultados

def busqueda_binaria(lista, valor):

    """

    Realiza una búsqueda binaria de libros por puntuación en una lista
ordenada de forma descendente.

    Importante: la lista debe estar ordenada por puntuación previamente.

    Retorna:

    list: Libros encontrados con la puntuación exacta.

```

```

"""

izquierda = 0 # Límite inferior de la búsqueda

derecha = len(lista) - 1 # Límite superior de la búsqueda

resultados = [] # Lista vacía que guarda los libros encontrados


while izquierda <= derecha:

    medio = (izquierda + derecha) // 2 # Índice del punto medio

    puntuacion_medio = lista[medio]["puntuacion"] # Se obtiene la
puntuación del libro en la posición media


    if puntuacion_medio == valor:

        # Si encuentra una coincidencia, busca alrededor por más
resultados iguales

        resultados.append(lista[medio])


        # Buscar hacia la izquierda del centro

        i = medio - 1

        while i >= 0 and lista[i]["puntuacion"] == valor:

            resultados.insert(0, lista[i]) # insertamos al inicio

            i -= 1


        # Buscar hacia la derecha del centro

        i = medio + 1

        while i < len(lista) and lista[i]["puntuacion"] == valor:

            resultados.append(lista[i]) # insertamos al final

```

```

        i += 1

        break # Ya encontramos todas las coincidencias

    elif puntuacion_medio < valor:

        derecha = medio - 1 # Valor está a la izquierda (porque es más
alto)

    else:

        izquierda = medio + 1 # Valor está a la derecha (porque es más
bajo)

    return resultados # Devuelve la lista con los libros encontrados (puede
estar vacía)

```

El archivo '[main.py](#)' importa las funciones desde algoritmos.py. Carga los datos una vez y ejecuta cada algoritmo. Mide el tiempo con `timeit.default_timer()` y muestra los resultados

```

import os

import timeit

from algoritmos import cargar_libros, bubble_sort, selection_sort_desc,
insertion_sort_desc, quicksort, busqueda_lineal, busqueda_binaria

# Cargar libros desde el archivo

directorio_actual = os.path.dirname(__file__)

ruta_archivo = os.path.join(directorio_actual, "libros-famosos.csv")

libros = cargar_libros(ruta_archivo)

```

```

# Elegir una puntuación objetivo

objetivo = 2.3

# Medir ordenamiento

# Bubble Sort

libros_bubble = libros.copy()

start_bubble = timeit.default_timer()

ordenados_bubble = bubble_sort(libros_bubble, "puntuacion")

end_bubble = timeit.default_timer()

print(f"Bubble Sort: Tiempo = {end_bubble - start_bubble:.6f} segundos")

print("Top 15 con Bubble Sort:")

for libro in ordenados_bubble[:15]:

    print(f"{libro['titulo']} - {libro['puntuacion']}")

# Selection Sort

libros_selection = libros.copy()

start_selection = timeit.default_timer()

ordenados_selection = selection_sort_desc(libros_selection, "puntuacion")

end_selection = timeit.default_timer()

print(f"Selection Sort: Tiempo = {end_selection - start_selection:.6f} segundos")

print("Top 15 con Selection Sort:")

for libro in ordenados_selection[:15]:

    print(f"{libro['titulo']} - {libro['puntuacion']}")

```

```

# Insertion Sort

libros_insertion = libros.copy()

start_insertion = timeit.default_timer()

ordenados_insertion = insertion_sort_desc(libros_insertion, "puntuacion")

end_insertion = timeit.default_timer()

print(f"Insertion Sort: Tiempo = {end_insertion - start_insertion:.6f} segundos")

print("Top 15 con Insertion Sort:")

for libro in ordenados_insertion[:15]:

    print(f"{libro['titulo']} - {libro['puntuacion']}")


# Quick Sort

libros_quick = libros.copy()

start = timeit.default_timer()

ordenados_quick = quicksort(libros.copy(), "puntuacion")

end = timeit.default_timer()

print(f"Quick Sort: Tiempo = {end - start:.6f} segundos")

print("Top 15 con Quick Sort:")

for libro in ordenados_quick[:15]:

    print(f"{libro['titulo']} - {libro['puntuacion']}")


