Tecnicatura Universitaria en Programación

Algoritmos de Ordenamiento y Búsqueda en Python

Programación I

Autores: Nicolás Marcelo Copertino
José Carlos Costa
Profesor: Ariel Enferrel
Tutor: Luciano Chiroli

20 de junio de 2025

Índice

1. Introducción

En el campo de la informática, los algoritmos desempeñan un papel esencial para la resolución eficiente de problemas. Particularmente, los algoritmos de ordenamiento y búsqueda son herramientas clave cuando se trata de organizar, analizar y recuperar datos. Su correcta implementación no solo mejora el rendimiento de las aplicaciones, sino que también permite un manejo más claro y estructurado de la información.

Los algoritmos de ordenamiento permiten reorganizar los datos según ciertos criterios, como orden ascendente o descendente. Esta reorganización es crucial en tareas como búsqueda eficiente, análisis estadístico, visualización y comparación de datos. Por otro lado, los algoritmos de búsqueda facilitan la localización de elementos dentro de estructuras de datos, algo fundamental en aplicaciones que requieren respuestas rápidas y precisas.

El presente trabajo tiene como objetivo aplicar y analizar dos de los algoritmos más representativos en estas categorías: el algoritmo de ordenamiento por burbuja (Bubble Sort) y la búsqueda lineal. Para ello, se desarrolló un programa en Python que simula un caso práctico: el usuario ingresa los años de nacimiento de un grupo de personas, los cuales se validan, se ordenan utilizando Bubble Sort y luego se analiza cuáles de esos años corresponden a años bisiestos mediante búsqueda lineal.

Este enfoque no solo permite observar la utilidad de ambos algoritmos en un escenario realista, sino también evaluar sus ventajas, desventajas y comportamientos en distintas condiciones. A lo largo del trabajo se mostrarán explicaciones teóricas, ejemplos prácticos y conclusiones sobre su aplicación.

2. Algoritmos de Ordenamiento

Importancia de los algoritmos de ordenamiento

En informática, los algoritmos de ordenamiento son fundamentales para optimizar diversas tareas. Su principal función es organizar los datos de manera que puedan ser accedidos y utilizados con mayor eficiencia.

Ordenar datos no solo mejora el rendimiento general de los programas, sino que también permite resolver una amplia variedad de problemas de forma más rápida y eficaz:

- **Búsqueda:** Encontrar un elemento específico en una lista resulta mucho más rápido si los datos están ordenados. Algoritmos como la búsqueda binaria dependen de este orden para funcionar correctamente.
- **Selección:** Determinar elementos según su posición relativa, como el k-ésimo valor (mayor o menor) o la mediana, es mucho más sencillo cuando la lista está organizada.
- **Detección de duplicados:** En listas ordenadas, los valores repetidos aparecen uno junto al otro, lo que permite identificarlos de forma más eficiente.
- **Análisis de distribución:** Calcular la frecuencia con la que se repiten ciertos valores, o detectar los elementos que aparecen con mayor o menor frecuencia, se facilita considerablemente cuando los datos están ordenados.

Desde aplicaciones comerciales hasta investigaciones científicas, la ordenación de datos es una herramienta clave para reducir tiempos de procesamiento y mejorar el uso de recursos, convirtiéndose en una técnica esencial en múltiples áreas de la informática.

A continuación veremos ejemplos en dos de los algoritmos de ordenamiento más conocidos en la programación, el **algoritmo de ordenamiento de burbuja (Bubble Sort)** y el **algoritmo de Ordenamiento por inserción (Insertion Sort)**.

2.1. Algoritmo de ordenamiento de burbuja (Bubble Sort)

El algoritmo de ordenamiento por burbuja es uno de los más simples pero menos eficientes. Funciona comparando pares de elementos e intercambiándolos si están en el orden incorrecto, este proceso se hace una y otra vez hasta que la lista esté ordenada de forma correcta. A continuación veremos dos ejemplos de código:

```
def bubble_sort(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista

print(bubble_sort([5, 2, 4, 3, 1]))
```

Esta función se llama *bubble sort* y sirve para ordenar una lista de números de menor a mayor. Lo que hace es comparar de a pares, dos elementos que están uno al lado del otro. Si el primero es más grande que el segundo, los cambia de lugar. Este proceso se repite muchas veces. En cada vuelta completa, el número más grande "sube"hacia el final de la lista, como si flotara, por eso se llama "burbuja". Al final, devuelve la lista ordenada.

```
def selection_sort(lista):
    for i in range(len(lista)):
        min_idx = i
        for j in range(i+1, len(lista)):
            if lista[j] < lista[min_idx]:
                min_idx = j
        lista[i], lista[min_idx] = lista[min_idx], lista[i]
    return lista

print(selection_sort([29, 10, 14, 37, 13]))
```

La función *selection sort*, y también sirve para ordenar una lista. La idea es ir buscando el número más chico en el resto de la lista, y cuando lo encuentra, lo cambia de lugar con el que está en la posición actual. Va dejando los más chicos al principio, y avanzando de a uno.

