**ChatGPT**

Para determinar la complejidad total del algoritmo que involucra la estructura Trie y las operaciones definidas (`insert`, `insert\_R`, `get\_words`, `get\_wordsR`), vamos a analizar cada operación por separado y luego sumarlas:

Inserción (`insert` y `insert\_R`)

La inserción en un Trie implica recorrer el árbol caracter por caracter de la palabra a insertar. Si consideramos que la longitud máxima de las palabras es `L` y el número total de palabras es `N`, la complejidad de inserción puede ser considerada como:

Peor caso de inserción: O(L \* N)

Esto se debe a que, en el peor de los casos, se podría tener que recorrer toda la longitud de cada una de las `N` palabras para insertarlas en el Trie.

Obtención de Palabras (`get\_words` y `get\_wordsR`)

Para obtener todas las palabras almacenadas en el Trie junto con sus frecuencias de términos (TF), se recorre todo el árbol Trie. La complejidad de esta operación es:

Complejidad de obtener palabras: O(N)

Esto se debe a que se debe recorrer cada nodo del Trie exactamente una vez para recuperar todas las palabras almacenadas.

Complejidad Total

Para calcular la complejidad total del algoritmo que utiliza Trie con las operaciones definidas (`insert` y `get\_words`), podemos sumar las complejidades de ambas operaciones:

Inserción: O(L \* N)

Obtención de Palabras: O(N)

Entonces, la complejidad total del algoritmo sería:

Complejidad Total = O( L x N) + O(N)

En notación Big-O, esto se simplifica a:

Complejidad Total = O(L x N)

Donde `L` es la longitud máxima de las palabras y `N` es el número total de palabras insertadas en el Trie. Esta complejidad refleja el coste computacional máximo del algoritmo en términos de tiempo para las operaciones de inserción y obtención de palabras.

**BlackBlox**

La complejidad temporal de la create\_db función es O (n \* m) donde n es el número de documentos y m es el número promedio de palabras por documento. Esto se debe a que la función recorre en iteración cada documento y tokeniza cada palabra del documento.

La complejidad temporal de la search función es O(m \* log(u)) donde m es el número de palabras en el texto de entrada y u es el número de palabras únicas en la base de datos. Esto se debe a que la función tokeniza el texto de entrada, calcula las puntuaciones TF-IDF para cada palabra en el texto de entrada y luego calcula la similitud del coseno entre el texto de entrada y cada documento en la base de datos.

La complejidad temporal de la comparetexts función es O (m \* n) donde m es el número de palabras únicas en la base de datos y n es el número de documentos en la base de datos. Esto se debe a que la función calcula las puntuaciones TF-IDF para cada palabra de cada documento.

La complejidad temporal de la insert\_tokens función es O (m \* n) donde m es el número de palabras únicas en la base de datos y n es el número de documentos en la base de datos. Esto se debe a que la función crea un trie para cada documento e inserta cada palabra del documento en el trie.

La complejidad temporal de la load\_file función es O(n) donde n es el número de archivos en el directorio. Esto se debe a que la función recorre en iteración cada archivo del directorio y lo agrega a una lista.

La complejidad temporal de la print\_progress\_bar función es O(n) donde n es el número de iteraciones. Esto se debe a que la función actualiza la barra de progreso después de cada iteración.

La complejidad temporal de la save\_file función es O(n) donde n es el tamaño de los datos que se guardan. Esto se debe a que la función escribe los datos en un archivo.

La complejidad temporal de la file\_upload función es O(n) donde n es el tamaño de los datos que se cargan. Esto se debe a que la función lee los datos de un archivo.

La complejidad temporal de las funciones leer\_txt y leer\_pdf es O(n) donde n es el número de caracteres del archivo. Esto se debe a que las funciones leen el contenido del archivo y lo devuelven como una cadena. La complejidad temporal de la

clean\_text función es O(n) donde n es el número de caracteres en el texto de entrada. Esto se debe a que la función recorre en iteración cada carácter del texto de entrada y realiza varias operaciones de limpieza en él.

La complejidad temporal de la rankingfunción es O (m \* n) donde m es la cantidad de documentos en la base de datos y n es la cantidad de palabras en el texto de entrada. Esto se debe a que la función calcula la similitud del coseno entre el texto de entrada y cada documento de la base de datos.

La complejidad temporal de las funciones vector\_module y cosine\_similarity es O(n) donde n es el número de palabras en el texto de entrada. Esto se debe a que las funciones iteran a través de cada palabra en el texto de entrada y realizan varios cálculos sobre ella.

La complejidad temporal del comparetexts\_new La función es O(m \* n) donde m es el número de palabras únicas en la base de datos y n es el número de documentos en la base de datos. Esto se debe a que la función compara las puntuaciones TF-IDF de cada palabra en cada documento con el texto de entrada.

La complejidad temporal de la insert\_tokensfunción es O (m \* n) donde m es el número de palabras únicas en la base de datos y n es el número de documentos en la base de datos. Esto se debe a que la función crea un trie para cada documento e inserta cada palabra del documento en el trie.

