

Quiz 1 - Búsqueda de código en Phind

Viviana Macela Garcia Valderrama
Braian Felipe Ramírez Ortiz
Brenda Lorena Vargas Parra

I. INTRODUCCIÓN

Este informe presenta un análisis del código en Python que implementa el algoritmo Merge Sort, uno de los métodos de ordenación más eficientes y ampliamente utilizados en Ciencias de la Computación. El objetivo principal es proporcionar una comprensión profunda de este algoritmo y su aplicación en la ordenación de grandes conjuntos de datos.

Merge Sort se basa en la estrategia "divide y vencerás" garantiza un rendimiento óptimo para conjuntos de datos considerables. A lo largo de este informe, analizaremos el código, explicaremos el algoritmo paso a paso y evaluaremos su complejidad temporal. También destacaremos la claridad del código, que facilita su comprensión, incluso para quienes no están familiarizados con los detalles del algoritmo.

Este informe servirá como una guía completa para comprender Merge Sort y su aplicación en el procesamiento eficiente de grandes volúmenes de datos.

II. ORIGEN DEL CÓDIGO A ANALIZAR

El código analizado se obtuvo de la plataforma Phind, que facilita la búsqueda y acceso a proyectos de código fuente. La elección de este código se basó en su representatividad y claridad en la implementación del algoritmo Merge Sort en Python, un algoritmo ampliamente utilizado para la ordenación de datos, a continuación se adjunta la imagen de como se realizó la búsqueda en Phind:

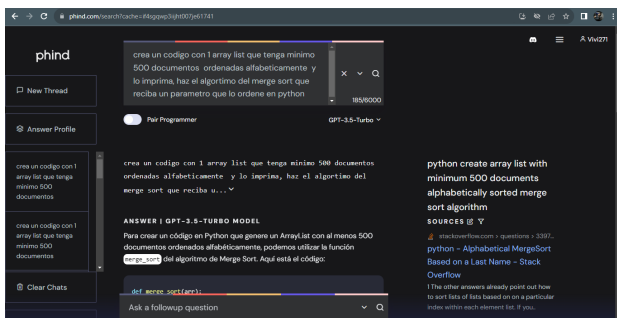


Figura 1. Consulta en Phind

III. CÓDIGO EN PYTHON

A continuación, se muestra el código Python:

```
1 import random # O(1)
2 import string # O(1)
3
4 documentos = [] # O(1)
5 for _ in range(500): # O(n)
6     documento =
7         ''.join(random.choices(string.ascii_uppercase +
8             string.ascii_lowercase, k=10)) # O(1)
9     documentos.append(documento) # O(1)
10 documentos.sort() # O(n log n)
11
12 for documento in documentos: # O(n)
13     print(documento) # O(1)
14
15 def merge_sort(arr): # O(n log n)
16     if len(arr) > 1: # O(1)
17         mid = len(arr) // 2 # O(1)
18         left_half = arr[:mid] # O(n)
19         right_half = arr[mid:] # O(n)
20
21         merge_sort(left_half) # O(n log n)
22         merge_sort(right_half) # O(n log n)
23
24         i = j = k = 0 # O(1)
25
26         while i < len(left_half) and j < len(right_half):
27             # O(n)
28             if left_half[i] < right_half[j]: # O(1)
29                 arr[k] = left_half[i] # O(1)
30                 i += 1 # O(1)
31             else:
32                 arr[k] = right_half[j] # O(1)
33                 j += 1 # O(1)
34             k += 1 # O(1)
35
36         while i < len(left_half): # O(n)
37             arr[k] = left_half[i] # O(1)
38             i += 1 # O(1)
39             k += 1 # O(1)
40
41         while j < len(right_half): # O(n)
42             arr[k] = right_half[j] # O(1)
43             j += 1 # O(1)
44             k += 1 # O(1)
45
46         merge_sort(arr) # O(n log n)
47
48 print("Lista ordenada:") # O(1)
49 for documento in documentos: # O(n)
50     print(documento) # O(1)
51
52 # La complejidad temporal del código es O(n log n)
```