Ambas tienen **complejidad** $O(n^2)$ por lo que no son muy eficientes si la lista es muy grande.

2.2. Ventajas y Desventajas del Ordenamiento de Burbuja

Ventajas:

- Es fácil de entender e implementar.
- Se escribe con poco código y sin estructuras complicadas.

Desventajas:

- Es lento para listas grandes.
- No aprovecha si la lista ya está casi ordenada.

2.3. Ordenamiento por inserción (Insertion Sort)

El algoritmo de ordenamiento por inserción es un algoritmo simple pero eficiente. Funciona dividiendo la lista en dos partes, una parte ordenada y otra desordenada, a medida que se recorre la lista desordenada, se insertan elementos en la posición correcta en la parte ordenada. A continuación veremos un ejemplo de código:

```
def insertion_sort(lista):
    for i in range(1, len(lista)):
        clave = lista[i]
        j = i - 1
        while j >= 0 and clave < lista[j]:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = clave
    return lista

print(insertion_sort([12, 11, 13, 5, 6]))
```

Este código sirve para ordenar una lista de números, usando el método de inserción. Lo que hace es agarrar un número y ver dónde encaja en los que ya están ordenados. Si el número es más chico que los anteriores, los corre para hacerle lugar. Cuando encuentra el lugar correcto, lo pone ahí. Así lo hace con todos, uno por uno, hasta que la lista queda ordenada.

En este caso la **complejidad** es $O(n)$ en el mejor caso y $O(n^2)$ en el peor caso.

2.4. Ventajas y Desventajas del Ordenamiento por Inserción

Ventajas:

- **Baja sobrecarga:** Requiere menos comparaciones y movimientos que otros algoritmos como el de burbuja, lo que lo hace más eficiente en cuanto a intercambios.
- **Simplicidad:** Es uno de los algoritmos más simples de implementar y entender. Ideal para enseñar conceptos básicos de ordenamiento.

Desventajas:

- **Ineficiencia en listas grandes:** Su rendimiento disminuye a medida que crece la lista. Su complejidad $O(n)$ en el peor caso lo vuelve ineficiente para grandes volúmenes de datos.
- **No escalable:** Como otros algoritmos cuadráticos, no se adapta bien a listas grandes, ya que el tiempo de ejecución aumenta mucho con el tamaño.

3. Algoritmos de Búsqueda

Importancia de los algoritmos de Búsqueda

Los algoritmos de búsqueda son métodos que nos permiten encontrar la ubicación de un elemento específico dentro de una lista de elementos. Dependiendo de la lista se necesita utilizar un algoritmo u otro, por ejemplo si la lista tiene elementos ordenados, se puede usar un algoritmo de búsqueda binaria, pero si la lista contiene los elementos de forma desordenada este algoritmo no te servirá, para búscar un elemento en una lista desordenada deberá utilizar un algoritmo de búsqueda lineal. Estos algoritmos son dos de los más relevantes y conocidos en la programación, a continuación veremos ejemplos de estos dos algoritmos.

3.1. Búsqueda Lineal

Los algoritmos de búsqueda lineal, también conocidos como búsqueda secuencial, implican recorrer una lista de elementos uno por uno hasta encontrar un elemento específico. Este algoritmo es muy sencillo de implementar en código pero puede ser muy ineficiente dependiendo del largo de la lista y la ubicación donde está el elemento. A continuación veremos un pequeño ejemplo de código en Python.

```python
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

# Lista de nmeros
numeros = [4, 7, 10, 15, 18, 23, 29, 34, 40, 45]

# Nmero que quiero buscar
numero_a_buscar = 23

# Llamamos a la funcin de bsqueda
posicion = busqueda_lineal(numeros, numero_a_buscar)

# Mostramos el resultado
if posicion != -1:
    print(f"El nmero {numero_a_buscar} se encuentra en la posicin {posicion}.")
else:
    print(f"El nmero {numero_a_buscar} no se encuentra en la lista.")
```

Lo que hacemos es definir una función que recibe una lista de números y el número que queremos buscar. La función recorre la lista desde el principio hasta el final, comparando cada elemento con el número que estamos buscando. Si lo encuentra, devuelve la posición donde está ese número. Si no lo encuentra, devuelve -1. En este caso, usamos una lista con varios números y queremos buscar el número 23. Cuando ejecutamos el código, si el número se encuentra, se muestra en qué posición está dentro de la lista. Y si no se encuentra, se muestra un mensaje avisando que el número no está. Este algoritmo es muy sencillo y útil, sobre todo cuando la lista no está ordenada, pero puede ser lento si la lista es muy larga, porque compara uno por uno. Por eso su **Complejidad** es $O(n)$, ya que en el peor caso tiene que revisar todos los elementos.