#Medir búsqueda

# Búsqueda lineal sobre la lista ordenada con bubble_sort

start_busqueda = timeit.default_timer()

```

```

resultados = busqueda_lineal(ordenados_bubble, objetivo)

end_busqueda = timeit.default_timer()

print(f"Búsqueda Lineal (ordenada con bubble): Resultados = {len(resultados)},
Tiempo = {end_busqueda - start_busqueda:.6f} segundos")

# Búsqueda lineal sobre la lista ordenada con quick sort

start_busqueda = timeit.default_timer()

resultados = busqueda_lineal(ordenados_quick, objetivo)

end_busqueda = timeit.default_timer()

print(f"Búsqueda Lineal: (ordenada con quick) = {len(resultados)}, Tiempo =
{end_busqueda - start_busqueda:.6f} segundos")

# Búsqueda binaria sobre lista ordenada (bubble sort)

start_binaria = timeit.default_timer()

resultado_binaria = busqueda_binaria(ordenados_bubble, objetivo)

end_binaria = timeit.default_timer()

if resultado_binaria:

    print(f"Búsqueda Binaria (ordenada con bubble): Resultados =
{len(resultado_binaria)}, Tiempo = {end_binaria - start_binaria:.6f}
segundos")

else:

    print(f"Búsqueda Binaria: No encontrado, Tiempo = {end_binaria -
start_binaria:.6f} segundos")

# Búsqueda binaria sobre lista ordenada con quick sort

start_binaria = timeit.default_timer()

```

```

resultado_binaria = busqueda_binaria(ordenados_quick, objetivo)

end_binaria = timeit.default_timer()

if resultado_binaria:

    print(f"Búsqueda Binaria (ordenada con quick): Resultados =
{len(resultado_binaria)}, Tiempo = {end_binaria - start_binaria:.6f}
segundos")

else:

    print(f"Búsqueda Binaria: No encontrado, Tiempo = {end_binaria -
start_binaria:.6f} segundos")

# Buscar libros con puntuación 5.0 (lineal)
resultado_5 = busqueda_lineal(libros, 5.0)

print("Libros con puntuación 5.0 (busqueda lineal):")

for libro in resultado_5:

    print(libro["titulo"])

objetivo_binario = 1.0

start_binaria = timeit.default_timer()

resultados_binarios = busqueda_binaria(ordenados_quick, objetivo_binario)

end_binaria = timeit.default_timer()

print(f"Libros con puntuación {objetivo_binario} (busqueda binaria):")

for libro in resultados_binarios:

