Experiment No.	10
Aim	Implement String Matching Algorithms
Name	Aarush Kinhikar
UID No.	2021300063
Class & Division	SE Comps A, Batch D

Theory:

The naive approach to string matching is the simplest and most straightforward algorithm for finding a pattern within a text string. The algorithm simply compares each character in the text string against the characters of the pattern string in a sliding window fashion until it finds a match or reaches the end of the text string.

The algorithm starts by aligning the first character of the pattern string with the first character of the text string. If the two characters match, the algorithm moves to the next character in both the pattern and the text strings and checks for a match. If the characters do not match, the algorithm slides the window one position to the right in the text string and starts the comparison process again. This process continues until a match is found or the end of the text string is reached.

The naive approach has a time complexity of O(mn), where m is the length of the pattern string and n is the length of the text string. This means that the algorithm can be very slow for large text strings or patterns. However, the naive approach is easy to understand and can be used as a starting point for more advanced string-matching algorithms.

The Rabin-Karp algorithm is a string-matching algorithm that uses hashing to compare a pattern string with a text string. It is named after its inventors, Michael O. Rabin, and Richard M. Karp, and is based on the concept of fingerprinting.

The algorithm works by computing the hash value of the pattern string and the hash value of each substring of the text string of the same length as the pattern string. If the hash values match, the algorithm compares the pattern string with the substring to confirm if it is a true match or not. If the hash values do not match, the algorithm moves to the next substring and computes its hash value.

The hash function used in the Rabin-Karp algorithm is a rolling hash function that efficiently updates the hash value of each substring by removing the hash value of the leftmost character and adding the hash value of the rightmost character. This allows the algorithm to compute the hash values of all substrings in O(n) time, where n is the length of the text string.

The Rabin-Karp algorithm has a time complexity of O(m + n), where m is the length of the pattern string and n is the length of the text string. In practice, the algorithm can be faster than the other string matching algorithms for some inputs, but can be slower if there are many hash collisions.

Algorithm:

Naive: -

```
1. Set n to be the length of the text string T.
2. Set m to be the length of the pattern string P.
3. For i from 0 to n-m:
    a. Set j to 0.
    b. While j < m and T[i+j] equals P[j]:
        (i). Increment j.
    c. If j equals m:
        (i). Return i.
4. Return -1.
```

Rabin-Karp: -

1. Compute the hash value h(P) of the pattern string P.
2. Set n to be the length of the text string T.
3. Set m to be the length of the pattern string P.
4. Compute the hash value h(T[0:m]) of the first substring of T of length m.
5. For i from 0 to n-m:

```
a. If h(T[i:i+m]) equals h(P):
    (i).If T[i:i+m] equals P, return i as the index of the first occurrence of P in T.
b. If i < n-m:
    (i). Update the hash value of the next substring T[i+1:i+m+1] using the rolling hash function.
```