3.2. Ventajas y Desventajas del Algoritmo de Búsqueda Lineal

Ventajas

- **Sencillez:** La búsqueda lineal es uno de los algoritmos de búsqueda más simples y fáciles de implementar. Solo requiere iterar a través de la lista de elementos uno por uno hasta encontrar el objetivo.
- **Flexibilidad:** La búsqueda lineal puede aplicarse a cualquier tipo de lista, independientemente de si está ordenada o no.

Desventajas

- **Ineficiencia en listas grandes:** La principal desventaja de la búsqueda lineal es su ineficiencia en listas grandes. Debido a que compara cada elemento uno por uno, su tiempo de ejecución crece de manera lineal con el tamaño de la lista.

- **No es adecuada para listas ordenadas:** Aunque puede funcionar en listas no ordenadas, la búsqueda lineal no es eficiente para listas ordenadas. En tales casos, algoritmos de búsqueda más eficientes, como la búsqueda binaria, son preferibles.

3.3. Búsqueda Binaria

El algoritmo de búsqueda binaria es un algoritmo muy eficiente que se aplica solo a listas ordenadas. Funciona dividiendo repetidamente la lista en dos mitades y comparando el elemento objetivo con el elemento del medio, esto reduce significativamente la cantidad de comparaciones necesarias. A continuación veremos un pequeño ejemplo de búsqueda binaria con Python.

```python
def binary_search(lista, objetivo, inicio, fin ):
    if inicio > fin:
        return -1

    centro = (inicio + fin) // 2
    if lista[centro] == objetivo:
        return centro
    elif lista[centro] < objetivo:
        return binary_search(lista, objetivo, centro + 1, fin)
    else:
        return binary_search(lista, objetivo, inicio, centro - 1)

# Ejemplo de uso
lista = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 15, 20, 27, 34, 39, 50]
numero_objetivo = 27
inicio_busqueda = 0
fin_busqueda = len(lista) - 1

resultado = binary_search(lista, numero_objetivo, inicio_busqueda, fin_busqueda)

if resultado != -1:
    print(f"El nmero {numero_objetivo} se encuentra en la posicin {resultado}.")
else:
    print(f"El nmeor {numero_objetivo} NO se encuentra en la lista.")
```

En este ejemplo usamos una función recursiva. Lo que hace es buscar el número deseado dividiendo la lista a la mitad en cada paso. Primero calcula el índice central, y compara el valor en esa posición con el número que queremos encontrar. Si lo encuentra, devuelve la posición. Si el número buscado es mayor que el del centro, la función se llama a sí misma para buscar en la mitad derecha de la lista. Y si es menor, busca en la mitad izquierda. En este caso, buscamos el número 27 en una lista ordenada. Como la lista tiene 16 elementos, la búsqueda se hace en solo 4 pasos, mucho más rápido que revisar uno por uno. Si el número está, muestra la posición. Si no está, devuelve -1 y muestra que no se encuentra en la lista. La principal ventaja de este algoritmo es su eficiencia: su **complejidad** es $O(logn)$, porque en cada paso descarta la mitad de los elementos.

3.4. Ventajas y Desventajas del Algoritmo de Búsqueda Binaria

Ventajas

- **Eficiencia en listas ordenadas:** La principal ventaja de la búsqueda binaria es su eficiencia en listas ordenadas. Su tiempo de ejecución es de $\mathcal{O}(\log n)$, lo que significa que disminuye rápidamente a medida que el tamaño de la lista aumenta.
- **Menos comparaciones:** Comparado con la búsqueda lineal, la búsqueda binaria realiza menos comparaciones en promedio, lo que la hace más rápida para encontrar el objetivo.

Desventajas

- **Requiere una lista ordenada:** La búsqueda binaria sólo es aplicable a listas ordenadas. Si la lista no está ordenada, se debe realizar una operación adicional para ordenarla antes de poder utilizarla.
- **Mayor complejidad de implementación:** Comparada con la búsqueda lineal, la búsqueda binaria es más compleja de implementar debido a su lógica condicional y, en algunos casos, su uso de recursividad.

