Anagram Finder – Design Explanaition

Overview

This Python script solves the *Anagram Finder* problem by reading a list of words from a file and grouping together those that are anagrams of one another.
An anagram is defined as a word formed by rearranging the letters of another, such as
“care” and **“race”**.

The program reads input from `sample.txt`, which contains one word per line, and prints each group of anagrams on a single line.

Approach and Logic

The code follows these steps:

1. **Read input**

   The input file is opened and read using:

   ```
   with open("sample.txt", "r") as file:
       content=file.read()
       words = content.split()
   ```

   This collects all words into a list. Since the file has one word per line, `.split()` works reliably here.

2. **Generate hashes of Sorted Words**

   For each word:

   - Its letters are extracted, sorted alphabetically, and joined back into a string
   - The `hash()` function is applied to the sorted string.

- This hash serves as a unique identifier for that word's letter combination.

Example:

```
'listen' → ['l','i','s','t','e','n'] → sorted → ['e','i','l','n','s','t'] →
→ 'eilnst' → hash('eilnst')
```

3. **Group and Print Anagrams**

   The code identifies all words with the same hash (i.e., sorted-letter signature) and prints them together. After printing, it removes those words and their hashes from the list to avoid repetition.

   This is repeated until all words are processed.

Design Decisions

- **Hashing sorted strings:**

  Using `hash(sorted_letters)` instead of using the sorted string itself simplifies and speeds up comparisons, especially when dealing with a large number of words.

- **Manual grouping instead of Dict:**

  The approach avoids dictionaries and uses basic lists to store hashes and words. While less efficient, it maintains clarity and aligns with the emphasis on code readability.

- **In-Place Removal of Processed Words:**

  Words are removed after being printed using reversed index removal to prevent index shifting issues during iteration.

- **No External Libraries Used:**

  The program relies only on built-in Python features (file I/O, sorting, lists, hashing), which keeps it lightweight and portable.

Scalability Considerations

For 10 Million Words:

- **Performance Improvements Needed:**
  - Replace the list-based approach with a `defaultdict` to store lists of anagrams based on sorted-letter keys.
  - Avoid repeated list traversals by processing all words in a single pass.

For 10 Billion Words:

- **Large-Scale Adaptation:**
  - **Streaming:** Read and process words line-by-line instead of loading them all into memory.
  - **External Storage:** Use databases or key-value stores to group anagrams incrementally.
  - **Efficient Handling:** Avoid collisions by using the actual sorted strings as keys, or even better, cryptographic hashes for massive scale

Summary

This solution is functional, readable and adheres closely to the assignment requirements. While not optimised for massive datasets, it serves as a solid base for further scaling. If extended for large-scale applications, improvements in data structure, streaming and distribution would be essential.