    print(f"{libro['titulo']} - {libro['puntuacion']}")

```


Explicación de decisiones de diseño

El archivo 'libros.famosos.csv' contiene 1000 datos de libros con su título, autor, género y puntaje.

En primera instancia se ejecutaron, para el caso presentado, cuatro métodos de ordenamiento y los dos métodos de búsqueda y para determinar cuál es el más conveniente, se midieron los tiempos de ejecución.

Para el caso del ordenamiento se decide hacerlo de forma descendente. Esto se debe a que consideramos prioritario un orden que le permita al usuario ver la lista de libros comenzado por los libros con puntuación más alta.

Debido al ordenamiento elegido, fue necesario hacer ajustes en el bloque de código de búsqueda binaria, ya que el método clásico solo funciona si la lista está ordenada de forma ascendente (menor a mayor). En nuestro caso, como hemos mencionado, el ordenamiento es descendente y no ascendente, por tratarse de puntuaciones de libros.

Para leer el archivo, se opta por un método de lectura del archivo "with open(...) as archivo" que asegure la apertura y el cierre del mismo luego de ejecutarse el bloque de código. Se utiliza `encoding="utf-8"` para prevenir errores de decodificación de lectura, usuales cuando hay símbolos especiales como la ñ o tildes. Se implementa `json.load(archivo)` para convertir el texto en una estructura Python, en este caso, en un diccionario.

Validación del funcionamiento

Para asegurarnos de que los algoritmos funcionan correctamente:

- Se verifica que los métodos de ordenamiento (Bubble, Selection, Insertion, Quick) dejan la lista de libros correctamente ordenada según la puntuación. Esto se confirma imprimiendo la lista ordenada.
- Se muestra un ejemplo con los 15 primeros libros utilizando cada caso

Bubble Sort: Tiempo = 0.062185 segundos

Top 15 con Bubble Sort:

Cometas en el cielo - 5.0
El nombre del viento - 5.0
Las crónicas de Narnia - 5.0
Matar a un ruiseñor - 5.0
Las crónicas de Narnia - 5.0
El perfume - 5.0
El alquimista - 5.0
El niño con el pijama de rayas - 5.0
Ensayo sobre la ceguera - 5.0
Orgullo y prejuicio - 5.0
Las crónicas de Narnia - 5.0
Matar a un ruiseñor - 4.9
Ensayo sobre la ceguera - 4.9
La sombra del viento - 4.9
Los pilares de la tierra - 4.9

Selection Sort: Tiempo = 0.041286 segundos

Top 15 con Selection Sort:

Cometas en el cielo - 5.0
El nombre del viento - 5.0
Las crónicas de Narnia - 5.0
Matar a un ruiseñor - 5.0
Las crónicas de Narnia - 5.0
El perfume - 5.0
El alquimista - 5.0
El niño con el pijama de rayas - 5.0
Ensayo sobre la ceguera - 5.0
Orgullo y prejuicio - 5.0
Las crónicas de Narnia - 5.0
Matar a un ruiseñor - 4.9
Ensayo sobre la ceguera - 4.9
La sombra del viento - 4.9
Los pilares de la tierra - 4.9

```
Insertion Sort: Tiempo = 0.028839 segundos
Top 15 con Insertion Sort:
Cometas en el cielo - 5.0
El nombre del viento - 5.0
Las crónicas de Narnia - 5.0
Matar a un ruiseñor - 5.0
Las crónicas de Narnia - 5.0
El perfume - 5.0
El alquimista - 5.0
El niño con el pijama de rayas - 5.0
Ensayo sobre la ceguera - 5.0
Orgullo y prejuicio - 5.0
Las crónicas de Narnia - 5.0
Matar a un ruiseñor - 4.9
Ensayo sobre la ceguera - 4.9
La sombra del viento - 4.9
Los pilares de la tierra - 4.9
```

```
Quick Sort: Tiempo = 0.004448 segundos
Top 15 con Quick Sort:
Las crónicas de Narnia - 5.0
Orgullo y prejuicio - 5.0
Ensayo sobre la ceguera - 5.0
El niño con el pijama de rayas - 5.0
El alquimista - 5.0
El perfume - 5.0
Las crónicas de Narnia - 5.0
Matar a un ruiseñor - 5.0
Las crónicas de Narnia - 5.0
El nombre del viento - 5.0
Cometas en el cielo - 5.0
Drácula - 4.9
El perfume - 4.9
Los pilares de la tierra - 4.9
Frankenstein - 4.9
```

- Se prueba que la búsqueda binaria sólo encuentra los libros si la lista está ordenada, ascendente o descendente. Se ajusta la lógica para el orden descendente.
- Se usa una función de medición de tiempo para confirmar que cada función se ejecuta correctamente y devuelve resultados esperados.
- Se verificó visualmente que los resultados devueltos son los correctos, como por ejemplo que la búsqueda lineal encuentra todos los libros con puntuación 5.0.

