Thought Process and Documentation

Process and Decision Making

```
[
  {
    "Title": string,
    "ISBN": string,
    "Content": [
      {
        "Page": integer,
        "Line": integer,
        "Text": string
      }, zero or more...
    ],
  }, zero or more...
]
```

Considering the Input

The input that `findSearchTermInBooks` receives is a list of books in this JSON structure. We are also given a "word or search term" as a string. It is not made explicit exactly what this search term can look like:

1. Can it be a phrase?
2. Can it be a part of a word?
3. Can it have punctuation or other non-alphabetical characters?

I will have to make assumptions for these questions in my implementation.

```
{
  "SearchTerm": string,
  "Results": [
    {
      "ISBN": string,
      "Page": integer,
      "Line": integer
    }, zero or more...
  ]
}
```

Considering the Output

The output of `findSearchTermInBooks` should be the following JSON structure that collects book ids, page numbers, and line numbers for the instances of the search term.

For the output, I will also have to make assumptions. It is not clear if the output should include inexact matches. For example, is "then" a match for "the"? What about "the!" or "the,".

Other Considerations

```
[
  {
    "Title": string,
    "ISBN": string,
    "Words": {
      "the": [
        {
          "Page": integer,
          "Line": integer,
        }, one or more...
      ], for each word...
    }
  }, zero or more...
]
```

It may also be important to consider a space-time tradeoff for my implementation. The most brute force implementation would loop through all the books and for every book it would look through each line and for each line it would loop through every character/word in the text. This is **O(b x l x w)** time (**b** is number of books, **l** is number of lines, and **w** is number of words) which is not that good. A potential way to get better time would be to change the input structure to a different lookup structure so that I could map words to pages and lines.

With this input structure, I could loop through each book and then in constant time grab a list of all the lines where the word occurs. The question here is: is it worth it to do this restructuring? I think **it is not**. As the function is outlined now, each call passes in the full list of books, meaning that I wouldn't be able to take advantage of the restructured lookup multiple times in a row. Having the restructuring step would then mean that each call to the function is slower than the brute force implementation.

Final Assumptions

I will assume that the user is looking for matches of whatever they type in as the search term (partial words, words with punctuation, phrases, etc.) since this is how finders tend to work in most tools. This also means that I don't have to handle parsing punctuation or identifying specific words within text (if I were to attempt that, I would likely bias the function towards English language texts, which I don't want to do).

Testing and Iteration

Testing Strategy

To test my implementation, I want to make sure that I am correctly identifying positives (returning everywhere where the search term is) and negatives (not returning matches when there are none). I also want to make sure I am handling important edge cases appropriately (e.g., terms being capitalized, terms that are sub parts of other words, phrases, empty inputs, etc.).

Best Part of My Solution

The best parts of my solution are its simplicity and the way it behaves like common finder tools that you might find on a browser, a PDF viewer, or code editor. My solution is simply a loop with an inner loop to go through all the books and lines and I use the `includes` method to check if the term is in the text. Many tools have a finder function (**Ctrl** + **F**) that allows for searching over the text on the page. The basic feature of these finder functions is to look for exact matches across the text regardless of whether the search term is part of a word, a full word, or a phrase. Having my function work the same way means that users are likely to be familiar with its functionality, which means they are less likely to be confused by its results.

Hardest Part of the Problem

The hardest part of this problem was thinking through the assumptions that I would have to make based on what we were given in the documentation and instructions. If there were more information about the specific problem (maybe we have one big list of books that we are searching every time and we know all the books are in English and we know we are always looking for a specific set of words), I could have written the function in a different way that was more efficient (e.g., changing the input structure to be more efficient for word lookups).

Edge Cases and Other Tests

The edge cases that I tested for are 1) lowercase and uppercase not being the same, 2) emptiness in the input whether it be lack of books or empty book content, 3) partial words (i.e., a word that is inside another word), and 4) phrases (i.e., multiple words together).

The tests that I didn't do, but would ideally, are 1) testing on a very large input (for performance tests) and 2) usability tests (asking real users questions about the tool. For example, “Do the results make sense to you?” and “Was the tool helpful for finding what you needed?”