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

Jion Kim
jion0615@gmail.com

1 OVERALL PROCESS + DECISION MAKING

1.1 Detailed Overview of Thought Process and Decisions

When tackling this project-based assessment, I wanted to structure my process towards understanding the heart and motivations behind the problem as it is not just a Leetcode style test in isolation. Off the bat, reading the prompt allows me to understand that this has real-world context that would be good to consider beyond what the instructions can clarify. The function findSearchTermInBooks() wants to log a list of objects with each of those objects containing the ISBN, Page, and Line numbers of each instance that the searchTerm can be found in the input scannedTextObj. In an everyday sense, this is straightforward; however, in coding the solution, it can be hard to implement it in the way we think of searching words intuitively. What if the search term is a part of a longer word - would this count? Or if a term is plural? What about a term attached to punctuation? Where can we look in scannedTextObj for our term? How should we format the function's output?

The first step in tackling this solution is to understand the data structure of the scannedTextObj and where we can look to search for our searchTerm. The scannedTextObj is an array with objects that represent books. These objects have a string Title, string ISBN, and an object array Content. This Content array is filled with objects that are made up of integer Page, integer Line, and string Text. The string Text is where we will need to find searchTerm. We will need to then search every book in the original scannedTextObj and the text in the Content array of each of those books. If a match is found, the function should store the ISBN, Page, and Line of the match in an object and push that object.

Now that we have discovered where to look for our searchTerm, we can now code a way to reach each of the "Text" elements in the Contents array in each book object. To first iterate through each instance of a book, a for loop must be written. This for loop should start at index 0 of scannedTextObj and continue until it reaches the last object in the array. Each index represents a book object.

Then for each book object, the Content array must be iterated through. This introduces another for loop that iterates through each object in the Content array for each book. Within this inner for loop, we now have access to the "Text" element of each object in the Content array of each book.

With access to "Text", we now need to find a way to search for the searchTerm in the element. The "Text" element is a String object which means that it is a piece of text and has a .includes() function. This function looks for all instances in which the searchTerm can be found in the string. This was my first approach; however, upon considering the real-world context of the problem, I opted to use a Regular Expression (regex) .match() function because of its selectiveness. The .includes function will return True for instances in which a part of the word can be found in another. For example, if searchTerm is "the", then even instances where "then" or "them" or even "thermoelectrometer" would cause the function to return True when we know that is not what we want. The regex `\\b${searchTerm}\\b` is more accurate in this instance because the \\b represents a boundary in which only white space and punctuation are included. This means that only standalone words count. Now we know how to determine if a piece of text includes the standalone word and how to get to that piece of text.

The last step is to figure out how to structure the result to match the sample output to pass the tests. The premade result variable has two keys: "SearchTerm" and "Results". The "SearchTerm" can be initialized with searchTerm so this change is reflected in the code. The "Results" key holds an array that is to be filled with objects with the ISBN, Page, and Line numbers of each instance that the searchTerm is found in the text. To do this, we need to go back to the for loop. After the initial outer for loop, the ISBN can be saved to a variable as it is constant throughout the duration of the next for loop. This is because each iteration of the outer for loop is its own book and should be updated every iteration. Then the object to be placed in the "Results" array should be constructed and pushed to the array whenever there is a match. This is done within the inner for loop and can be done with an if statement. If the text matches the regex (includes the searchTerm), then an object with the ISBN, Page, and Line is constructed and pushed to the "Results" array. Then the for loops continue until each object is searched. At the end, the result variable is made and returned.

All in all, the function is very simple and readable. In terms of the time complexity of the function, it can be stated as O(n * m) where "n" is the number

of books in each scannedTextObj and “m” is the average number of objects in the “content” array of each book. This is because there are two for loops in which the first loop iterates through each book and the inner loop looks at each object within the content array of each book. “m” can be an average number of objects in the “content” array of each book because “m * n” will represent the total amount of text that the function looks at. Time complexity has an additional factor of the regex tests. However, since each line of text can be estimated to have a maximum length (real-world context), the regex operation can be viewed as constant and the majority of the growth in time can be attributed to m and n at worst case scenarios. This is similar for space complexity which is also O(m * n). The amount of positive results determines how much memory the result variable needs. In the worst case, this is again O(m * n) because every object in every “Content” array of each book can turn up positive which would lead to “m * n” positive results to be stored in the result variable.

