Anett Martínez Vázquez. A01645033

Reflection.

During the development of this activity, we selected and implemented different algorithms to meet the requirements of each case. Each choice of these algorithms was guided by both the requirements of the task and the efficiency we wanted to achieve.

The first part required checking whether the malicious code sequences were present inside the transmission files. We identified that this type of problem a naive solution to find the pattern even if it could simply compare every possible position in the transmission against the malicious code, this has a time complexity of $O(m \times n)$, which becomes inefficient as the amount of characters of each of the files grows. To avoid this, we implemented the Knuth-Morris-Pratt (KMP) algorithm. KMP preprocesses the malicious code to build a Longest Prefix Suffix (LPS) array, to indicate where the next comparison should continue when a mismatch occurs, so instead of restarting the search from the next character, the algorithm uses the LPS to skip ahead, preventing redundant checks and at the same time reduces the time complexity to $O(m + n)$, which is far more efficient for large inputs, this efficiency becomes especially noticeable when dealing with long sequences of characters, since the algorithm avoids repeating comparisons that will fail.

The second part focused on detecting the longest palindrome within the transmissions. For this part we had to consider the assumption that malicious code often contains mirrored or symmetric patterns. An algorithm using brute force could test all possible combinations and check if they are palindromes, which is $O(n^2)$ or even $O(n^3)$for that reason ro solve this problem efficiently, we implemented Manacher's Algorithm which preprocesses the transmission by inserting a different character between characters so that all palindromes can be odd-length then, by expanding around each character as a potential center, it calculate the largest palindrome for each position in $O(n)$. Also Manacher's Algorithm can reuse what it previously computed while trying to expand palindromes. This choice made it possible to quickly locate the starting and ending positions of the longest sequence.

Finally the third part involved comparing the two transmission files in order to determine how similar they are, we identified that the task needed to find the longest common

substring between the transmissions and output the starting and ending positions of this substring in the first file. Naive implementations can lead to exponential time complexities making them inefficient. That is why we used a dynamic programming approach where we used a 2D matrix which stores results in a matrix and builds up the answer in steps. The resulting complexity is reduced to $O(m \times n)$, making the algorithm efficient enough for the input sizes. Whenever characters match, the value is extended from the previous diagonal cell and if characters are different the value stays as zero.

During this activity we were able to identify more clearly how different algorithms can be used depending on what we are trying to achieve, and also how that choices affect the efficiency of the process to find the solution we are looking for.

Test cases for part 1.

Test Case 1:

Files:

- transmission1.txt : 1234ABCD
- transmission2.txt: F1A23B
- mcode1.txt: 123
- mcode2.txt : A23
- mcode3.txt : ABCD

Results:

1. mcode1 = 123

Found in transmission1 starting at position 0 (1234ABCD)

Output: true 0

2. mcode2 = A23

Found in transmission2 starting at position 2 (F1A23B)

Output: true 2

3. mcode3 = ABCD

Found in transmission1 starting at position 4 (1234ABCD)

Output: true 4

This case tests basic matching at start, middle, and end of a transmission.

```
true 0
false
true 4
false
true 2
false
```

Test Case 2:

Files:

- transmission1.txt : ABCDEF
- transmission2.txt : 123456
- mcode1.txt :999
- mcode2.txt : EEE
- mcode3.txt: FA1

Results:

- None of the mcode strings exist in either transmission.

```
false
false
false
false
false
false
```

This checks that the program correctly returns false when no match exists.

Test Case 3:

Files:

- transmission1.txt: ABABABAB
- transmission2.txt: 123123123

- mcode1.txt : ABAB
- mcode2.txt : 123
- mcode3.txt : BAB

Results:

```
true 0
true 2
true 4
false
true 1
true 3
true 5
false
true 0
true 3
true 6
false
```

This case tests multiple possible matches, we can observe the program prints all of them.