Título del trabajo: Algoritmos de Búsqueda y Ordenamiento en Python

Alumnos: Luca Giovanni Arlettaz - luca.arlettaz@tupad.utn.edu.ar

Materia: Programación I

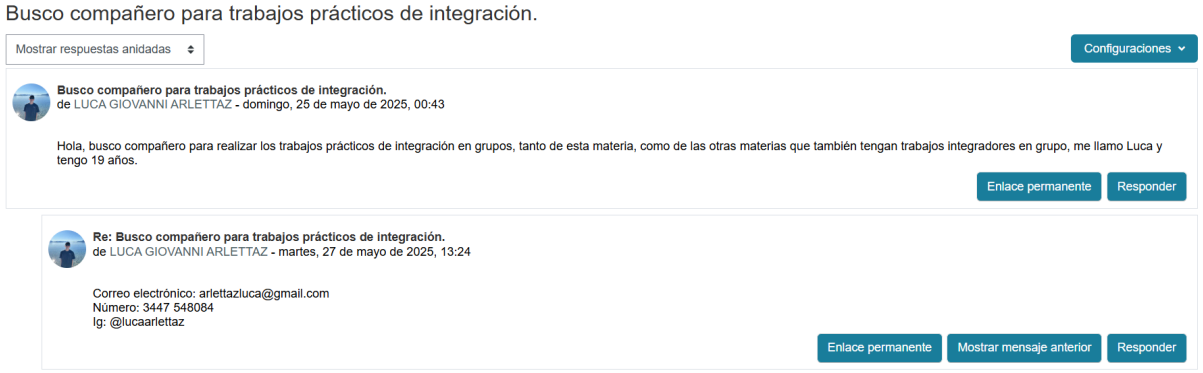
Profesor/a: Cinthia Rigoni

Fecha de Entrega: 09/06/2025

Índice

Indicaciones Importantes

Excepcion de trabajo en pareja: El trabajo fue realizado de manera individual ya que no logré tener contacto con ninguno de mis compañeros, aún comentando en foros.



Introducción

Este trabajo tiene como objetivo comprender el funcionamiento de diferentes algoritmos de búsqueda y ordenamiento, analizando su eficiencia y comportamiento.

La aplicación o uso de dichos algoritmos son fundamentales en el campo de la programación, permiten organizar y localizar información de forma eficiente, y son esenciales en múltiples áreas del desarrollo de software, como bases de datos, sistemas operativos, inteligencia artificial.

Se eligió este tema por su relevancia, importancia e impacto en el rendimiento de programas. Ya que nos será de gran utilidad aprender a fondo el funcionamiento de estos algoritmos, en qué casos conviene utilizar uno u otro.

Marco Teórico

Un algoritmo es un conjunto de pasos ordenados que permite resolver un problema o realizar una tarea específica. En programación, son la base sobre la que se construyen las soluciones lógicas.

Algoritmos de búsqueda

Los algoritmos de búsqueda permiten localizar un elemento dentro de una estructura de datos. Podemos clasificarlos en:

Búsqueda lineal: Recorre los elementos uno a uno hasta encontrar el valor buscado o llegar al final de la estructura. Es simple pero poco eficiente para grandes volúmenes de datos, a medida que el volumen de los datos aumenta se hace menos eficiente. Tiene una complejidad de tiempo O(n).

Búsqueda binaria: Sólo funciona en estructuras previamente ordenadas. Funciona dividiendo el conjunto a la mitad en cada iteración, descartando la mitad que no contiene el valor buscado. Su eficiencia es mucho mayor, con una complejidad de O(log n).

Algoritmos de ordenamiento

Los algoritmos de ordenamiento se utilizan para reorganizar los elementos de una estructura según un criterio determinado por ejemplo, de menor a mayor. Algunos son:

Bubble Sort: Compara elementos adyacentes y los intercambia si están en el orden incorrecto. Es fácil de implementar pero ineficiente en listas grandes. Caso promedio: $O(n^2)$

Selection Sort: Selecciona el elemento más pequeño o más grande según corresponda y lo coloca en su posición. Caso promedio: $O(n^2)$.