1.2 Visual Summary of Solution

```
/**
 * Searches for matches in scanned text.
 * @param {string} searchTerm - The word or term we're searching for.
 * @param {JSON} scannedTextObj - A JSON object representing the scanned text.
 * @returns {JSON} - Search results.
 * */
function findSearchTermInBooks(searchTerm, scannedTextObj) {
   /** You will need to implement your search and
    * return the appropriate object here. */

   // result is now initialized with the SearchTerm
   var result = {
       "SearchTerm": searchTerm,
       "Results": []
   };

   // outer for loop represents looking through different books and saves the ISBN in case a ma
   for (i = 0; i < scannedTextObj.length; i++) {
       let ISBN = scannedTextObj[i]["ISBN"];

       // inner for loop goes through the each object in the content and if the text includes t
       for (j = 0; j < scannedTextObj[i]["Content"].length; j++){
           // regex introduced instead of includes because the regex match is more selective (n
           let regex = new RegExp(`\\b${searchTerm}\\b`);
           if (regex.test(scannedTextObj[i]["Content"][j]["Text"])){

               result["Results"].push({
                                       "ISBN" : ISBN,
                                       "Page" : scannedTextObj[i]["Content"][j]["Page"],
                                       "Line" : scannedTextObj[i]["Content"][j]["Line"]
               })
           }
       }

   }

   return result;
}
```

Overview of function / solution.

```
25      // result is now initialized with the SearchTerm
26      var result = {
27          "SearchTerm": searchTerm,
28          "Results": []
29      };
```

The first step in the code is to initialize the result variable with the searchTerm with an empty "Results" array. This is in case no matches are found, in which case the variable can be returned. If matches are found in the for loop, then the corresponding object will be pushed into the empty array.

```
31      // outer for loop represents looking through diff
32      for (i = 0; i < scannedTextObj.length; i++) {
33          let ISBN = scannedTextObj[i]["ISBN"];
```

This is the outer for loop that loops through each book object. Each book object has a unique ISBN string that must be saved for later in case the searchTerm is found in the text.

```
54  /** Example input object. */
55  const twentyLeaguesIn = [
56      {
57          "Title": "Twenty Thousand Leagues Under the Sea",
58          "ISBN": "9780000528531",
59          "Content": [
60              {
61                  "Page": 31,
62                  "Line": 8,
63                  "Text": "now simply went on by her own momentum.  The dark-"
64              },
65              {
66                  "Page": 31,
67                  "Line": 9,
68                  "Text": "ness was then profound; and however good the Canadian\'s"
69              },
70              {
71                  "Page": 31,
72                  "Line": 10,
73                  "Text": "eyes were, I asked myself how he had managed to see, and"
74              }
75          ]
76      }
77  ]
```

This is found outside the function, but this is the example input. Notice how there is another array within Content with objects inside. This means that we

need another for loop to iterate through each of these objects. The inner for loop for this is found below.

```
35          // inner for loop goes through the each object in the content and if the text includes
36          for (j = 0; j < scannedTextObj[i]["Content"].length; j++){
37              // regex introduced instead of includes because the regex match is more selective
38              let regex = new RegExp(`\\b${searchTerm}\\b`);
39              if (regex.test(scannedTextObj[i]["Content"][j]["Text"])){
40
41                  result["Results"].push({
42                                              "ISBN" : ISBN,
43                                              "Page" : scannedTextObj[i]["Content"][j]["Page"],
44                                              "Line" : scannedTextObj[i]["Content"][j]["Line"]
45                  })
46              }
47          }
```

The inner for loop goes through each object in "Content" and matches the regex (`\\b${searchTerm}\\b`) to all the standalone words in the text. If the if statement is True, then the ISBN, Page, and Line numbers are formatted into a JSON object and pushed into the "Results" array of the result variable that was created in the first step of the function.

After this, the function has "return result;" in line 51, ending the function and returning the solution.

1.3 TLDR

The solution is to go through each book. Each book has an ISBN save this into a variable each time a new book is encountered. Then each book has an array or list of Content. Go through each object in the Content list. Each object in the Content list has a Page and Line number along with Text. If the searchTerm can be found in Text, then save the ISBN, Page, and Line numbers into a new object and push the newly made object to the results array. After going through each object in each book, return the object with the results array.

2 TESTING AND ITERATION

2.1.0 Testing Strategy + Next Steps

The function passed the given tests in the original test suite. However, these tests were not comprehensive of situations that may need more selectiveness and scrutiny. The main topics to tackle were multiple books, no books, capitalizations, terms within words, contractions, words ending in punctuation, and multiple words. These were determined to be important to test because this

should function as a real-world solution and not just some coding exercise. The following tests model this after what a person would look for in a function for search terms in scanned books. Although these tests are meant to be comprehensive, the next steps to test this even further would be to develop test cases in which the amount of data passed through the function is much larger. This would test things like time and space complexity limits. Another thing to test next is how it handles different forms of words such as present and past tense along with plural versions of the word.

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 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 lower case. 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 multiple 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 are the test cases. I believed 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 really 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 one 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.