Integrative Activity 1

José Carlos Martínez Núñez, Tania Sayuri Guizado Hernández

Tecnológico de Monterrey

September 11, 2022

Contents

Problem Description 3
Algorithm and Implementation 3
Kuth-Morris-Pratt Algorithm (KMP) 3
Manacher's Algorithm 5
Longest Common Substring 8
Test Cases 9
Hardware 11

Problem Description

In teams of maximum 3 people, write a C++ program that reads 5 text files (fixed name, not requested from the user) containing only characters from 0 to 9, characters between A and F and line breaks.

- `transmission1.txt`
- `transmission2.txt`
- `mcode1.txt`
- `mcode2.txt`
- `mcode3.txt`

Transmission files contain text characters that represent the sending of data from one device to another. The mcode#.txt files represent malicious code that can be found within a transmission.

The program must analyze if the contents of the files mcode1.txt, mcode2.txt and mcode3.txt are contained in the files transmission1.txt and transmission2.txt and display a true or false if the sequences of chars are contained or not. If true, display true, followed by exactly one space, followed by the position in the transmissionX.txt file where the code of mcodeY.txt begins.

Assuming that malicious code always has "mirrored" code (palindromes of chars), it would be a good idea to look for this type of code in a stream. The program should then look for "mirrored" code inside the transmission files (palindrome at chars level, do not mess with bit level). The program displays in a single line two integers separated by a space corresponding to the position (starting at 1) where the longest "mirrored" code (palindrome) for each stream file starts and ends. It can be assumed that this type of code will always be found.

Finally, the program analyzes how similar the transmission files are, and must show the initial position and the final position (starting at 1) of the first file where the longest common substring between both transmission files is found.

Algorithm and Implementation

Kuth-Morris-Pratt Algorithm (KMP)

For the first part of the problem (finding the mcode files in the transmission files), we used the KMP algorithm, which is a string searching algorithm that uses a table to store the results of previous calculations, and it uses this table to avoid recalculating the same results. The table the algorithm uses is called the **Longest Proper Prefix Suffix** table (LPS), and it stores the length of the longest proper prefix that is also a suffix of the for every substring of the pattern. This allows the program to solve the first part of the problem in linear time (The Worst Case). The algorithm is implemented in the `KMP()` function.

```
// KMP Algorithm
int KMP(string text, string pattern)
{
    int n = text.length();
    int m = pattern.length();
    int lps[m];
    int j = 0;
    int i = 0;
    computeLPSArray(pattern, m, lps);
    while (i < n)
    {
        if (pattern[j] == text[i])
        {
            j++;
            i++;
        }
        if (j == m)
        {
            return i - j;
            j = lps[j - 1];
        }
        else if (i < n && pattern[j] != text[i])
        {
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
    return -1;
}
```

The `computeLPSArray()` function is used to compute the LPS table, and it is implemented as follows:

```
// Longest Proper Prefix Suffix Table
void computeLPSArray(string pattern, int m, int *lps)
{
    int len = 0;
```

```
    lps[0] = 0;
    int i = 1;
    while (i < m)
    {
        if (pattern[i] == pattern[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if (len != 0)
            {
                len = lps[len - 1];
            }
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

This algorithm in comparison with the naive algorithm, has a time complexity of $O(n)$, where n is the length of the text, and the naive algorithm has a time complexity of $O(nm)$, where m is the length of the pattern.

Manacher's Algorithm

For the second part of the problem (finding the longest palindrome in the transmission files), we used the Manacher's algorithm, which is a linear time algorithm that finds the longest palindrome in a string. This algorithm also uses a LPS table, in order to work. It is important to note that both this algorithm and the KMP algorithm because they both use LPS tables, they both have a space complexity of $O(n)$, where n is the length of the text. The algorithm is implemented in the `findLongestPalindromicString()` function.

```
void findLongestPalindromicString(string text)
{
    int n = text.length();
    if (n == 0)
        return;

    n = 2 * n + 1;
    int lps[n];
    computeLPSArray(text, n, lps);

    int center = 1, right = 2, i = 0;

    int mirror;
    int expand = -1, diff = -1;
    int max_length = 0, max_center = 0;
    int start = -1, end = -1;

    for (i = 2; i < n; i++)
    {
        mirror = 2 * center - i;
        expand = 0;
        diff = right - i;

        if (diff >= 0)
        {

            if (lps[mirror] < diff)
                lps[i] = lps[mirror];

            else if (lps[mirror] == diff && right == n - 1)
                lps[i] = lps[mirror];

            else if (lps[mirror] == diff && right < n - 1)
            {
                lps[i] = lps[mirror];

                expand = 1;
            }
```

```
            else if (lps[mirror] > diff)
            {
                lps[i] = diff;

                expand = 1;
            }
        }
        else
        {
            lps[i] = 0;

            expand = 1;
        }

        if (expand == 1)
        {

            while (((i + lps[i]) < n && (i - lps[i]) > 0) && (((i + lps[i] + 1) % 2 == 0) ||
            ↪  (text[(i + lps[i] + 1) / 2] == text[(i - lps[i] - 1) / 2])))
            {
                lps[i]++;
            }
        }

        if (lps[i] > max_length)
        {
            max_length = lps[i];
            max_center = i;
        }

        if (i + lps[i] > right)
        {
            center = i;
            right = i + lps[i];
        }
    }
```

