Algoritmos de Búsqueda y Ordenamiento

Alumnos:
Franco Enzo Petrozzelli – francopetrozzelli@gmail.com
Facundo Bistevins – facundobistevins@gmail.com

Materia: Programación I

Profesor: Sebastián Bruselario

Fecha de Entrega: 9 de junio de 2025

Índice

1. Introducción

Los algoritmos de búsqueda y ordenamiento son esenciales en la programación, siendo el corazón de la gestión de datos en casi cualquier sistema informático. Su eficiencia es directamente proporcional al rendimiento de las aplicaciones que utilizamos a diario. Este Trabajo Integrador de Programación I se centra en el estudio y la implementación práctica de estos algoritmos en Python. Nuestro objetivo es comprender su funcionamiento, analizar su eficiencia y consolidar los conocimientos teóricos con el desarrollo de una aplicación funcional. El informe se organiza de la siguiente manera: el **Marco Teórico** cubre los fundamentos. El **Caso Práctico** detalla la implementación de algoritmos de ordenamiento (Bubble y Quick Sort) y búsqueda (Lineal y Binaria) en listas de números y objetos (`Persona`), incluyendo la medición de su rendimiento. La **Metodología** describe el proceso de desarrollo, los **Resultados Obtenidos** presentan las mediciones, y las **Conclusiones** resumen los aprendizajes clave.

2. Marco Teórico

2.1.1 Bubble Sort (Ordenamiento de Burbuja) Este método de ordenación procede comparando pares de elementos vecinos y ajustando su posición si el orden es incorrecto. La operación se reitera hasta que cada elemento alcanza su lugar definitivo, culminando en una lista ordenada. Si bien su simplicidad lo hace fácil de comprender, su rendimiento es considerablemente limitado en colecciones de gran tamaño, caracterizado por una complejidad algorítmica de O(n^2).

2.1.2 Quick Sort (Ordenamiento Rápido) Quicksort se erige como una estrategia de ordenación de alto rendimiento, fundamentada en el principio de "divide y vencerás". Su mecanismo implica la selección de un elemento de referencia (el "pivote") para, posteriormente, segmentar la lista en dos subconjuntos: uno conteniendo valores inferiores al pivote y otro con valores superiores. El proceso se aplica de forma recursiva a estas subdivisiones hasta lograr la ordenación completa. Su destacada eficiencia lo convierte en una opción predilecta para el procesamiento de volúmenes extensos de información, exhibiendo consistentemente una complejidad algorítmica de O(n log n).

2.2 Algoritmos de Búsqueda Los algoritmos de búsqueda tienen como objetivo localizar un elemento específico dentro de una estructura de datos.

2.2.1 Búsqueda Lineal (Sequential Search)
-Concepto: Es el método de búsqueda más simple. Funciona recorriendo secuencialmente cada elemento de la lista hasta encontrar el valor deseado o hasta que se haya revisado toda la lista

-Principio de funcionamiento: Comienza desde el primer elemento y compara cada elemento con el valor buscado. Si los valores coinciden, devuelve la posición del elemento. Si se llega al final de la lista sin encontrar el elemento, se concluye que no está presente.

-Complejidad Temporal:
- -Mejor Caso: O(1) (el elemento es el primero de la lista).
- -Caso Promedio: O(n)
- -Peor Caso: O(n) (el elemento no está en la lista o es el último).

-Ventajas: Simple de implementar, no requiere que la lista esté ordenada.
-Desventajas: Ineficiente para listas grandes.

2.2.2 Búsqueda Binaria (Binary Search)
-Concepto: Es un algoritmo de búsqueda mucho más eficiente que la búsqueda lineal, pero con una condición fundamental: la lista debe estar previamente ordenada. Utiliza la estrategia de "divide y vencerás" para reducir el espacio de búsqueda a la mitad en cada paso.

- Principio de funcionamiento: Se compara el valor buscado con el elemento central de la lista. Si coinciden, se encuentra el elemento. Si el valor buscado es menor, la búsqueda continúa solo en la mitad inferior de la lista. Si es mayor, la búsqueda se restringe a la mitad superior. Este proceso se repite hasta que el elemento sea encontrado o el espacio de búsqueda se agote.

-Complejidad Temporal:
- -Mejor Caso: O(1) (el elemento es el central de la lista).
- -Caso Promedio: O(log n)
- -Peor Caso: O(log n)

-Ventajas: Extremadamente eficiente para listas grandes (mucho más rápida que la búsqueda lineal).
-Desventajas: Requiere que la lista esté ordenada, lo cual puede implicar un costo adicional si la lista no lo está.

3. Caso Práctico

Algoritmos de ordenamiento:
-Lista usada:

```
numeros = [random.randint(0, 100) for _ in range(1000)]
print(numeros)
```

-Bubble Sort:

```
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]
```

compara de a pares elementos vecinos, los intercambia si están desordenados repitiendo este proceso muchas veces, con el fin de que la lista quede ordenada

-Quick Sort:

```
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivote = lista[0]
        menores = [x for x in lista[1:] if x <= pivote]
        mayores = [x for x in lista[1:] if x > pivote]
        return quick_sort(menores) + [pivote] + quick_sort(mayores)
```

elige un “pivote”, separa los que son menores o mayores y ordena las dos partes recursivamente

Algoritmos de búsqueda
-Búsqueda Lineal:

```
def busqueda_lineal(lista, objetivo):
    for i, valor in enumerate(lista):
        if valor == objetivo:
            return i
    return -1
```

