Algoritmos de búsqueda y ordenamiento
Héctor Romero
Programación I
13/07/2025

Índice:

1. Introducción:

Este tema es algo importante en la implementación de sistemas dado que constantemente estamos manejado datos, y trabajando sobre ellos. Algo fundamental para que los datos dejen de ser "datos" y pasen a ser información.
Los algoritmos de ordenamiento son técnicas fundamentales en la informática que permiten reorganizar los datos de una estructura, generalmente listas o arreglos, en un orden específico (ascendente o descendente). Ordenar datos facilita la búsqueda, análisis y procesamiento eficiente de la información (Cormen et al., 2009).

En este trabajo se presentan los principales algoritmos de ordenamiento, se comparan sus eficiencias y se realiza una implementación en Python con visualización paso a paso para comprender su funcionamiento (Cormen et al., 2009).

Intentamos explorar estos algoritmos, comprenderlos e implementarlos dado que esto ultimo es la forma de entender su funcionamiento.

2. Marco Teórico:

¿Qué es un algoritmo de ordenamiento?
Es un conjunto de instrucciones que recibe una cantidad determinada de datos y los reorganiza según un criterio definido, como el orden numérico ascendente o descendente (Cormen et al., 2009).

Esto nos lleva a distintos criterios para determinar el proceso. Dentro de los más comunes podemos hablar de:

Bubble Sort (Ordenamiento Burbuja): Compara elementos adyacentes y los intercambia si están en orden incorrecto; se repite hasta que la lista esté ordenada. Es simple pero poco eficiente para grandes volúmenes de datos (Cormen et al., 2009).

Insertion Sort (Ordenamiento por inserción): Construye el arreglo ordenado insertando cada elemento en su posición correcta dentro de la parte ordenada del arreglo.

Merge Sort (Ordenamiento por Mezcla): Divide el arreglo en mitades, ordena cada mitad y las combina. Divide y vencerás (Cormen et al., 2009).

Quick Sort (Ordenamiento Rápido): Selecciona un pivote y divide el arreglo en sub arreglos menores y mayores al pivote, luego ordena recursivamente. Rendimiento promedio muy bueno. Luego, tenemos los algoritmos de búsqueda que tienen como objetivo encontrar un valor dentro de la colección de datos. De estos destacamos dos casos:

Búsqueda Lineal: Recorre secuencialmente todos los elementos hasta encontrar el solicitado.

Búsqueda Binaria: Divide el espacio de búsqueda en mitades sucesivas. Comparando el valor buscado con el elemento central. Si es menos se sigue con la mitad izquierda si es mayor con la mitad derecha.

3. Caso Práctico:

Se implementará el algoritmo Quick Sort para el ordenamiento ascendente de una determinada cantidad de números. Se elije este método teniendo en cuenta su tiempo de ejecución. Además, se desarrolló una visualización gráfica para entender el proceso de ordenamiento.
Se implementará por otro lado, un algoritmo de búsqueda binaria. También teniendo en cuenta su tiempo de ejecución.

