Búsqueda y Ordenamiento en Programación

Alumnos:
Franco Luciano Berro - francoberro99@gmail.com
Akier Aguirrezabala - akieraguirrezabala@gmail.com

Materia: Programación I

Profesor: Cinthia Rigoni

Fecha de entrega: 9 de Junio 2025

Índice

1. Introducción:

El manejo eficiente de información es un aspecto crucial en programación. Entre las herramientas más utilizadas para ello se encuentran los algoritmos de búsqueda y ordenamiento. Este trabajo aborda dichas estrategias, enfocándose en su implementación con Python y su impacto en el rendimiento de los programas. El objetivo principal es comprender el funcionamiento de estos algoritmos y analizar en qué contextos resulta ventajoso aplicar uno u otro, mediante su implementación práctica.

2. Marco Teórico:

La búsqueda y el ordenamiento son operaciones esenciales en programación, necesarias para la manipulación eficiente de conjuntos de datos.

- *BÚSQUEDA*

La búsqueda es el proceso de localizar un elemento específico dentro de una colección de datos.

- *Tipos de algoritmos de búsqueda:*

1. Búsqueda lineal: Recorre cada elemento de la lista secuencialmente hasta encontrar el valor deseado o llegar al final. Es simple de implementar y no requiere que los datos estén ordenados, pero es ineficiente en listas grandes

2. Búsqueda binaria: Sólo se aplica a listas previamente ordenadas. Divide la lista en mitades y compara el valor medio con el objetivo. Repite el proceso en la mitad correspondiente. Es mucho más eficiente que la búsqueda lineal, pero requiere orden previo.

3. Búsqueda de interpolación: Mejora la búsqueda binaria al estimar la posición del valor objetivo en función de su magnitud. Es útil para listas grandes con distribución uniforme. Su eficiencia puede superar a la binaria en estos casos.

4. Búsqueda por hash: Utiliza una función hash que asocia claves a posiciones únicas en una tabla. Permite acceso directo en tiempo constante promedio, siendo ideal para grandes volúmenes de datos.

Ventajas y desventajas:

- La búsqueda lineal es versátil, pero lenta.
- La binaria es rápida, pero depende del orden.
- La interpolación requiere conocimiento sobre la distribución de los datos.

- La búsqueda hash es muy rápida, pero su implementación es un poco más compleja.

Ejemplos de uso:

- Buscar una palabra clave en un documento.
- Buscar un archivo en un sistema.
- Encontrar un registro en una base de datos.
- Buscar la ruta más corta en un grafo.
- Resolver problemas de optimización.

El tamaño de la lista influye directamente en el rendimiento. Por ejemplo, en listas de 100,000 elementos, la búsqueda binaria tarda unas 16 comparaciones, mientras que la lineal puede tardar 100,000.

ORDENAMIENTO

El ordenamiento organiza los datos según un criterio: numérico, alfabético, etc. Es una operación clave que facilita búsquedas eficientes y análisis de datos.

Principales algoritmos de ordenamiento:

1. Ordenamiento por burbuja (Bubble Sort): Compara elementos adyacentes e intercambia si están desordenados. Es fácil de implementar, pero muy ineficiente en listas grandes

2. Ordenamiento por selección (Selection Sort): Encuentra el elemento más pequeño y lo ubica en su lugar. Repite hasta ordenar la lista.

3. Ordenamiento por inserción (Insertion Sort): Inserta cada elemento en su lugar correcto. Es eficiente para listas pequeñas o casi ordenadas.

4. Quick Sort: Divide la lista en dos sublistas, menores y mayores que un pivote, y las ordena recursivamente. Es muy eficiente.

5. Merge Sort: divide en mitades, ordena y luego junta. Es estable y también muy eficaz.

Importancia de ordenar los datos:

- Permite aplicar búsqueda binaria (más eficiente que la lineal).
- Facilita el análisis de tendencias y patrones.
-Mejora el rendimiento de muchas operaciones como eliminación de duplicados o combinación de listas.

La elección del algoritmo depende del tamaño de los datos, si la lista está ordenada parcialmente, y los requerimientos de eficiencia.

Tabla de Complejidad

Algoritmo	Mejor Caso	Peor Caso	Promedio
Lineal Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
QuickSort	O(n log n)	$O(n^2)$	O(n log n)

3. Caso Práctico:

Se desarrolló un script en Python que permite ordenar una lista de números utilizando el método de Burbuja y buscar un número mediante búsqueda binaria.

```
class ordenamiento_burbuja:

    @staticmethod

    def ordenar(arr):
```

```
        n = len(arr)
        for i in range(n):
            for j in range(0, n - i - 1):
                if arr[j] > arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
class busqueda_binaria:
    @staticmethod
    def buscar_todos(arr, target):
        posiciones = []
        low = 0
        high = len(arr) - 1
        encontrado = False
        while low <= high and not encontrado:
            mid = (low + high) // 2
            if arr[mid] == target:
                i = mid
                while i >= 0 and arr[i] == target:
                    i -= 1
                i += 1
                while i < len(arr) and arr[i] == target:
                    posiciones.append(i)
```

```
                i += 1
            encontrado = True
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return posiciones
datos = [3, 19, 1, 2, 87, 3]
print ("Arreglo Original:", datos)
ordenamiento_burbuja.ordenar(datos)
print ("Arreglo Ordenado:", datos)
valor = int(input("Ingresa el valor que deseas buscar: "))
posiciones = busqueda_binaria.buscar_todos(datos, valor)
if posiciones:
    print("El valor: ", valor , " , encontrado en las posiciones:", posiciones)
else:
    print("No se encontró el valor solicitado.")
```

4. Metodología Utilizada

- Se investigaron fuentes teóricas oficiales y académicas.
- Se diseñaron e implementaron algoritmos en Python.

- Se probaron los algoritmos con diferentes entradas.
- Se analizaron los resultados y su rendimiento.
- Se documentó el proceso y se realizó el video explicativo correspondiente.

5. **Resultados Obtenido:**

- Se logró ordenar correctamente listas no ordenadas
- La búsqueda binaria funcionó con alta eficiencia al aplicar previamente ordenamiento.
- Validación: el código se ejecutó sin errores y se probó con múltiples listas.

6. **Conclusiones:**

Este trabajo permitió afianzar el conocimiento sobre algoritmos fundamentales en programación. Se comprobó que la elección del algoritmo adecuado puede optimizar significativamente los tiempos de respuesta. La búsqueda binaria se destacó, pero requiere una lista ordenada, lo cual implica evaluar previamente un método de ordenamiento eficiente. En resumen, esta mezcla entre ordenamiento y búsqueda es clave para la escalabilidad de los sistemas.

7. **Bibliografía:**

TU Programación UTN (2025) PDF: “Búsqueda y Ordenamiento en Programación”

TU Programación UTN (2025) Video Ordenamiento: https://www.youtube.com/watch?v=xntUhrhtLaw

TU Programación UTN (2025) Video Búsqueda: https://www.youtube.com/watch?v=gJlQTq80llg

8. **Anexos:**

Video explicativo: https://www.youtube.com/watch?v=DnIwpN1AthU

Repositorio en Github:

https://github.com/Apyrrr/TP-Integrador-Programaci-n-Aguirrezabala-y-Berro

PowerPoint usado en el video explicativo: (subido al repositorio de Github)