IV. ANÁLISIS DEL CÓDIGO

IV-A. Generación de Datos

- En esta sección, se crea una lista desordenada de 500 documentos. Se utiliza un bucle `for` para generar cadenas aleatorias de 10 caracteres y agregarlas a la lista. La generación de cada cadena es una operación de tiempo constante $O(1)$ y, dado que se generan 500 cadenas independientemente del tamaño de n , la complejidad de tiempo total es $O(500)$, que se puede simplificar a $O(1)$, ya que es una constante.

IV-B. Implementación de Merge Sort

- La función `merge_sort()` implementa el algoritmo Merge Sort de manera recursiva. Divide la lista en mitades hasta que queden sublistas de un solo elemento y luego fusiona las sublistas ordenadas. La complejidad temporal de Merge Sort es $O(n \log n)$ en el peor caso, debido a la división y fusión de las sublistas.

IV-C. Ordenamiento y Verificación

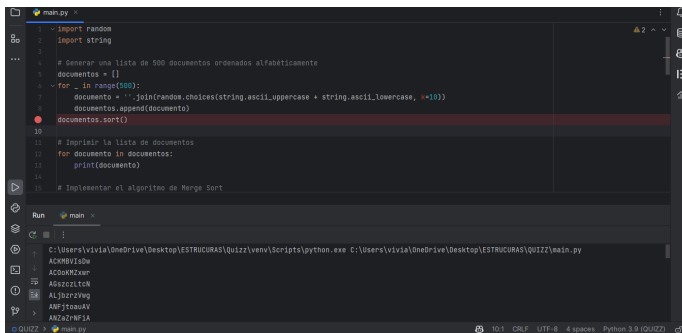
- La llamada a `merge_sort(documentos)` ordena la lista desordenada de documentos en $O(n \log n)$. La impresión posterior verifica visualmente el correcto funcionamiento del algoritmo.

V. COMPLEJIDAD ALGORÍTMICA

El algoritmo Merge Sort tiene una complejidad temporal de $O(n \log n)$, lo que lo hace eficiente para ordenar grandes volúmenes de datos. Además, tiene una complejidad espacial de $O(n)$ debido a la necesidad de memoria auxiliar para las fusiones.

VI. RESULTADOS DE LA EJECUCIÓN

La implementación del algoritmo Merge Sort en Python se ha ejecutado con éxito en un conjunto de datos compuesto por 500 documentos aleatorios. A continuación, se presenta una representación visual de la ejecución en la figura:



```

1 import random
2 import string
3
4 # Generar una lista de 500 documentos aleatorios
5 documentos = []
6 for _ in range(500):
7     documento = ''.join(random.choices(string.ascii_uppercase + string.ascii_lowercase, k=10))
8     documentos.append(documento)
9
10 documentos.sort()
11
12 # Imprimir la lista de documentos
13 for documento in documentos:
14     print(documento)
15
16 # Implementar el algoritmo de Merge Sort

```

Figura 2. Ejecución del código

VII. CONCLUSIÓN

En conclusión, el análisis detallado del algoritmo Merge Sort en Python demuestra su eficiencia y relevancia en la ordenación de datos. Algunos puntos clave a destacar son:

- El algoritmo Merge Sort, basado en la estrategia "divide y vence", ofrece un rendimiento óptimo en términos de tiempo para ordenar grandes volúmenes de datos.
- Su complejidad temporal de $O(n \log n)$ lo convierte en una herramienta valiosa en el procesamiento de datos a gran escala.
- La implementación en Python resulta en un código claro y legible, lo que facilita su comprensión y mantenimiento.
- Si bien es eficiente en la mayoría de los casos, es importante tener en cuenta que el rendimiento de Merge Sort puede variar según el tamaño de los conjuntos de datos y las aplicaciones específicas.