Shubhan Singh

SE-Comps B/Batch C

2022300118

DAA Experiment 10: String Matching (Knuth Morris Pratt Algorithm)

Problem statement: Given a string and a pattern, write an algorithm to find all occurrences of the pattern in the string.

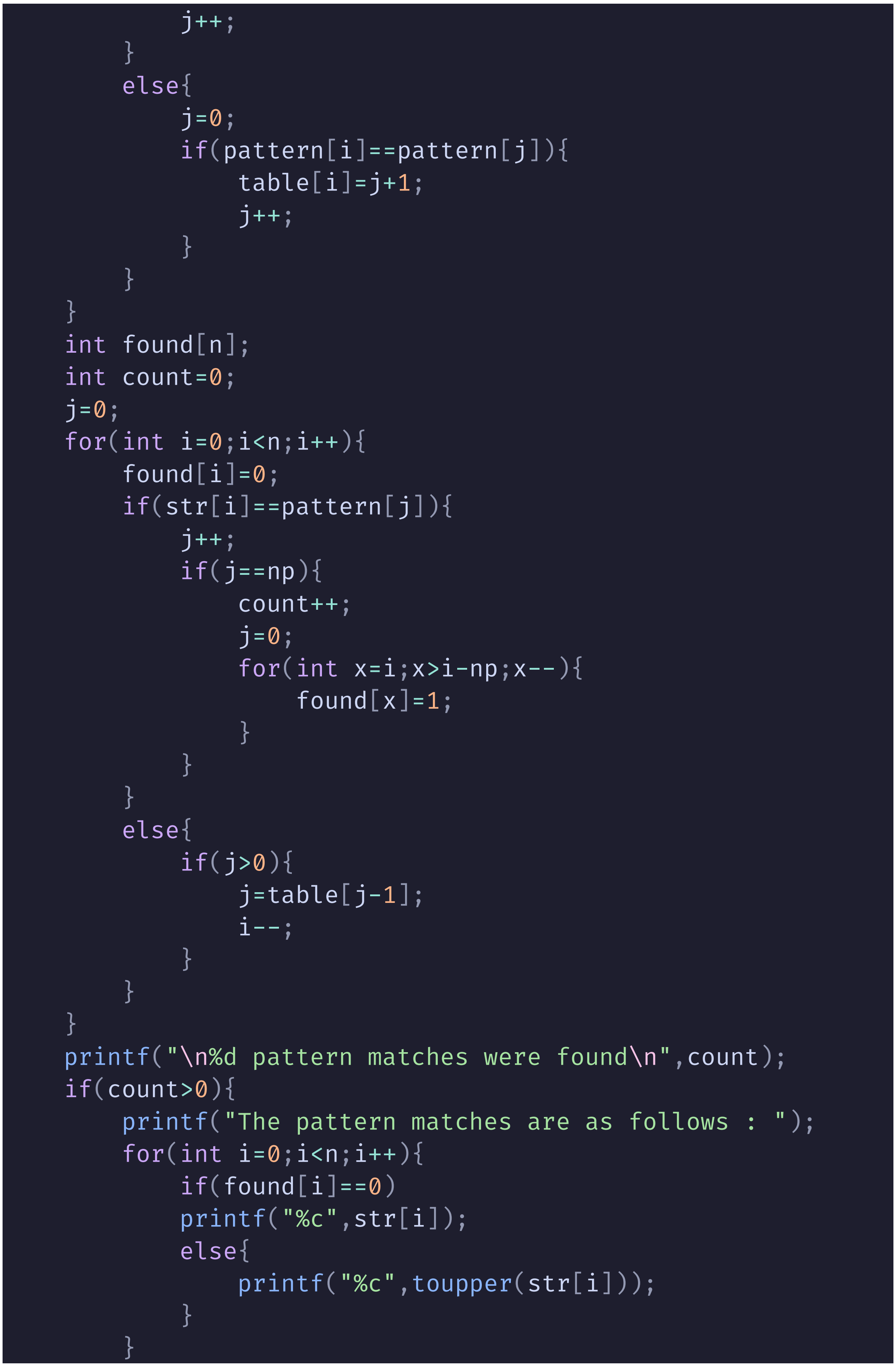
Pseudocode:

```
Algorithm KMP-String-Matching
  Input: A string str and a pattern pattern
  Output: The number of occurrences of the pattern in the string


 1. Read str from the user
 2. Read pattern from the user
 3. Remove the newline character from the end of str and pattern
 4. np = length of pattern
 5. n = length of str
 6. Initialize an array table[1..np] with all elements as 0
 7. j = 0
 8. for i = 1 to np
 9.     if pattern[i] == pattern[j]
10.         table[i] = j + 1
11.         j = j + 1
12.     else
13.         j = 0
```

```
14.         if pattern[i] == pattern[j]
15.             table[i] = j + 1
16.             j = j + 1
17. Initialize an array found[1..n] with all elements as 0
18. count = 0
19. j = 0
20. for i = 0 to n
21.     if str[i] == pattern[j]
22.         j = j + 1
23.         if j == np
24.             count = count + 1
25.             j = 0
26.             for x = i down to i - np + 1
27.                 found[x] = 1
```