4. Comparación de Algoritmos

Algoritmo	**Tipo**	**Mejor caso**	**Peor caso**	**Lista ordenada**
Bubble Sort	Ordenamiento	$O(n)$	$O(n^2)$	No
Selection Sort	Ordenamiento	$O(n^2)$	$O(n^2)$	No
Insertion Sort	Ordenamiento	$O(n)$	$O(n^2)$	No
Búsqueda Lineal	Búsqueda	$O(1)$	$O(n)$	No
Búsqueda Binaria	Búsqueda	$O(1)$	$O(\log n)$	Sí

5. Caso Practico

Ordenamiento y Verificación de Años Bisiestos

Desarrollamos un programa en Python que permite al usuario ingresar información sobre una cantidad determinada de personas. El objetivo es aplicar conocimientos sobre algoritmos de ordenamiento (burbuja) y algoritmos de búsqueda (lineal), junto con validaciones de entrada

Requisitos del programa

Ingreso de datos

- Solicitar al usuario la cantidad de personas.

- Luego, pedir el año de nacimiento de cada persona.
- Validar que cada año ingresado sea un número válido de cuatro dígitos.

Ordenamiento de datos

- Usar el algoritmo de burbuja (*Bubble Sort*) para ordenar la lista de años de nacimiento de menor a mayor.

Análisis de años bisiestos

- Verificar, utilizando una búsqueda lineal, cuáles de los años ingresados son bisiestos.
- Mostrar todos los años que cumplen esta condición, o informar si ninguno lo es.

```
def es_bisiesto(anio):
    """Devuelve True si el ao es bisiesto, False si no lo es"""
    return (anio % 4 == 0 and anio % 100 != 0) or (anio % 400 == 0)

def ordenar_burbuja(lista):
    """Ordena la lista usando el algoritmo de burbuja"""
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista

# 1. Solicitar cantidad de personas
while True:
    try:
        cantidad = int(input("Ingrese la cantidad de personas: "))
        if cantidad > 0:
            break
        else:
            print("Debe ingresar un nmero mayor a 0.")
    except ValueError:
        print("Entrada invlida. Ingrese un nmero entero.")

# 2. Pedir aos de nacimiento con validacin
anios = []
for i in range(cantidad):
    while True:
        anio = input(f"Ingrese el ao de nacimiento de la persona {i+1}: ")
        if anio.isdigit() and len(anio) == 4:
            anios.append(int(anio))
            break
        else:
            print("Ao invlido. Debe ser un nmero de 4 dgitos.")

# 3. Ordenar usando burbuja
```

```python
anios_ordenados = ordenar_burbuja(anios.copy())
print("Aos ordenados de menor a mayor:", anios_ordenados)

# 4. Mostrar cules aos son bisiestos
bisiestos = [anio for anio in anios_ordenados if es_bisiesto(anio)]

print("\nVerificando aos bisiestos:")
if bisiestos:
    print("Los siguientes aos son bisiestos:", bisiestos)
else:
    print("Ninguno de los aos ingresados es bisiesto.")
```

6. Metodología Utilizada

Para el desarrollo del trabajo práctico se siguieron una serie de pasos estructurados que permitieron abordar de manera ordenada la problemática planteada, desde la investigación teórica hasta la implementación del código en Python. A continuación se detallan las etapas principales:

Investigación previa

Antes de comenzar con la implementación del código, se realizó una búsqueda bibliográfica y consulta de fuentes confiables.

Diseño y prueba del código

Se diseñaron pequeñas funciones en Python para implementar cada uno de los algoritmos estudiados. Estas funciones fueron probadas inicialmente con listas simples y conocidas, con el fin de verificar su correcto funcionamiento y validar los resultados obtenidos. Luego, se integraron en un caso práctico.

Herramientas y recursos utilizados

- **Lenguaje:** Python 3 por su claridad sintáctica y facilidad de aprendizaje.
- **IDE:** Se utilizó principalmente Visual Studio Code, ya que permite una organización clara del código, resaltado de sintaxis y acceso rápido a la terminal para pruebas.

7. Conclusión

En el mundo de la programación, los algoritmos de ordenamiento y búsqueda son fundamentales para para la manipulación y búsqueda de datos, los algoritmos de ordenamiento nos permiten organizar conjuntos de datos de forma ascendente o descendente mientras que los

algoritmos de búsqueda nos permiten localizar información de manera más rápida dependiendo de la situación. Comprender estos algoritmos es esencial para optimizar el rendimiento de programas y fortalecer la capacidad de resolución de problemas en programación. A través del lenguaje Python, es posible visualizar de manera sencilla su funcionamiento y aplicación.

8. Bibliografía

- Python Docs: `https://docs.python.org/`
- GeeksforGeeks: `https://www.geeksforgeeks.org/`
- Real Python: `https://realpython.com/`