6. Return -1 to indicate that P was not found in T.

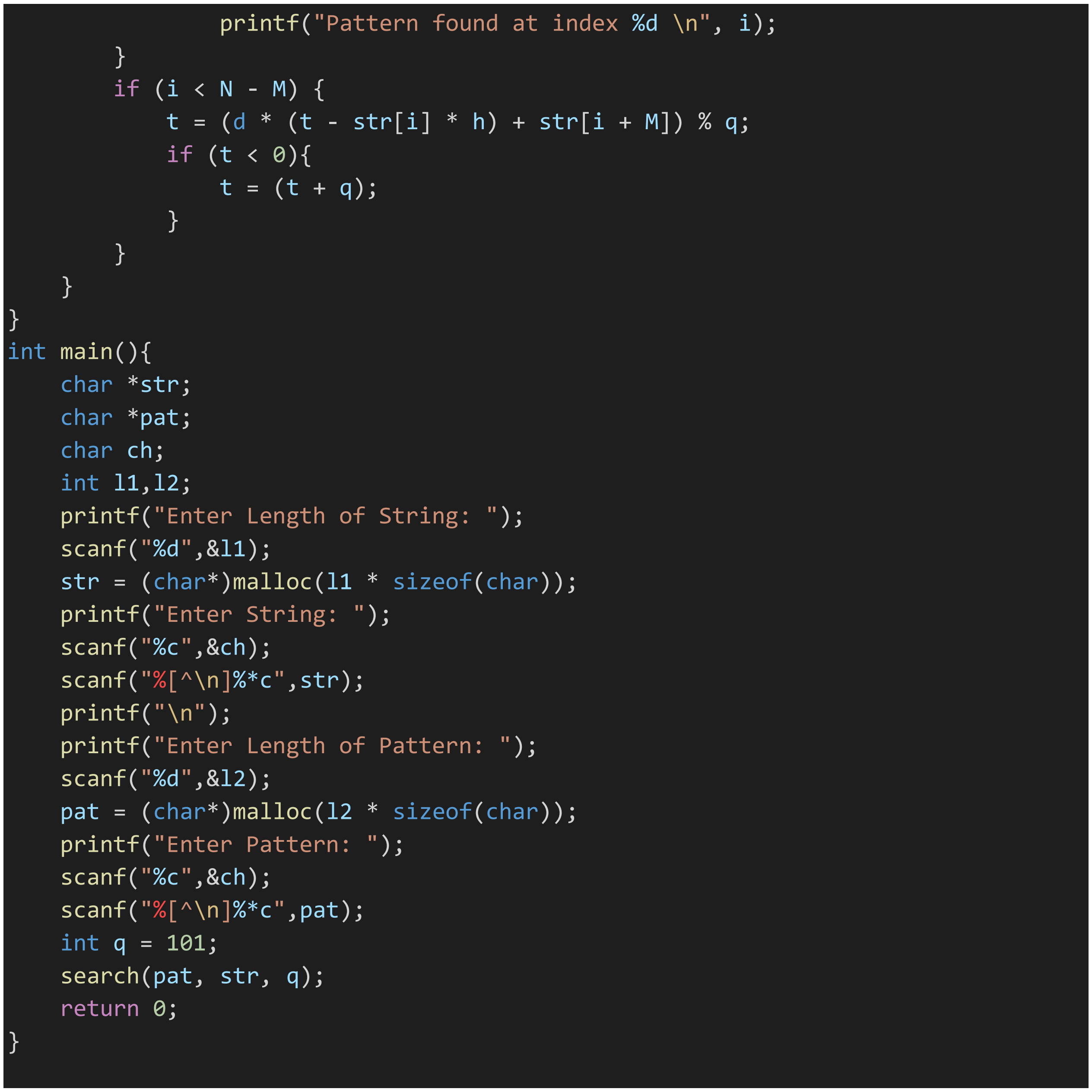
Program:

