Assignment Data structure and Algorithm

Instructor: Prof Fahad Iqbal

Student name :Syed Muhammad Maroob

Roll No: 19-BSCS-29

Q1: Write an algorithm and develop a program that finds a string is palindrome or not?

Explanation:

This problem can be trivially solved by looping through each character and checking it against the character on the opposite side. There is a problem with this though because half the work being done is redundant as it's checking all characters two times. Consider the palindrome "madam", this algorithm would make the following comparisons:

m ↔ m

a ↔ a

d ↔ d

a ↔ a

m ↔ m

All that needs to be compared to prove it's a palindrome are the first two characters against the last two since the middle one does not need to be checked:

m ↔ m

a ↔ a

Time complexity of below algorithm is O(n / 2) which is equal to O(n) since Big-O notation ignores constant terms.

Source code :

```
#include <iostream>
#include <string.h>
using namespace std;
int main()
{
   char str1[20], str2[20];
   int i, j, len = 0, flag = 0;
   cout << "Enter the string : ";
   gets(str1);
   len = strlen(str1) - 1;
   for (i = len, j = 0; i >= 0 ; i--, j++)
      str2[j] = str1[i];
```

```
    if (strcmp(str1, str2))
        flag = 1;
    if (flag == 1)
        cout << str1 << " is not a palindrome";
    else
        cout << str1 << " is a palindrome";
    return 0;
}
```

Source code link: https://github.com/Maroob123/check-string-is-palindrome-or-not

Q2: Write an algorithm and develop a program that finds the substring and its position?

Explanation:

For solving the above problem I used KMP(Knuth Morris Pratt) algorithm instead of Naive pattern searching algorithm for the less time complexity. The worst case complexity of the Naive algorithm is O(m(n-m+1)). The time complexity of KMP algorithm is O(n) in the worst case.

Algorithm:

Step1: start comparison of pat[j] with j = 0 with characters of current window of text.

Step2: keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep matching.

Step3: When we see a mismatch

- Characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).
- lps[j-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.
- From above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match

Source code:

```
// C++ program for implementation of KMP pattern searching
// algorithm
#include <bits/stdc++.h>

void computeLPSArray(char* pat, int M, int* lps);

// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
    // values for pattern
```

```
    int lps[M];

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while (i < N) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }

        // mismatch after j matches
        else if (i < N && pat[j] != txt[i]) {
            // Do not match lps[0..lps[j-1]] characters,
            // they will match anyway
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

// Fills lps[] for given patttern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
```

```
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0) {
                len = lps[len - 1];

                // Also, note that we do not increment
                // i here
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// Driver program to test above function
int main()
{
    char txt[] = "ABABDABACDABABCABAB";
    char pat[] = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
}
```

Source code link : https://github.com/Maroob123/kmp-algorithm-by-c-