Código de ejemplo:

```
import matplotlib.pyplot as plt             #visualización de datos
import random                               #crear listas aleatorias para ordenar
import time                                 #medir el tiempo de ejecución

# ------------------- Quick Sort ( -------------------
def quick_sort(arr, low=0, high=None):  # Ordena el arreglo usando el algoritmo Quick Sort
    if high is None:                # Si high no se ha definido, lo establece al último índice del arreglo
        high = len(arr) - 1
    if low < high:
        pi = partition(arr, low, high)          # Llama a la función de partición y obtiene el índice del
pivote
        visualize(arr, low, high, pi)         # Visualiza el proceso de ordenamiento
        quick_sort(arr, low, pi - 1)          # Llama recursivamente a quick_sort para las dos mitades
del arreglo
        quick_sort(arr, pi + 1, high)

def partition(arr, low, high):          # Divide el arreglo en dos partes y devuelve el índice del pivote
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
        visualize(arr, low, high, high)
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def visualize(arr, low, high, pivot_index):     # Visualiza el proceso de ordenamiento
    plt.clf()
    color_array = []
    for idx in range(len(arr)):
```

```
        if idx == pivot_index:
            color_array.append('green')   # pivote
        elif low <= idx <= high:
            color_array.append('red')     # segmento actual
        else:
            color_array.append('blue')    # resto
    plt.bar(range(len(arr)), arr, color=color_array)
    plt.pause(0.3)

# -------------------- Búsqueda Binaria --------------------
def busqueda_binaria(arr, objetivo):
    izquierda, derecha = 0, len(arr) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        print(f"Buscando... izquierda: {izquierda}, derecha: {derecha}, medio: {medio}")
        if arr[medio] == objetivo:
            return medio
        elif arr[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1

# -------------------- Principal --------------------
if __name__ == "__main__":
    # Crear lista aleatoria
    data = random.sample(range(1, 50), 20)        #Lista de 20 números aleatorios entre 1 y 50
    print("Lista original:", data)                #Imprime la lista generada

    # Visualización inicial
    plt.ion()                                     #Activa el “modo interactivo” de matplotlib.
    plt.figure(figsize=(10, 6))                   #Define el tamaño de la figura

    # Ordenamiento con Quick Sort
    start = time.time()                           # Medir el tiempo de inicio
    quick_sort(data)
    end = time.time()                             # Medir el tiempo de finalización

    plt.ioff()                                    # Desactiva el modo interactivo
    plt.show()                                    # Muestra la visualización final

    print("Lista ordenada:", data)
    print(f"Tiempo de ordenamiento: {end - start:.4f} segundos")

    # Búsqueda binaria
    try:
        valor = int(input("Ingrese un número para buscar en la lista ordenada: "))
        resultado = busqueda_binaria(data, valor)
        if resultado != -1:
            print(f"Número {valor} encontrado en la posición {resultado}")
        else:
            print(f"Número {valor} no encontrado en la lista")
```

```
except ValueError:
    print("Entrada inválida. Por favor ingrese un número entero.")
```

Explicación:
1- Ordena una lista de números aleatorios con Quick Sort. Genera una lista aleatoria de 20 números distintos entre 1 y 49.
Ordena esa lista usando el algoritmo Quick Sort, que: Elige un pivote y divide la lista en menores y mayores. Ordena recursivamente cada parte.
Muestra una visualización animada del proceso de ordenamiento con matplotlib:
Rojo: el segmento que se está ordenando.
Verde: el pivote actual.
Azul: el resto de la lista.
Mide y muestra el tiempo que tardó en ordenar.

2- Permite buscar un número usando Búsqueda Binaria. Una vez que la lista está ordenada, el programa pide al usuario que ingrese un número. Usa el algoritmo de búsqueda binaria para ver si ese número está en la lista: Divide la lista en mitades sucesivamente. Imprime el proceso de búsqueda. Informa si encontró el número y en qué posición. Si el número no está, lo avisa.

Resultado final: Muestra la lista original, la lista ordenada, el tiempo de ordenamiento, y el resultado de la búsqueda del número que el usuario elige.

4. Metodología:

Implementación:
Se codificaron los algoritmos en Python desde cero, sin utilizar funciones de ordenamiento predefinidas. Cada algoritmo fue escrito respetando su lógica estructural tal como se presenta en la literatura especializada (Cormen et al., 2009).
Visualización: Se utilizó la librería matplotlib para animar el proceso de ordenamiento y mostrar gráficamente la evolución del arreglo durante la ejecución. Esta representación visual complementa la comprensión del funcionamiento interno de cada algoritmo, en línea con el análisis paso a paso propuesto por Cormen et al. (2009).

Evaluación: Se midió el tiempo de ejecución con el módulo time para distintos tamaños de lista. Esto permitió observar cómo se comportan los algoritmos en la práctica en relación con su complejidad teórica (Cormen et al., 2009).

5. Resultados

Quick Sort, demostró más eficiente para volúmenes grandes, confirmando lo que señala la teoría (Cormen et al., 2009).
La visualización facilitó la comprensión del funcionamiento interno del algoritmo, permitiendo observar cómo se reorganizan los elementos paso a paso. Los resultados prácticos se alinearon con la complejidad computacional descrita por Cormen et al. (2009), reafirmando la validez del análisis teórico.