La complejidad temporal de las funciones insert y insert\_R es O(m \* n) donde m es el número de palabras únicas en la base de datos y n es el número de documentos en la base de datos. Esto se debe a que las funciones crean un trie para cada documento e insertan cada palabra del documento en el trie.

La complejidad temporal de las funciones get\_words y get\_wordsR es O(m \* n) donde m es el número de palabras únicas en la base de datos y n es el número de documentos en la base de datos. Esto se debe a que las funciones iteran a través de cada nodo del trie y calculan la puntuación TF para cada palabra.

La complejidad temporal de la mainfunción es O(n) donde n es el número de documentos en la base de datos. Esto se debe a que la función llama a las funciones create\_bdy search\_textuna vez.

La complejidad temporal de la operaciones función es O(n) donde n es el número de documentos en la base de datos. Esto se debe a que la función llama al create\_bdysearch\_text

La complejidad de cada algoritmo.

1. TokenizeWords La complejidad de este algoritmo es O(n) donde n es el número de palabras de los documentos. Esto es porque:

* Itera sobre cada documento (O(n))
* Para cada documento, itera sobre cada palabra y la tokeniza (O(m))

La complejidad total es O(n) porque solo itera sobre cada palabra de los documentos.

1. Tf La complejidad de este algoritmo es O(n) donde n es el número de palabras de los documentos. Esto es porque:

* Itera sobre cada documento (O(n))
* Para cada documento, itera sobre cada palabra y calcula su frecuencia (O(m))

La complejidad total es O(n) porque solo itera sobre cada palabra de los documentos.

1. Tf\_Idf La complejidad de este algoritmo es O(n) donde n es el número de palabras de los documentos. Esto es porque:

* Itera sobre cada documento (O(n))
* Para cada documento, itera sobre cada palabra y calcula su IDF (O(m))
* Multiplica la frecuencia por el IDF para cada palabra (O(m))

La complejidad total es O(n) porque solo itera sobre cada palabra de los documentos.

En general, la complejidad del algoritmo depende de la cantidad de palabras de los documentos.

1. Insert La complejidad de este algoritmo es O(m), donde m es el número de caracteres de la palabra a insertar. Esto es porque:

* Crea un nuevo nodo si el trie está vacío (O(1))
* Incrementa el contador de palabras totales en el nodo raíz (O(1))
* Llama a la función recursiva insert\_Rpara insertar la palabra en el trie (O(m))

La complejidad total es O(m) porque solo itera sobre cada carácter de la palabra a insertar.

1. insert\_R La complejidad de este algoritmo es O(m), donde m es el número de caracteres de la palabra a insertar. Esto es porque:

* Comprueba si el nodo actual tiene un hijo con el carácter actual (O(1))
* Crea un nuevo nodo si no existe ningún hijo con el carácter actual (O(1))
* Llama a la función recursiva insert\_Rpara insertar el resto de la palabra en el trie (O(m-1))

La complejidad total es O(m) porque solo itera sobre cada carácter de la palabra a insertar.

1. get\_words La complejidad de este algoritmo es O(n) donde n es el número de palabras del trie. Esto es porque:

* Llama a la función recursiva get\_wordsR para obtener todas las palabras del trie (O(n))

La complejidad total es O(n) porque solo itera sobre cada palabra del trie.

1. get\_wordsR La complejidad de este algoritmo es O(m), donde m es el número de caracteres de la palabra actual. Esto es porque:

* Concatena la clave del nodo actual con la palabra actual (O(1))
* Comprueba si el nodo actual es el final de una palabra (O(1))
* Almacena la palabra actual y su frecuencia en el diccionario de palabras (O(1))
* Llama a la función recursiva get\_wordsR para obtener las palabras en los hijos del nodo actual (O(m))

La complejidad total es O(m) porque solo itera sobre cada carácter de la palabra actual.

En general, la complejidad del algoritmo depende de la cantidad de caracteres de las palabras y de la cantidad de palabras del trie.

1. clean\_text La complejidad de este algoritmo es O(n) donde n es el número de caracteres del texto. Esto es porque:

* Itera sobre cada carácter del texto (O(n))

La complejidad total es O(n) porque solo itera sobre cada carácter del texto.

1. vector\_module La complejidad de este algoritmo es O(n) donde n es el número de palabras del texto. Esto es porque:

* Itera sobre cada palabra del texto (O(n))

La complejidad total es O(n) porque solo itera sobre cada palabra del texto.

1. cosine\_similarity La complejidad de este algoritmo es O(n) donde n es el número de palabras del texto. Esto es porque:

* Itera sobre cada palabra del texto (O(n))

La complejidad total es O(n) porque solo itera sobre cada palabra del texto.

1. comparetexts La complejidad de este algoritmo es O(n) donde n es el número de palabras del texto. Esto es porque:

* Itera sobre cada palabra del texto (O(n))

La complejidad total es O(n) porque solo itera sobre cada palabra del texto.

1. ranking La complejidad de este algoritmo es O(n) donde n es el número de documentos. Esto es porque:

* Itera sobre cada documento (O(n))