2024 Software Engineering Project-Based Assessment - U.S. Digital Corps (Written)

Jion Kim
jion0615@gmail.com

1 OVERALL PROCESS + DECISION MAKING

The first step is to understand the data structure of the inputs and outputs along with the objective of the function. The function findSearchTermInBooks() takes in a String searchTerm and an object array scannedTextObject. scannedTextObject is an array of book objects with a title, ISBN, and Content array. Within the content array are scanned objects with Page, Line, and Text elements. If the searchTerm is found within this Text element, then the ISBN, Page, and Line number must be turned into an object and placed into the results list.

Thus, we will need to search the Text element for every object within the Content element's array in each book to see if the searchTerm can be found. Because of the structure of scannedTextObject, to search every instance of Text, there needs to be an outer for loop to traverse every book in the array and a nested for loop inside the outer for loop to traverse every Content object with a text element in each book. Because the ISBN is the same for each content object that shares the same book but is unique to each book, the ISBN should be stored in a variable at the start of each iteration of the outer for loop. Then, within the inner for loop, there should be a part to search for the match. If there is a match, then the saved ISBN, Page, and Line number are pushed into an array. This array is a part of the "Result" element in the result object that will be returned by the function. This object has one other element which is also the searchTerm being searched for by the function. Each resulting match will result in an addition to the "Result" element's array. After the two loops are exhausted, the result array is returned.

The final part is to implement the search of the searchTerm in each of the Text elements. My first instinct was to use Javascript's native .includes() function for String objects to find all instances of the term. However, this way of doing things was not how I interpreted the use case of the function to be. The .includes() function also returns all instances in which the term is a substring of the Text. This meant that it included "then" as a positive match for "the". To make it more selective, I opted for a Regular Expression match with `\\b${searchTerm}\\b` as this allowed for exact standalone matches to be found while not ignoring terms that have punctuations or white spaces surrounding the word. This narrower selectivity is more in line with how the real-world context of the function should look like.

The time and space complexity of the function can be represented as O(n*m) where "n" is the number of books in scannedTextObj and "m" is the average number of content objects per book. Although the time complexity is n * m for all cases, the worst case in terms of space complexity is reached only when all the objects return a match.

2 TESTING AND ITERATION

2.1.0 Testing Strategy + Next Steps

The function passed the given tests in the original test suite but there were many other situations where it may fail. To test for real-world applications of this program, the test cases were centered around how this function may be used in practice. Additionally, edge cases were tested to ensure that the program would not behave in unexpected ways.

2.1.1 Positive Tests

1. The first positive test (positiveTest1) is a simple search for the term "you" in a set of 3 books with three objects each in its content. The term "you" was selected for this test because there are instances of you across the first two books but not the third. This tests whether or not the function works across different content objects and book objects. This needed to be done because the unit tests that come with this only test across a single book object. The function passes the test cases with 5 instances in which the term "you" can be found.
2. The second positive test (positiveTest2) is a search for the term "are". This tests for behavior when there is an instance of the word "are" in another word (in this case "rarely"). The unit test passes as the regex makes sure to not include "rarely" as a match, returning the objects of two lines where the word "are" can be found as a standalone word.

2.1.2 Negative Tests

1. The first negative test (negativeTest1) is a unit test used on the example input provided by the USDC template, customInput. It searches for the word "negative" which cannot be found in customInput. The function should return an output with the value of the "SearchTerm" key updated to "negative" but with an empty "Results" array. The function does so and passes the test.
2. The second negative test (negativeTest2) is a test on what happens if the scannedTextObj is an empty array (no books and no observations). Since there is no match the value of the "SearchTerm" key is updated to "negative" and the "Results" array is empty. This passes the test.

2.1.3 Capitalization Tests

1. The capitalization test (capitalTest1) takes in a new input (capitalInput) that is a list of 3 books and 3 objects within each book. The searchTerm to be found is "These". Notice that this is capitalized. In the input, there are 8 instances of the word "These" but only 5 of these are capitalized and the other 3 are in lowercase. The result should not include the 3 that are not capitalized and the function correctly does so, passing the test case. This matches the behavior described in the instructions where "The" is not considered the same as "the".

2.1.4 Edge Cases Tests

1. The first edge case test (edgeTest1) uses the same input as the capitalization test and tests for the inclusion of contractions or words that use apostrophes. In this case, the word "I'm" successfully shows up in the result, passing the test case.
2. The second edge case test (edgeTest2) tests for words ending not in white space but in things like a period, question mark, and a pluralizing "s". The first two are instances in which the word should show up in the search but the last is not. The function correctly returns instances of "element", "element." and "element?" but not "elements". This is done using the input used for the positive tests.
3. The third edge case test (edgeTest3) tests for instances of multiple words. It searches for the term "oysters and salmon" in the capitalInput. It successfully finds the instances where the words are found together in one line and inputs it into the "Results" array.

2.2 Portion of Solution That I'm Proud of

The portion of the solution that I'm most proud of is the test cases. I believe that planning these out gave me better insight into how a function like this should work in real-world situations instead of just a coding exercise. To build functional programs, I think that this was a good exercise to think through the design and use cases of these functions. Related to this, I am also really proud of how I was able to come up with a solution with regex's .match() with boundaries and to justify using it over the String method .includes(). This is because using the latter would have not been useful for real-world situations as it would include words that have the searchTerm as a substring of it which is not what we want.

2.3 Portion of Solution That Was Most Difficult

The portion of the solution that was most difficult was trying to find the most optimal solution. Although I don't believe that I have the most optimal solution in terms of time and space complexity, I believe that I have written an optimal enough code to work in real-world use cases while also being digestible and easy to explain. I also believe that I have addressed this problem the best I could without using external libraries and modules.

2.4 Edge Cases Addressed and Not Addressed

The edge case tests were designed to pick up untraditional words such as contractions and multi-words in addition to words that ended in punctuation. Additionally, the other test cases tested for cases that were not addressed by the original test cases such as empty inputs or multiple books. These edge cases were passed but there may have been others that were not addressed. If given more time, I would push the time and space complexity to the limit by introducing large test inputs and seeing how the function handles them. This may lead to further time and space optimization. More real-world scenarios may be introduced that may cause changes in the script; however, I wanted to maintain only what the instructions ask for specifically to not disqualify myself by accident.