Recorre la lista de a uno, al encontrar su valor devuelve su posición, si no devuelve “-1”

-Búsqueda Binaria:

```
def busqueda_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1
```

Después de ordenar la lista con sorted(numeros), busca dividiendo la lista en dos, es una forma mucho mas rapida que la búsqueda lineal

importando Time podemos medir cuánto tarda cada proceso

```
inicio = time.time()
bubble_sort(numeros.copy())
fin = time.time()
print(f"Bubble Sort: {fin - inicio:.6f} segundos")

inicio = time.time()
quick_sort(numeros.copy())
fin = time.time()
print(f"Quick Sort: {fin - inicio:.6f} segundos")
```

Con la función .copy() copiamos la lista original si que se modifique nada
Este sería un ejemplo de los resultados obtenidos usando diez números:

Bubble Sort: 0.000010 segundos
Quick Sort: 0.000005 segundos
Lista ordenada: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Búsqueda lineal de 7: 3
Búsqueda binaria de 4: 3

4. Metodología Utilizada

La elaboración del presente trabajo práctico se basó en un enfoque de **investigación, implementación y prueba**, estructurado en las siguientes fases:

- Investigación Preliminar: Se inició con una fase de investigación teórica sobre los algoritmos de ordenamiento (Bubble Sort, Quick Sort) y búsqueda (Lineal y Binaria), comprendiendo sus principios de funcionamiento y su complejidad algorítmica.

- Implementación en Python: Basándose en la investigación, se procedió a la implementación práctica de estos algoritmos utilizando el lenguaje de programación Python. El código se desarrolló de forma modular, buscando la claridad y replicabilidad de las lógicas algorítmicas.

- Prueba y Medición de Rendimiento: Se realizaron pruebas empíricas del código implementado. Esto incluyó la generación de listas de datos para cada algoritmo y la medición de su tiempo de ejecución, permitiendo una comparación práctica con las complejidades teóricas.

- Herramientas Utilizadas: Para el desarrollo y gestión del proyecto, se empleó **Visual Studio Code (VS Code)** como entorno de desarrollo integrado (IDE) y **GitHub** para el control de versiones y la colaboración.

5. Resultados Obtenidos

Tras la fase de pruebas y la observación de los resultados empíricos (cuyas capturas se adjuntan en el anexo digital), se derivaron las siguientes conclusiones sobre el comportamiento de los algoritmos de ordenamiento y búsqueda:

Algoritmos de Ordenamiento

La evaluación del rendimiento de los algoritmos de ordenamiento reveló una clara jerarquía de eficiencia:

- **Quick Sort** se consolidó como el método de ordenación de mayor rendimiento, demostrando una superioridad indiscutible en términos de velocidad.
- **Bubble Sort**, por otro lado, mostró ser práctico únicamente para conjuntos de datos de tamaño muy reducido o con propósitos didácticos, dada su limitada eficiencia.
- La disparidad en el rendimiento entre los algoritmos se acentúa de manera exponencial a medida que el volumen de los datos a procesar aumenta.

Algoritmos de Búsqueda

Las pruebas de búsqueda proporcionaron insights sobre la idoneidad de cada algoritmo según el estado de los datos:

- **Búsqueda Binaria** justificó plenamente la inversión inicial de ordenar la información, ya que su eficiencia y escalabilidad la posicionan como la opción predilecta para aplicaciones con grandes volúmenes de datos.
- **Búsqueda Lineal** mantiene su relevancia y utilidad en escenarios donde la lista de datos no se encuentra previamente ordenada, ofreciendo una solución directa aunque menos eficiente.

6. Conclusión

Este trabajo práctico nos permitió reafirmar la importancia fundamental de los algoritmos de ordenamiento y búsqueda en la ciencia de la computación, así como la crítica relevancia de su eficiencia. Mediante la implementación y experimentación empírica en Python, pudimos validar la teoría de la Complejidad Algorítmica (Notación Big O), observando cómo la elección del algoritmo impacta drásticamente el rendimiento, especialmente con el crecimiento del volumen de datos.

Confirmamos la superioridad de Quick Sort para tareas de ordenamiento en grandes conjuntos de datos (O(n log n)), contrastándola con la limitada aplicabilidad de Bubble Sort a escenarios reducidos. En cuanto a la búsqueda, la Búsqueda Binaria se mostró notablemente más eficiente (O(log n)), justificando el costo de un pre-ordenamiento de la lista, mientras que la Búsqueda Lineal (O(n)) mantuvo su utilidad sólo en ausencia de un orden previo.

7. Bibliografía / Referencias

- GeeksforGeeks. (2025, 17 de abril). Bubble Sort Algorithm. Recuperado de https://www.geeksforgeeks.org/bubble-sort-algorithm/
- -GeeksforGeeks. (2025, 17 de abril). Quick Sort Algorithm. Recuperado de https://www.geeksforgeeks.org/quick-sort-algorithm/
- Algoritmos de Ordenamiento y Búsqueda en Python: Optimizando la Gestión de Datos. (n.d.). 4Geeks. Recuperado de https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python

8. Bibliografía / Referencias

Enlace al repositorio: https://github.com/FacuBiste/TpProgramacion

Enlace al video explicativo: https://youtu.be/0zYnP_WdD8s?si=JOlba3BGVy0GzdUK&utm_source=MTQxZ