ReflexActInt1_A01638996

For the Integral Activity 1, there were three main problems, for each one of them we used a different algorithm that could resolve the problem efficiently.

The first part of this activity asked us to analyze whether the transmission files contained the contents of any of the mcode files. We could have used a naïve approach, checking each character and restarting the comparison every time there was a mismatch. In the worst case, this approach would take ($O(n * m)$). However, we decided to use the KMP algorithm for its better efficiency. KMP keeps information that the naïve approach wastes: it stores a Longest Prefix Suffix (LPS) array, which allows the algorithm to avoid useless shifts of the pattern. This reduces the complexity to **($\boldsymbol{O(n+m)}$)**, making the algorithm very efficient for searching patterns in long strings.

For the second part, we were asked to search for the biggest palindrome in the transmissions files. A brute-force algorithm would compare all substrings ($\mathrm{O}(\mathrm{n}^3)$), while a dynamic programming approach would reduce this to $O(n^2)$. Nevertheless, we decided to use Manacher´s algorithm, the key idea behind this algorithm is that palindromes have a symmetry property. If we know the longest palindrome centered at some position, we can reuse this information to estimate the palindrome length at its mirrored position when evaluating a new center. This prevents us from repeatedly expanding substrings that we already know cannot be longer., we also added # between the characters to account for even length palindromes, the complexity of this algorithm is $\boldsymbol{O(n)}$.

For the final part, we had to search for the Longest Common Substring between the transmission files. Similarly to the previous problems, we could have chosen between a brute force ($O(n * m^2)$), dynamic programming ($O(n * m)$) or a suffix tree $O(n + m)$. However, unlike the last problem, we decided to use dynamic programming. Our solution builds a matrix where each cell $[i][j]$ represents the length of the longest common suffix ending at positions i in the first string and j in the second; Whenever the characters match, we update the cell as $[i][j] = [i-1][j-1] + 1$, if the value was higher than the last one, its updates as the LCS, giving us a complexity of $\boldsymbol{(O(n * m))}$. The reason we did not use the most efficient method, was for a lack of knowledge and time to implement the algorithm properly.

Test cases part 1

Test Case 1:

Files:

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

Expected Output:

true 0

false

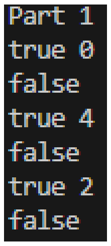
true 4

false

true 2

false

Output:



```
Part 1
true 0
false
true 4
false
true 2
false
```

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

Test Case 2:

Files:

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

Expected Output:

false

false

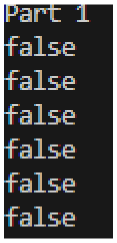
false

false

false

false

Output:



```
Part 1
false
false
false
false
false
false
```

Purpose of test: This checks that the program correctly returns false when no match exists.

Test Case 3:

Files:

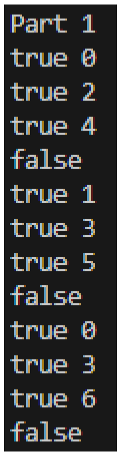
- transmission1.txt: ABABABAB
- transmission2.txt: 123123123
- mcode1.txt: ABAB
- mcode2.txt: 123
- mcode3.txt: BAB

Expected Output:

```
true 0

true 2

true 4

false

true 1

true 3

true 5

false

true 0

true 3

true 6

false
```

(Various instances of the same mcode in the same transmission (first 7: transmission 1) (last 4: transmission 2))

Output:



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

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

Test cases part 2

Test Case 1:

Files:

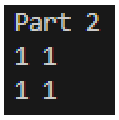
- Transmission1.txt: wertyuiorefcqgaryklcvweyugfdiopñzamqdtrejhgfiuytam
- Transimission2.txt: zxcasdqwevbndfgertmnlkjhyuiopñwertyuiopñgdazmcdjx

Expected Output:

1 1

1 1

Output:



```
Part 2
1 1
1 1
```

Purpose of test: This case verifies that the algorithm correctly handles situations where there are no palindromes in either transmission.

Test Case 2:

Files:

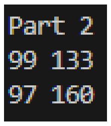
- Transmission1.txt: calkiuxfdqsxbazyunmakqncawqpwprlfkowvlegknxbhhwmajjhlqqpimqymoczqnoilmjybdny rojjgkibkpkolhbuknpnlrjikagdcqffchdjtzmkmztjdhcffqcdgakijaxa
- Transimission2.txt: gixwjdlpgogmocqolikvcibybrvfaywtypttsyldwrxndjhgfkfkqscevmjhpxphjmvecsqkfkfghjdn xrwdlysttpythnwpbtysedkrapuecptfvlruqmtuhrlizzuccuzzilrhutmqurlvftpceuparkdesytbz

Expected Output:

99 133

97 160

Output:



```
Part 2
99 133
97 160
```

Purpose of test: This case verifies that the algorithm correctly handles situations where there is a palindrome in both transmissions.

Test cases part 3

Test Case 1:

Files:

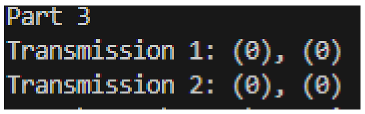
- transmission1.txt: ABCDEFG
- transmission2.txt: HIJKLMN

Expected Output:

Transmission 1: (0), (0)

Transmission 2: (0), (0)

Output:



```
Part 3
Transmission 1: (0), (0)
Transmission 2: (0), (0)
```

Purpose of test: This case verifies that the algorithm correctly handles situations where there is no common substring at all. It tests if the matrix initializes properly and avoids false positives.

Test Case 2:

Files:

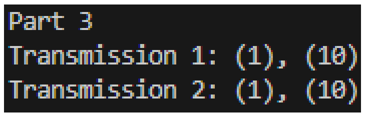
- transmission1.txt: HELLOWORLD
- transmission2.txt: HELLOWORLD

Expected Output:

Transmission 1: (1), (10)

Transmission 2: (1), (10)

Output:



```
Part 3
Transmission 1: (1), (10)
Transmission 2: (1), (10)
```

Purpose of test: This ensures the algorithm works correctly when one string is entirely contained in the other. It validates that the algorithm can find the maximum possible substring without prematurely stopping.

Test Case 3:

Files:

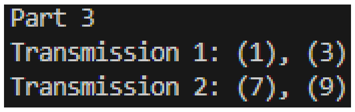
- transmission1.txt: ABCXXX
- transmission2.txt: YYYZZZABC

Expected Output:

Transmission 1: (1), (3)

Transmission 2: (7), (9)

Output:



```
Part 3
Transmission 1: (1), (3)
Transmission 2: (7), (9)
```

Purpose of test: This test verifies that the algorithm can correctly identify a substring that appears at the end of one string and at the beginning of the other. It ensures the solution does not only work for substrings that make a complete match, but also for substrings located wherever in the strings.