Insertion Sort: Inserta cada elemento en su lugar adecuado dentro de la parte ordenada de la lista. Tiene buen rendimiento con listas pequeñas o casi ordenadas. Caso promedio: $(O(n^2)$

Quick Sort: Está basado en “divide y vencerás”, elige un pivote y divide la lista según su valor. Su rendimiento promedio es O(n log n), en el peor caso puede llegar a $O(n^2)$.

En Python, los algoritmos de búsqueda y ordenamiento se pueden implementar utilizando estructuras como listas, ciclos for, condicionales if, funciones recursivas, y realizando mediciones de tiempo con time.

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

Python Software Foundation. (2024). *Python 3 Documentation*. https://docs.python.org/3/

Geeks for Geeks. (s.f.). *Data Structures and Algorithms*. https://www.geeksforgeeks.org/

Caso Práctico

Se desarrolló un programa en Python que permite comparar el uso de búsqueda lineal y búsqueda binaria partiendo de una lista de números desordenados.

En el primer caso, se utiliza la búsqueda lineal directamente sobre la lista desordenada.

En el segundo caso, se utiliza búsqueda binaria, pero como este algoritmo requiere que la lista esté ordenada, se aplica previamente un algoritmo de ordenamiento para que funcione correctamente.

Código Fuente:

```
#Comenzamos creando el Insertion Sort, necesario para que funcione la
búsqueda binaria mas adelante

def insertion_sort(lista):

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

        actual = lista[i]

        j = i - 1

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

            lista[j + 1] = lista[j]

            j -= 1

        lista[j + 1] = actual
```

```python
#Búsqueda Lineal, funciona directamente en listas desordenadas

def busqueda_lineal(lista, valor):

    for i in range(len(lista)):

        if lista[i] == valor:

            return i

    return -1



#Búsqueda Binaria, requiere que la lista esté ordenada previamente,
como lo hicimos con insertion sort

def busqueda_binaria(lista, valor):

    inicio = 0

    fin = len(lista) - 1

    while inicio <= fin:

        medio = (inicio + fin) // 2

        if lista[medio] == valor:

            return medio

        elif lista[medio] < valor:

            inicio = medio + 1

        else:

            fin = medio - 1

    return -1



#Lista desordenada

numeros = [17, 3, 25, 9, 30, 8, 41, 6, 22, 14, 12, 19, 5, 27, 11]
```

```
#Valor a buscar

objetivo = 12



#Aplicacion de búsqueda lineal, no hace falta ordenar la lista

print("Lista original:", numeros)

pos_lineal = busqueda_lineal(numeros, objetivo)

print(f"Busqueda Lineal: Elemento {objetivo} encontrado en la
posicion:", pos_lineal if pos_lineal != -1 else "Numero objetivo no
encontrado")



#Aplicacion de búsqueda binaria

#Primero ordenando la lista

lista_ordenada = numeros.copy()

insertion_sort(lista_ordenada)

print("Lista ordenada para búsqueda binaria:", lista_ordenada)

pos_binaria = busqueda_binaria(lista_ordenada, objetivo)

print(f"Busqueda Binaria: Elemento {objetivo} encontrado en la
posicion:", pos_binaria if pos_binaria != -1 else "Número objetivo no
encontrado")
```

Explicación de decisiones de diseño

Se utilizó una misma lista desordenada para ambos tipos de búsqueda, así nos permite comparar cómo se comportan en un entorno idéntico.

La búsqueda lineal se aplicó directamente, sin modificar la lista, mostrando su utilidad inmediata, aunque menos eficiente en listas grandes.

La búsqueda binaria se aplicó solo después de ordenar la lista, lo que introduce un costo adicional inicial, pero resulta más eficiente en búsquedas repetidas o listas muy grandes.

Se eligió Insertion Sort como método de ordenamiento previo a la búsqueda binaria debido a su simpleza y buen rendimiento en listas de tamaño pequeño o moderado. La búsqueda lineal, si bien menos eficiente, es ideal cuando la lista no está ordenada ya que no se invierte tiempo en ordenarla previamente.