```
Búsqueda Lineal: Resultados = 11, Tiempo = 0.000128 segundos  
Búsqueda Binaria: Resultados = 11, Tiempo = 0.000023 segundos
```

Algoritmos implementados

- Búsqueda:
 - `busqueda_lineal(lista, clave)`
 - `busqueda_binaria(lista_ordenada, clave, campo='titulo')`
- Ordenamiento:
 - `quicksort(lista, campo='puntuacion')`
 - `insertion_sort(lista, campo='puntuacion')`
 - `selection_sort_desc(lista, clave)`

4. Metodología Utilizada

Para el desarrollo del presente trabajo, se adoptó un enfoque metodológico que combinó la investigación documental con el análisis práctico. Las etapas se detallan a continuación:

Revisión bibliográfica y marco teórico:

Se investigaron fuentes oficiales, unidades académicas, Wikipedia, artículos especializados y el uso de inteligencia artificial para establecer un marco conceptual sólido sobre los algoritmos de búsqueda y ordenamiento.

Pruebas experimentales y análisis de rendimiento

Las pruebas experimentales realizadas tuvieron como objetivo principal evaluar el rendimiento y la eficiencia de distintos algoritmos de búsqueda y ordenamiento. Nuestro enfoque primordial fue determinar la velocidad de ejecución de cada algoritmo en un caso de uso práctico y relevante: la búsqueda de libros en un sistema de biblioteca con puntuación según su relevancia comercial.

En este escenario, la capacidad de un sistema para localizar rápidamente un libro entre un vasto catálogo es crucial para la experiencia del usuario. Por ello, estas pruebas no solo nos permitieron observar el comportamiento teórico de los

algoritmos, sino también medir su desempeño real bajo condiciones controladas. Se buscó identificar cuál de los algoritmos probados ofrecía la solución más eficiente para la gestión y consulta del inventario de una biblioteca, considerando que la velocidad de respuesta es un factor determinante en la usabilidad de la aplicación.

Etapas de diseño y prueba del código

Para el diseño del código se siguieron los ejemplos expuestos en el material bibliográfico de la materia Programación I.

Se realizaron pruebas de código para comprender el funcionamiento de los algoritmos de búsqueda y de ordenamiento.

En una segunda instancia se pusieron a prueba los algoritmos, para conocer el tiempo de ejecución de cada uno, tomando como ejemplo, una lista de números.

Luego, se continuó con el desarrollo del programa, para la búsqueda y ordenamiento de libros, realizando una adaptación del código que permita trabajar ya no con listas, sino con diccionarios.

El trabajo comenzó con una etapa de investigación y luego siguió con el intercambio de datos. A partir de la lectura y la práctica, se determinó el caso de estudio y se procedió a definir el marco teórico. Luego, se decidió profundizar de forma individual en los algoritmos para concluir con un intercambio de información. Para el diseño del código se trabajó en conjunto, evidenciando complejidades con el uso del Github.

5. Resultados Obtenidos

Detalla qué se logró con el caso práctico, qué aspectos funcionaron correctamente y qué dificultades se presentaron.

Se pueden incluir:

Casos de prueba realizados

Algoritmo	Tiempo (segundos)
------------------	--------------------------

Bubble Sort	0.056221
Selection Sort	0.036758
Insertion Sort	0.031779
Quick Sort	0.004792
Búsqueda lineal	0.000166
Búsqueda binaria	0.000022

Para el caso del ordenamiento:

Se puede comprobar que Quick Sort fue el más rápido (7-10 veces más rápido que Bubble).

Los algoritmos cuadráticos (Bubble, Selection, Insertion) toman mucho más tiempo.

Bubble Sort fue el peor (el más lento).

Para el caso de la búsqueda:

La búsqueda binaria fue notablemente más rápida que la lineal (aprox. 7.5 veces). Pero requiere que los datos estén ordenados, lo que implica un coste previo de ordenamiento.

En conclusión

Quick Sort, de acuerdo con la teoría, es el más eficiente para listas grandes (como el ejemplo utilizado). Queda evidenciado que algoritmos como Bubble y Selection son ineficientes en la práctica, por su alta complejidad temporal. Insertion Sort es útil para listas pequeñas o casi ordenadas.

