Design Decisions: AnagramFinder

Overview

The AnagramFinder program reads words from a .txt file and groups them into groups of anagrams which consist of words that have the same characters in a different order.

High-Level Design

Input: A plain .txt file containing words, one per line.
Processing:

1. Read each word from the file.
2. Convert the word to a signature (sorted characters `'cat' -> 'act'`).
3. Group words by their signature using a `HashMap`.

Output: Print all groups of anagrams to the console.

Data Structures

```
HashMap<String, List<String>> dict
```

Key: Sorted character signature of the word.
Value: List of words matching the signature.

I used the **HashMap** as a dictionary here because a **HashMap** provides **O(1)** average lookup and insertion, making grouping efficient and **List<String>** allows storage of multiple words per signature.

`Char[] chars` (used briefly in toSignature)

I used this character array as a temporary array to sort characters of a word. (Sorting ensures that all anagrams have the same key)

Methods

`toSignature(String word)`: Converts word to signature (array of sorted characters)
`printDict(Map<String, List<String>> dict)`: Prints all the anagrams from dict

Performance

Time Complexity:

1. **Reading words:** O(n)
2. **Generating signature:** O(k log k) per word, where k = length of word
3. **Grouping in HashMap:** O(1) per word (average)
4. **Printing groups:** O(n)

Overall: O(n * k log k)

Space Complexity:

O(n * k) for storing words in the HashMap.

Scalability Considerations

How would your solution handle 10 million words?

Using the current approach in the case of 10 million words we can estimate the memory use by taking an average word length of ~10 characters which would be ~20 bytes per word in Java as a String. Now 10 million words would be ~200 MB + overhead from the HashMap and that should fit well on a modern computer(>= 8 GB RAM). Also the sorting part depends on the word length which in most cases means it is really fast and wouldn't be a problem in this case.

What changes would be needed for 100 billion words?

100 billion words is a whole other story. With words being ~20 bytes, that would total at ~2 TB just for the words without the overhead. That is impossible to hold in RAM. I think there are multiple solutions each with its pros and cons but I think an easy to implement solution here would be a divide and conquer approach by splitting the words into chunks that are processed one at a time and then merge the results.