6. Conclusiones:

Los algoritmos de ordenamiento son una pieza fundamental en el desarrollo de software, ya que permiten organizar datos de forma eficiente para su posterior uso. La implementación y visualización en Python facilitaron la comprensión de sus mecanismos internos.

Se busco por sobre las comparaciones, comprender el funcionamiento de uno de los algoritmos con mejor rendimiento, visualizar el proceso y medir el proceso. Dejando la posibilidad de comparar entre distintos algoritmos para su elección en un futuro.

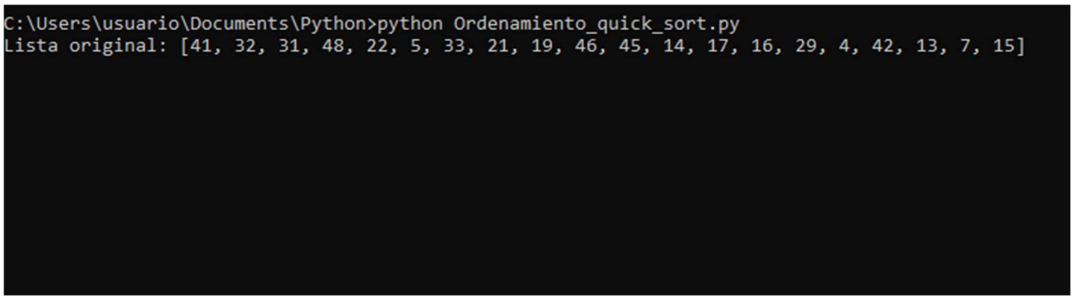
Bibliografía:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3.ª ed.). MIT Press.

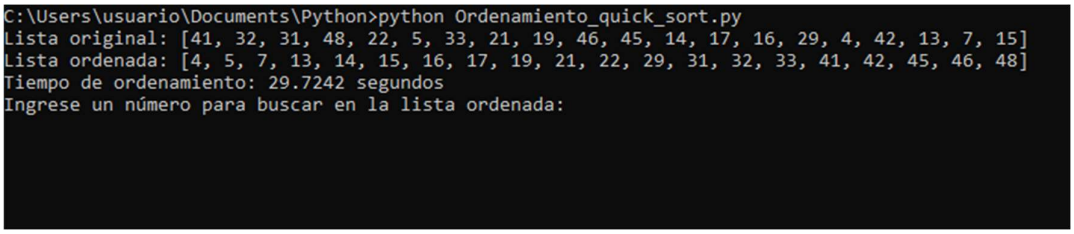
Python Software Foundation. (2024). Python documentation. https://docs.python.org/3/

Anexos:

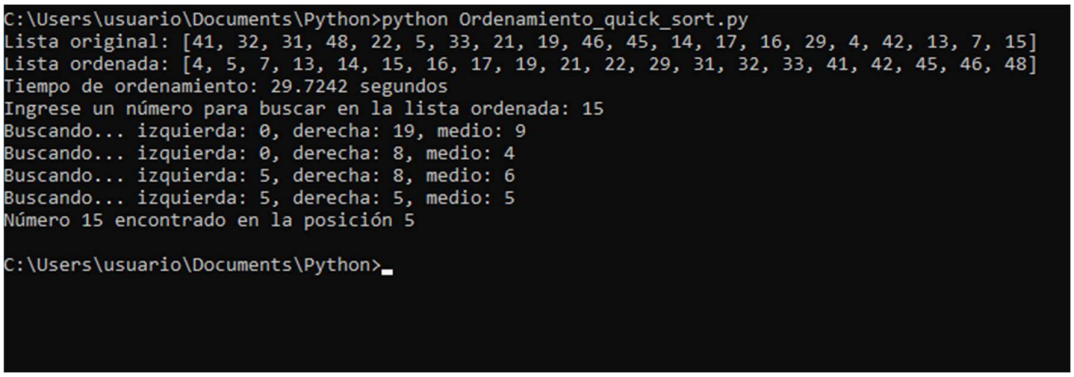
Captura de pantallas del programa funcionando:



Genera un conjunto de números aleatorios



Ordeno el conjunto de números de manera ascendente. Calculo el tiempo de ejecución.



Los gráficos muestran el proceso de ordenamiento. **Rojo**: el segmento que se está ordenando. **Verde**: el pivote actual. **Azul**: el resto de la lista.