Cabe destacar que Quick Sort mostró eficiencia para el caso estudiado pero también presenta riesgos en cuanto a los resultados, ya que si hay elementos con claves iguales, puede cambiar su orden relativo.

Ejemplo: Si hay dos libros con puntuación 5.0, podrían quedar en diferente orden tras ordenar.

Por otro lado, si bien el promedio es $O(n \log n)$, su peor caso es $O(n^2)$. Esto puede ocurrir, por ejemplo, si el pivote elegido es siempre el mínimo o el máximo, como al ordenar una lista ya ordenada. Esto puede provocar una gran cantidad de llamadas recursivas y mala eficiencia.

En cuanto a la búsqueda binaria es la mejor opción si la lista está ordenada. La búsqueda lineal sigue siendo útil para listas pequeñas o desordenadas.

Además se estudió el caso de búsquedas con listas ordenadas por método Bumble y por método Quick. Aunque Quick Sort ordena mucho más rápido que Bubble Sort, la búsqueda lineal tarda algo más sobre la lista ordenada con Quick Sort debido a la diferencia en el orden específico de los elementos, y a que la búsqueda lineal no aprovecha la ordenación ni optimiza su recorrido. Además, la diferencia de tiempos es muy pequeña y puede deberse a factores externos.

6. Conclusiones

Este trabajo práctico sobre algoritmos de búsqueda y ordenamiento en Python nos permitió comprender de manera sólida conceptos fundamentales de la programación. A partir de la investigación teórica, logramos entender el funcionamiento de distintos algoritmos y luego aplicarlos en código, integrando teoría y práctica.

Implementamos métodos de búsqueda lineal y binaria, y algoritmos de ordenamiento como Bubble Sort, Selection Sort, Insertion Sort y Quick Sort. Esto nos permitió comparar su funcionamiento, eficiencia y adecuación según el tipo de datos. Además, incorporamos una librería externa, lo cual enriqueció el proyecto y nos introdujo en el uso de herramientas adicionales en Python.

El desarrollo presentó algunos desafíos, especialmente al combinar diferentes algoritmos de búsqueda y ordenamiento. Sin embargo, estas dificultades fueron clave para nuestro aprendizaje, ya que nos impulsaron a razonar de forma crítica y a encontrar soluciones creativas.

En resumen, este trabajo fortaleció nuestra comprensión de los algoritmos, su aplicación práctica y su impacto en la eficiencia de los programas. Además,


representa un primer paso hacia la implementación de soluciones más complejas en el futuro, ampliando nuestra base como programadores.

7. Bibliografía

- (a) 4Geeks. (s.f.). Algoritmos de Ordenamiento y Búsqueda en Python: Optimizando la Gestión de Datos. <https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python>
- (b) Meneses, E. R. (2021). *Clase9-Ordenamiento*. Universidad Veracruzana. <https://www.uv.mx/personal/ermeneses/files/2021/08/Clase9-Ordenamiento.pdf>
- (c) Universidad X. (s.f.). Búsqueda y Ordenamiento en Programación. Recuperado de <https://www.universidadx.edu/recursos/busquedayordenamiento.pdf>
- (d) Jorge Antilef Blog. (s.f.). Algoritmos de búsqueda y ordenamiento. Recuperado de <https://jorgeantilefblog.wordpress.com/algoritmos-de-busqueda-y-ordenamiento/>
- (e) Pérez, J. (2023). Principios de psicología. Editorial ABC. <https://books.google.com/books?id=ejemplo123>
- (f) García, M. (2023, 15 de marzo). La importancia de los algoritmos de búsqueda y ordenamiento. Blog de Programación. <https://www.ejemplo.com/algoritmos-busqueda-ordenamiento>
- Luigi's Box. (2023, 29 de mayo). 6 Tipos de Algoritmos de Búsqueda Que Debes Conocer. <https://www.luigisbox.es/blog/tipos-de-algoritmos-de-busqueda/>
- (h) **La complejidad de los algoritmos simples y las estructuras de datos en JS.** (s.f.). freeCodeCamp. <https://www.freecodecamp.org/espanol/news/la-complejidad-de-los-algoritmos-simples-y-las-estructuras-de-datos-en-js-videos>

- European Valley. (2023, 10 de marzo). La complejidad de los algoritmos. European Valley.
<https://europeanvalley.es/noticias/la-complejidad-de-los-algoritmos/>

8. Anexos

-  video1436593369.mp4
- [Enlace de GitHub](#)