```
    start = (max_center - max_length) / 2;
    end = start + max_length - 1;

    cout << start << " " << end << endl;

    if (debug)
    {
        cout << "The longest palindrome is: " << text.substr(start, end - start + 1) << endl;
    }
}
```

It is important to note that the `computeLPSArray()` function is the same as the one used in the KMP algorithm. The algorithm in comparison with the naive algorithm, has a time complexity of $O(n)$, where n is the length of the text, and the naive algorithm has a time complexity of $O(n^2)$. It is also important to note that for this part of the problem, we are considering both spaces and new lines as characters that can be part of the palindrome.

Longest Common Substring

For the third part of the problem (finding the longest common substring in the transmission files), we used the Longest Common Substring algorithm, which is a dynamic programming algorithm that finds the longest common substring in two strings. This algorithm uses a 2D table in order to work. It is important to note that this algorithm has a time complexity of $O(nm)$, where n is the length of the first string and m is the length of the second string. The algorithm is implemented in the `longest_common_substring()` function.

```
void longest_common_substring(string str1, string str2)
{
    int m = str1.length();
    int n = str2.length();
    int result = 0;
    int endingIndex = m;
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
```

```
        {
            if (str1[i - 1] == str2[j - 1])
            {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                if (dp[i][j] > result)
                {
                    result = dp[i][j];
                    endingIndex = i;
                }
            }
            else
                dp[i][j] = 0;
        }
    }
    cout << endingIndex - result << " " << endingIndex - 1 << endl;
    if (debug)
    {
        cout << "The longest common substring is: " << str1.substr(endingIndex - result, result)
    << endl;
    }
}
```

Test Cases

In order to perfom test cases more easily, we created the program `generate_transmissions.cpp`, which generates random transmission files. The program takes as input a range for the length of the malicious code files, and a range for the length of the transmission files. Based on this the program generates two transmission files of random length, random number of codes inside them (placed in random positions), and it also generates three `mcodeX.txt` files, which contain the random codes generated based on the range. This program also gives us useful information in order to test our algorithms, such as the length of each malicious code, which transmission files it was placed in and where it was placed. ***Note: once you compile the `main.cpp` program you can add any argument to it, and it will run in debug mode, which will print the information about the malicious codes and the transmission files.***

An example output of the `generate_transmissions.cpp` program is shown below.

Given the following ranges:

Malicious code: 10 - 20 characters

Tranmission Files: 500 - 1000 characters

```
--- Generating codes ---

Generating code mcode1.txt
Generated code of length 19 and saved to mcode1.txt
Code: F1D0B6ba0b39e6f043B

Generating code mcode2.txt
Generated code of length 14 and saved to mcode2.txt
Code: 27d2fe91F917e4

Generating code mcode3.txt
Generated code of length 19 and saved to mcode3.txt
Code: 0aC88A4A1B8FfffABb6



--- Generating transmissions ---

Generating transmission transmission1.txt
Inserted code of length 19 at position 153
Inserted code: F1D0B6ba0b39e6f043B

Inserted code of length 14 at position 727
Inserted code: 27d2fe91F917e4

Inserted code of length 19 at position 381
Inserted code: 0aC88A4A1B8FfffABb6

Generated transmission of length 847 with 3 codes and saved to transmission1.txt

Generating transmission transmission2.txt
Inserted code of length 19 at position 591
Inserted code: F1D0B6ba0b39e6f043B

Inserted code of length 19 at position 484
Inserted code: 0aC88A4A1B8FfffABb6
```

```
Generated transmission of length 881 with 2 codes and saved to transmission2.txt
```

Attached you can find the `test_cases` folder, which contains the test cases we used to test our program.

Hardware

For this activity we used a MacBook Pro (13-inch, 2018, Four Thunderbolt 3 Ports) with the following specs:

- **Processor:** 2.3 GHz Quad-Core Intel Core i5
- **Memory:** 8 GB 2133 MHz LPDDR3
- **Graphics:** Intel Iris Plus Graphics 655 1536