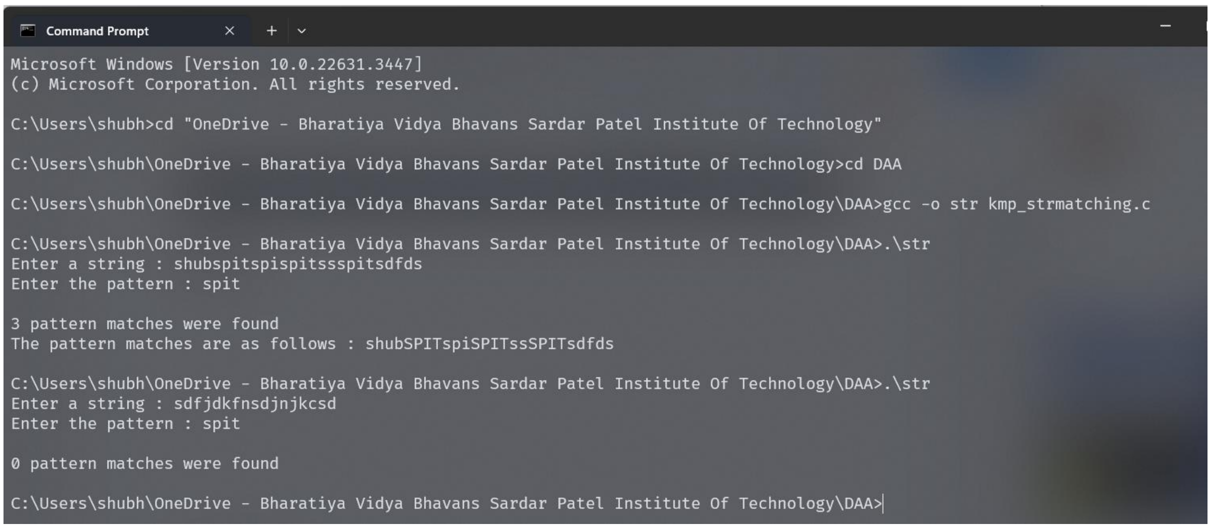
<u>Source code(C language):</u>

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>

int main(){
    char str[1024];
    char pattern[256];
    printf("Enter a string : ");
    fgets(str,1023,stdin);
    printf("Enter the pattern : ");
    fgets(pattern,255,stdin);
    str[strlen(str)-1]=pattern[strlen(pattern)-1]='\0';
    int np=strlen(pattern),n=strlen(str);
    int table[np];
    memset(table,0,np*sizeof(int));
    int j=0;
    for(int i=1;i<np;i++){
        if(pattern[i]==pattern[j]){
            table[i]=j+1;
```



