Algoritmos de Búsqueda y Ordenamiento en Python

Lucio Rivas – lucior0900@gmail.com

Máximo Ponce – maximoponce31@gmail.com

Materia: Programación I

Fecha de reentrega: 19/06

Índice

1. Introducción

Los algoritmos de búsqueda y ordenamiento son pilares fundamentales en la informática. Permiten organizar, acceder y manipular datos de manera eficiente. En este trabajo se realiza una investigación aplicada sobre cuatro algoritmos clásicos: Bubble Sort, Insertion Sort, Quick Sort y Binary Search, utilizando el lenguaje Python.

La elección de estos algoritmos permite no solo entender su lógica y funcionamiento, sino también reflexionar sobre su eficiencia, sus ventajas y limitaciones, y cómo aplicarlos en contextos reales.

2. Marco Teórico

Algoritmos de Ordenamiento

Bubble Sort

Algoritmo simple que compara pares adyacentes y los intercambia si están en el orden incorrecto. Es fácil de entender pero poco eficiente en listas grandes.

Complejidad: $O(n^2)$

Insertion Sort

Construye la lista ordenada uno por uno, insertando cada nuevo elemento en su posición correcta. Más eficiente que Bubble Sort en listas pequeñas.

Complejidad: $O(n^2)$

⚡Quick Sort

Algoritmo de ordenamiento rápido basado en dividir y conquistar. Elige un pivote y separa los elementos menores a la izquierda y los mayores a la derecha, repitiendo recursivamente.

Complejidad: O(n log n) en el mejor caso, $O(n^2)$ en el peor.

Algoritmo de Búsqueda

Binary Search

Busca un valor en una lista ordenada dividiéndola sucesivamente a la mitad. Es muy rápido, pero exige que los datos estén previamente ordenados.

Complejidad: O(log n)

3. Algoritmos Implementados

```
def bubble_sort(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]


def insertion_sort(arr):
    for i in range(1, len(arr)):
        valor = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > valor:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = valor


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


def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
```

```
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

4. Caso Práctico

Gestión de Catálogo de Productos

Simulamos un sistema de gestión de productos, donde el usuario carga una lista de precios desordenada, elige un método de ordenamiento y luego busca un producto por precio:

```
import time

productos = [4500, 1200, 7800, 400, 3300, 5600, 900]

print("Catálogo original:", productos)

# Ordenamiento
productos_bubble = productos.copy()
bubble_sort(productos_bubble)
print("Ordenado con Bubble Sort:", productos_bubble)
```

```
productos_insertion = productos.copy()

insertion_sort(productos_insertion)

print("Ordenado con Insertion Sort:", productos_insertion)


productos_quick = quick_sort(productos)

print("Ordenado con Quick Sort:", productos_quick)


# Búsqueda

precio = int(input("Ingrese el precio que desea buscar: "))

pos = binary_search(productos_quick, precio)

if pos != -1:

    print(f"El producto de ${precio} está en la posición {pos}.")

else:

    print(f"El producto de ${precio} no se encuentra.")
```

5. Comparación de Algoritmos

Algoritmo	Facilidad de Implementación	Eficiencia	Recomendado para
Bubble Sort	Alta	Baja	Aprendizaje inicial
Insertion Sort	Media	Media	Listas pequeñas o parcialmente ordenadas
Quick Sort	Baja	Alta	Grandes volúmenes de datos
Binary Search	Alta	Muy Alta	Búsqueda en listas ordenadas

Además, con listas grandes (>1000 elementos), Quick Sort mostró resultados visiblemente más rápidos que los otros dos algoritmos.

6. Conclusiones

Este trabajo permitió integrar conocimientos teóricos con la práctica real de la programación. Se aprendió a aplicar distintos algoritmos, entender su eficiencia, y reconocer cuándo conviene usar cada uno.

Los algoritmos simples como Bubble Sort son útiles para comenzar, pero no son eficientes. Por el contrario, Quick Sort y Binary Search muestran un rendimiento superior, y son la base de procesos reales que usan bases de datos y motores de búsqueda.

Comprender estos algoritmos permite programar de forma más profesional y resolver problemas de forma más eficiente.

7. Bibliografía

Python Software Foundation. (2024). Documentación oficial

Khan Academy. Algoritmos de ordenamiento y búsqueda

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). Introduction to Algorithms.

8. Anexos

Enlace al repositorio: https://github.com/RivasLucio/tp-integradot-utn

Enlace al video explicativo (reentrega):
https://www.youtube.com/watch?v=s17Bm172k8g&ab_channel=LucioR.