Algoritmos de Búsqueda y Ordenamiento en Python

Alumnos: Alejandro Saavedra, Juan Ignacio Rouge

Materia: Programación I

Profesor/a: Nicolas Quiros

Tutor/a: Neyen Bianchi

Fecha de Entrega: 09/06/2025

Correo: saavedrakjjk@gmail.com, juanir.ignacio23@gmail.com

Introducción

Cuando programamos, muchas veces tenemos que trabajar con listas de datos: buscarlos, ordenarlos, filtrarlos, etc. Para eso existen los algoritmos de búsqueda y ordenamiento, que básicamente nos ayudan a encontrar o acomodar información de forma más rápida y eficiente. Elegimos esté tema porque el orden y la organización son esenciales para trabajar de manera eficiente.

En Python se pueden usar funciones ya hechas, como sorted() o index(), pero está bueno entender cómo funcionan estos procesos por dentro. Saber eso te da una mejor base para programar y resolver problemas más complejos más adelante.

En este trabajo se explican los algoritmos de búsqueda más conocidos (como la búsqueda lineal y binaria) y algunos de los métodos de ordenamiento más usados (como burbuja, inserción y quicksort), con ejemplos simples en Python para ver cómo se aplican en la práctica.

Marco Teórico

Algoritmos de Búsqueda

Los algoritmos de búsqueda permiten encontrar un elemento dentro de una estructura de datos, como una lista. En Python, se usan principalmente dos tipos:

Búsqueda Lineal

También llamada búsqueda secuencial, recorre uno por uno los elementos hasta encontrar el valor buscado o llegar al final de la lista.

Ventajas: Fácil de implementar, Funciona en listas no ordenadas.

Desventajas: Ineficiente en listas grandes.

Por ejemplo:

```python
def busqueda_lineal(arr, objetivo):
    """Búsqueda lineal: recorre la lista uno por uno hasta encontrar el objetivo."""
    for i in range(len(arr)):
        if arr[i] == objetivo:
            return i  # Devuelve la posición
    return -1  # No encontrado
```

Búsqueda Binaria

Solo se puede aplicar a listas ordenadas. Divide la lista a la mitad repetidamente para reducir el espacio de búsqueda.

Ventajas: Mucho más rápida que la búsqueda lineal.

Desventajas: Requiere que la lista esté ordenada.

Por ejemplo:

```python
def busqueda_binaria(arr, objetivo):
    """Búsqueda binaria: busca dividiendo a la mitad, requiere lista ordenada."""
    izquierda = 0
    derecha = len(arr) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if arr[medio] == objetivo:
            return medio
        elif arr[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1  # No encontrado
```

Algoritmos de Ordenamiento

Los algoritmos de ordenamiento organizan los elementos de una lista en un orden específico, generalmente ascendente. Algunos de los más conocidos son:

Bubble Sort

Compara elementos adyacentes y los intercambia si están en el orden incorrecto. Se repite el proceso hasta que la lista esté ordenada.

Por ejemplo:

```python
def ordenamiento_burbuja(arr):
    """Ordenamiento burbuja: compara elementos adyacentes y los intercambia si están en el orden incorrecto."""
    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]
    return arr
```

Insertion Sort

Inserta cada elemento en la posición correcta respecto a los anteriores, como si se ordenaran cartas en la mano.

Ventajas: Mucho más rápida que la búsqueda lineal.

Desventajas: Requiere que la lista esté ordenada.

Por ejemplo:

```python
def ordenamiento_insercion(arr):
    """Ordenamiento por inserción: inserta cada elemento en su posición correcta dentro de una lista ordenada."""
    for i in range(1, len(arr)):
        clave = arr[i]
        j = i - 1
        while j >= 0 and clave < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = clave
    return arr
```

Quick Sort

Divide la lista en sublistas menores y mayores que un pivote, y las ordena recursivamente.

Por ejemplo:

```python
def quicksort(arr):
    """Quicksort: divide y conquista, elige un pivote y ordena los elementos menores y mayores a ese pivote."""
    if len(arr) <= 1:
        return arr
    else:
        pivote = arr[0]
        menores = [x for x in arr[1:] if x <= pivote]
        mayores = [x for x in arr[1:] if x > pivote]
        return quicksort(menores) + [pivote] + quicksort(mayores)
```

Caso Práctico

Ejemplo de caso práctico con Quick Sort:

```python
# Caso práctico
if __name__ == "__main__":
    print("=== Ordenamiento con Quicksort ===")

    # Ingreso simple de datos
    entrada = input("Ingresá números separados por comas: ")
    numeros = [int(x.strip()) for x in entrada.split(",")]

    ordenada = quicksort(numeros)

    print("Lista ordenada:", ordenada)
```

Salida esperada:

```
#Input: 5, 2, 9, 1, 7
#Salida esperada: [1, 2, 5, 7, 9]
```

Metodologia utilizada

Para este trabajo, primero se investigaron los conceptos básicos de los algoritmos de búsqueda y ordenamiento más comunes, entendiendo cómo funcionan y cuándo conviene usarlos. Se seleccionaron

algoritmos representativos como la búsqueda lineal, búsqueda binaria, bubble sort, insertion sort y quick sort.

Luego, se implementaron estos algoritmos en Python desde cero, priorizando un código claro y sencillo, para enfocarse en la lógica detrás de cada uno y evitar funciones ya hechas del lenguaje.

Finalmente, se probaron los algoritmos con ejemplos prácticos para ver cómo se comportan con distintos tipos y tamaños de datos, lo que ayudó a comprender su funcionamiento real y sus ventajas o limitaciones.

Resultados obtenidos

Después de implementar y probar los algoritmos en Python, se observó que:

Los algoritmos como bubble sort e insertion sort funcionan bien con listas pequeñas, pero su rendimiento baja bastante cuando la lista crece.

El quick sort es mucho más eficiente y rápido, especialmente para listas medianas y grandes.

La búsqueda lineal es sencilla y útil para listas desordenadas, aunque puede ser lenta con listas largas.

La búsqueda binaria es mucho más rápida, pero requiere que la lista esté ordenada previamente.

Conclusion

En este trabajo se pudo ver que entender y aplicar algoritmos de búsqueda y ordenamiento es fundamental para manejar datos de forma eficiente. Cada algoritmo tiene sus puntos fuertes y débiles, y elegir el correcto depende del tipo y tamaño de la lista.

Los algoritmos simples como bubble sort o búsqueda lineal sirven para casos básicos o listas pequeñas, pero no son tan prácticos para datos grandes. En cambio, quick sort y búsqueda binaria ofrecen un rendimiento mucho mejor en esos casos.

Por eso, saber cuándo usar cada uno es clave para mejorar el rendimiento de cualquier programa o análisis de datos.

Bibliografia

https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python

Apuntes y bibliografía de la UTN

Para encontrar bibliografía más compleja consultamos con Chat GPT y esto fue lo que usamos:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. (https://github.com/Zelechos/Introduccion_a_los_Algoritmos)

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in Python*. Wiley. (https://archive.org/details/data-structures-and-algorithms-in-python-goodrich/mode/2up)

Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in Python*. Pearson. (https://nibmehub.com/opac-service/pdf/read/Data%20Structures%20and%20Algorithms%20in%20Python.pdf)