John Melton
USDC Software Engineering Assessment

# Documentation of Thought Process for Coding Project

For this assessment, I approached the question using a "solve first, improve later" mindset. My initial goal was to come up with a way to get the results that the team was looking for, in this case creating a way to search a list of books and text for a given search term, such as "the". As a developer, I believe that if something can be done without nesting loops, then we already have sped up the process.

In the case of this project, I started off by splitting the book ISBNs and their respective text content and creating a new searchable object called 'contentObj'. Then, for each of the ISBNs in the 'contentObj', I checked if the Text included the search term, in this case "the". Whenever any of the Text was found to include the search term, I would add the found data to the Results array. This included the ISBN of the book, the Page Number, and the Line Number of where the search term was found. This yielded returned data that looked like this:

```
{
    "SearchTerm": "the",
    "Results": [
        {
            "ISBN": "9780000528531",
            "Page": 31,
            "Line": 9
        }
    ]
}
```

After getting successful return data and passing the two pre-written tests, I then began to think about what ways, if any, could I speed up the process without using any modules. I ended up re-writing the logic in multiple different ways to try and speed up the process.

The first change I tried sounded promising to me, but actually ended up performing a bit slower as well. This change was to use two pointer variables when checking the arrays containing the text. The two pointers would be used to check two indexes at a time instead of going through every index one at a time like the code I submitted does. One pointer would start at the first index of the array which is 0, and the second pointer would start at the last index of the array, which is the length of the array minus 1 (in this case 2). After every check, the first index pointer would be increased by 1, while the last index pointer would be decreased by 1. This logic would continue while the first index pointer was less than the last index pointer. The final check would be for arrays that had a length that was odd (i.e. 3 or 5 etc.). I would check the middle index as well at that point to see if the middle index included the search term. As I mentioned briefly above, this method ended up being slower than the split logic function I submitted that utilizes an if statement and checks each text content one at a time. This is probably due to the fact that JavaScript is natively a synchronous language (handling one task at a time), and therefore would not actually process the two pointers at the same time, but rather, one at a time. This ends up being not much different than reading one index at a time in the first place.

The last change I tried was to use the cleaner looking methods of map() and filter() to save on lines of code and not use if statements. To satisfy the exact result output request in the assessment, this change did not save many lines of code. Yes, it looked cleaner, but it ended up performing a little bit slower as well due to being more nested than the code I ended up submitting. With that said, its slower performance was hardly noticeable, and the cleaner code may be preferred by some developers. I tend to like the way it currently is written for readability, nonetheless.

While using the map() and filter() methods, I came up with another idea that is indeed faster and much cleaner than my other ideas above. The only issue with this idea is that it would entail changing the format of the output ever so slightly. Instead of having a return value of

```
    "SearchTerm": "the",
    "Results": [
        {
            "ISBN": "9780000528531",
            "Page": 31,
            "Line": 9
        }
    ]
}
```

it would have a return value that now included a Details section that held the Page number, Line Number, and the Text the search term was found in as seen below:

```
      {

    "SearchTerm": "the",
    "Results": [
        {
            "ISBN": "9780000528531",
            "Details": [
                {
                    "Page": 31,
                    "Line": 9
                    "Text": "ness was then profound; and however good the
Canadian\'s"
                }
            ]
        }
    ]
}
```

The idea above, with the slight change in formatting, allowed me to remove 12 lines of code by using the map() and filter() methods native to JavaScript. It also, as I mentioned above, sped up the process. I kept the code snippet in my submitted .js file to show how much cleaner the code would be. It is commented out and I give more details regarding the implementation as well (for me, the code can be found on line 54-57 and the comments for it begin on line 37).

Another idea I had, that I did not have time to implement, would be to create a function that matched books together who had the same number of scanned text samples. This would allow us to potentially search multiple books' text content at the same time using the same index. Even though JavaScript is natively synchronous, I believe this would be faster for much larger scanned objects. However, it would also be slower for smaller scanned objects due to the initial setup process of matching up equal length content arrays. So, in the case of this project, I did not implement this idea and was not able to test it.

Next came the testing. Two unit-tests were pre-written for this project and I used those as baselines for my next tests. Using the logic of these two tests, I wrote another two tests that checked for if a search term was found and if a search term was not found (positive and negative result tests). This would be a quick way to see if any given scanned object had found or not found the given search term instead of matching an exact returned object. The third test I wrote was a function that allows to test all the capitalization variations of a search term. Given the word "the", for example, this function would test to see if the three variations, "the, The, and THE", would return the proper object. This test would fail if "the" came back matching more than just its lowercase form. The last test I wrote was to test a scanned object that contained more than one book. This test ensures that, even with multiple books, the correct object is still returned given a search term. Another good test, depending on what kind of input is expected or allowed, would be to check if the user is inputting one word as a search term. Usually, most book searches allow for sentences, or multiple words, however, a test could be put in place to ensure a specific number of words is input or allowed.