```c
            j++;
        }
        else{
            j=0;
            if(pattern[i]==pattern[j]){
                table[i]=j+1;
                j++;
            }
        }
    }
    int found[n];
    int count=0;
    j=0;
    for(int i=0;i<n;i++){
        found[i]=0;
        if(str[i]==pattern[j]){
            j++;
            if(j==np){
                count++;
                j=0;
                for(int x=i;x>i-np;x--){
                    found[x]=1;
                }
            }
        }
        else{
            if(j>0){
                j=table[j-1];
                i--;
            }
        }
    }
    printf("\n%d pattern matches were found\n",count);
    if(count>0){
        printf("The pattern matches are as follows : ");
        for(int i=0;i<n;i++){
            if(found[i]==0)
            printf("%c",str[i]);
            else{
                printf("%c",toupper(str[i]));
            }
        }
        printf("\n");
    }

return 0;
}
```

Output:

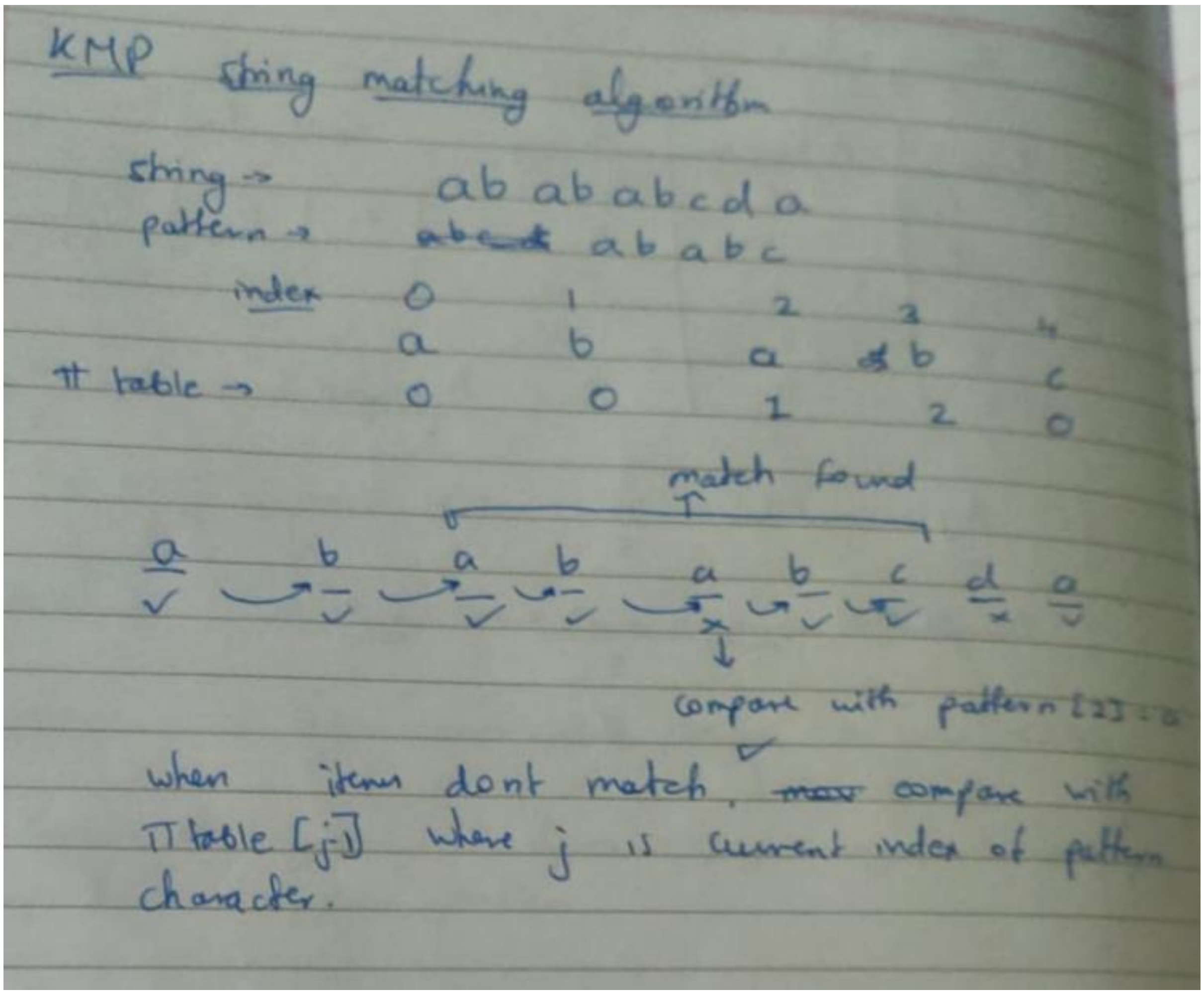


```
Microsoft Windows [Version 10.0.22631.3447]
(c) Microsoft Corporation. All rights reserved.

C:\Users\shubh>cd "OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology"

C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology>cd DAA

C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology\DAA>gcc -o str kmp_strmatching.c

C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology\DAA>.\str
Enter a string : shubspitspispitssspitsdfds
Enter the pattern : spit

3 pattern matches were found
The pattern matches are as follows : shubSPITspiSPITssSPITsdfds

C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology\DAA>.\str
Enter a string : sdfjdkfnsdjnjkcsd
Enter the pattern : spit

0 pattern matches were found

C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute Of Technology\DAA>
```

Conclusion:

- We solved the 15 puzzle problem using branch and bound technique.
- We greedily chose the node closest to the solution and killed all other nodes, and applied this recursively.
- We checked beforehand whether the puzzle was solvable or not to prevent infinite recursion

Rough Working :



KMP string matching algorithm

string → ab ab ab c d a

pattern → ~~abcd~~ ab a b c

index	0	1	2	3	4
	a	b	a	b	c
π table →	0	0	1	2	0

match found
a ✓
b ✓
a ✓
b ✓
a ✗
b ✓
c ✓
d ✗
a ✓

compare with pattern[2] : a ✓

when items dont match, ~~mov~~ compare with π table [j-1] where j is current index of pattern character.