Validación del funcionamiento

Se verificó que ambos algoritmos encuentran correctamente el valor buscado si está presente, o devuelven una respuesta adecuada si no lo está.

Se pueden realizar pruebas adicionales variando el tamaño de la lista o el valor buscado.

Metodología Utilizada

En primer lugar, se investigó sobre algoritmos de búsqueda y ordenamiento, consultando bibliografía básica de programación para comprender sus definiciones, funcionamiento y eficiencia, principalmente de material brindado por la universidad tecnológica nacional y documentación oficial.

Luego, se diseñó el caso práctico que permitió aplicar y comparar distintos algoritmos. Se utilizó la búsqueda lineal sobre una lista desordenada y búsqueda binaria acompañada de un algoritmo de ordenamiento (Insertion Sort), con el objetivo de observar diferencias de comportamiento.

Se validó el funcionamiento de los algoritmos, asegurando que respondan correctamente ante diferentes entradas, tanto si el valor se encuentra en la lista como si no.

Resultado Obtenidos

A partir del desarrollo del caso práctico, se logró implementar con éxito los algoritmos de búsqueda lineal y binaria, y el algoritmo de ordenamiento Insertion Sort, validando su correcto funcionamiento ante diferentes entradas. Ambos métodos de búsqueda localizaron

el valor objetivo de forma precisa cuando estaba presente en la lista y devolvieron resultados adecuados cuando no lo estaba.

La búsqueda lineal funciona correctamente sin necesidad de ordenar la lista, pero a medida que esta crece, se vuelve más lenta debido a que recorre cada elemento.

La búsqueda binaria, tras ordenar la lista previamente, demostró ser mucho más rápida en encontrar el elemento, especialmente en listas más extensas, aunque requiere el coste computacional inicial de ordenar.

No se observaron errores graves en la ejecución del código. Para listas pequeñas o medianas, el rendimiento es aceptable, pero obviamente si se incrementara la cantidad de datos, el tiempo de ejecución aumentaría y la eficiencia iría disminuyendo levemente, pero progresivamente en el caso de la búsqueda lineal.

https://github.com/LucaArlettaz/Trabajo-Integrador-Programacion-1

Conclusiones

Realizar este trabajo permitió profundizar en la comprensión de los algoritmos de búsqueda y ordenamiento, fundamentales en la programación. A través del caso práctico se logró observar el comportamiento real de la búsqueda lineal y la búsqueda binaria, destacando sus diferencias en términos de eficiencia y condiciones de aplicación. La búsqueda lineal resulta simple y efectiva, aún en listas desordenadas y de tamaño pequeño o moderado, mientras que la búsqueda binaria, aunque requiere que la lista esté ordenada previamente, ofrece un rendimiento superior en listas grandes.

Se utilizó el algoritmo de ordenamiento Insertion Sort para preparar la lista para la búsqueda binaria, lo que permitió entender la importancia de seleccionar un algoritmo de ordenamiento adecuado según el contexto, en este caso, uno eficiente para listas pequeñas.

Aunque en el trabajo no se implementaron algoritmos como Bubble Sort, Selection Sort o Quick Sort, su estudio teórico fue relevante para comprender estrategias y complejidades que existen, así como ventajas y limitaciones. Por ejemplo, Quick Sort es preferible en listas grandes por su eficiencia promedio O(n log n).

Comprender cómo optimizar procesos clave como la localización y organización eficiente de datos, nos convertirá en mejores programadores ya que resulta fundamental para el rendimiento y optimización de cualquier sistema.

Bibliografía

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

Python Software Foundation. (2024). *Documentación oficial*. https://docs.python.org/3/

Geeks for Geeks. (2024). *Data Structures and Algorithms - Searching and Sorting*. https://www.geeksforgeeks.org/fundamentals-of-algorithms/

W3Schools. (2024). *Python Algorithms*. https://www.w3schools.com/python/python_examples.asp

Anexos

Código fuente del caso práctico.

Repositorio de GitHub: https://github.com/LucaArlettaz/Trabajo-Integrador-Programacion-1