**unordered\_map<string, int> map** is used to efficiently store unique words as keys and their corresponding counts as values in a hash table. Using an unordered map allows for fast insertion and retrieval of word counts. It's suitable for this task because it ensures that each word is unique, and it provides a quick way to update the count for each word as it appears in the text file. **vector<HashTable> ht** is vector of custom objects **HashTable** is used to store the word-count pairs, allowing us to sort them later. The use of a custom object encapsulates both the word and its count, making it easier to manipulate and sort as a whole. Additionally, this choice allows for sorting based on both word count and lexicographical order. The time complexity of my code is O(n log n) because the main time-consuming operation is the sorting of the **vector<HashTable> ht** using the standard sort function. The standard sort function has a time complexity of O(n log n) in the worst case. The rest of the operations such as reading words from the file, updating the unordered map, and creating the **vector<HashTable> ht** have a time complexity of O(n), where n is the number of words in the file. Therefore, the overall time complexity is dominated by the standard sorting operation, resulting in O(n log n) complexity.

A white background with black text

Description automatically generated

As a disclaimer, these times were recorded when I ended the timer before printing the results to the terminal. With “input1.txt”, this performance is excellent and suggests that the program can handle small files almost instantaneously. With “input2.txt”, the performance remains relatively good, especially considering the substantial increase in the input size. It's able to process a moderately large file in a reasonable amount of time. With “input3.txt”, the program's performance is still acceptable, especially given the growing input size. It can handle a relatively large text file within a reasonable time frame. The program's performance seems to scale well with input size. It's important to note that the execution time is likely influenced by factors such as the speed of the machine running the program and the efficiency of the C++ compiler and standard library functions used. The provided execution times suggest that the program is efficient and capable of processing input files of varying sizes, making it suitable for analyzing text data and generating word frequency statistics.