Theory:

String Matching Algorithms

String matching algorithms are used to find one, several, or all occurrences of a pattern in a text. They are fundamental in various fields such as information retrieval, bioinformatics, data mining, and software engineering. There are several string-matching algorithms, each with its own strengths and weaknesses, and they are chosen based on the specific requirements of the task.

Some common string-matching algorithms include:

1. **Naive String-Matching Algorithm**: This is the simplest and most straightforward string matching algorithm. It checks all characters from the start to the end of the text for a match with the pattern. Its time complexity is O(m*n), where m is the length of the text and n is the length of the pattern.

2. **Rabin-Karp Algorithm**: This algorithm uses hashing to find any one of the set of pattern strings in the text. It is beneficial when the pattern size is small and the text size is large. It uses a rolling hash to quickly filter out positions of the text that cannot match the pattern.

3. **Boyer-Moore Algorithm**: This algorithm skips sections of the text, resulting in a lower time complexity in practice when compared to its competitors. The key feature of the algorithm is that it starts comparing from the last character of the pattern.

4. **Aho-Corasick Algorithm**: This algorithm is used to locate all occurrences of any of a finite set of strings within some input text. It constructs a finite state machine from a keyword tree, which significantly speeds up the search.

Knuth-Morris-Pratt (KMP) String Matching Algorithm

The Knuth-Morris-Pratt algorithm offers a more efficient way to handle the situation when a mismatch occurs. The key idea in this algorithm is to avoid the comparisons with the text that we know will anyway match.

The KMP algorithm preprocesses the pattern and constructs an auxiliary array, called a prefix function or failure function, that is used to skip characters while matching.

When a mismatch occurs, the algorithm uses this table to skip over the characters in the pattern that we know will certainly match and continues to check the rest of the characters.

The time complexity of the KMP algorithm is O(n), where n is the length of the text. This makes it much more efficient than the naive string-matching algorithm, especially for longer texts and patterns.