Naive: -

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void search(char* pat, char* str)
{
    int M = strlen(pat);
    int N = strlen(str);
    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++){
            if (str[i + j] != pat[j]){
                break;
            }
        }
        if (j == M){
            printf("Pattern found at index %d \n", i);
        }
    }
}
int main(){
    char *str;
    char *pat;
    char ch;
    int l1,l2;
    printf("Enter Length of String: ");
    scanf("%d",&l1);
    str = (char*)malloc(l1 * sizeof(char));
    printf("Enter String: ");
    scanf("%c",&ch);
    scanf("%[^\n]%*c",str);
    printf("\n");
    printf("Enter Length of Pattern: ");
    scanf("%d",&l2);
    pat = (char*)malloc(l2 * sizeof(char));
    printf("Enter Pattern: ");
    scanf("%c",&ch);
    scanf("%[^\n]%*c",pat);
    search(pat, str);
    printf("\n");
    return 0;
}
```

Rabin-Karp: -

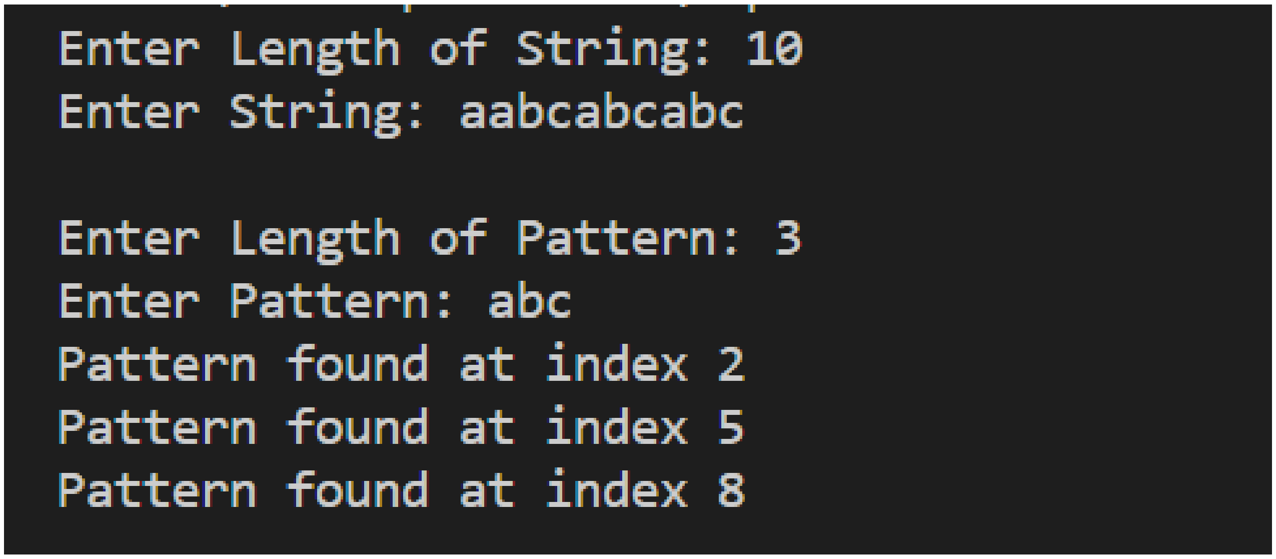
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define d 256
void search(char pat[], char str[], int q)
{
    int M = strlen(pat);
    int N = strlen(str);
    int i, j;
    int p = 0;
    int t = 0;
    int h = 1;
    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;
    for (i = 0; i < M; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + str[i]) % q;
    }
    for (i = 0; i <= N - M; i++) {
        if (p == t) {
            for (j = 0; j < M; j++) {
                if (str[i + j] != pat[j])
                    break;
            }
            if (j == M)
```



```c
                printf("Pattern found at index %d \n", i);
        }
        if (i < N - M) {
            t = (d * (t - str[i] * h) + str[i + M]) % q;
            if (t < 0){
                t = (t + q);
            }
        }
    }
}
int main(){
    char *str;
    char *pat;
    char ch;
    int l1,l2;
    printf("Enter Length of String: ");
    scanf("%d",&l1);
    str = (char*)malloc(l1 * sizeof(char));
    printf("Enter String: ");
    scanf("%c",&ch);
    scanf("%[^\n]%*c",str);
    printf("\n");
    printf("Enter Length of Pattern: ");
    scanf("%d",&l2);
    pat = (char*)malloc(l2 * sizeof(char));
    printf("Enter Pattern: ");
    scanf("%c",&ch);
    scanf("%[^\n]%*c",pat);
    int q = 101;
    search(pat, str, q);
    return 0;
}
```

Results:

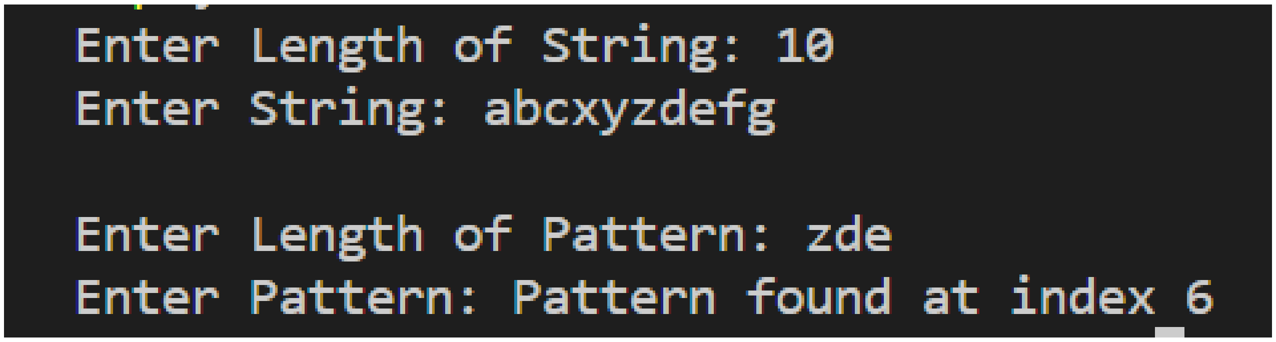
Naive: -



```
Enter Length of String: 10
Enter String: aabcabcabc

Enter Length of Pattern: 3
Enter Pattern: abc
Pattern found at index 2
Pattern found at index 5
Pattern found at index 8
```

Rabin-Karp: -



```
Enter Length of String: 10
Enter String: abcxyzdefg

Enter Length of Pattern: zde
Enter Pattern: Pattern found at index 6
```

Conclusion:

In this experiment, I studied two string matching algorithms: the naive approach and the Rabin-Karp algorithm. The naive approach is a simple and intuitive algorithm that compares each character of the pattern string with the corresponding characters in the text string. While it has a worst-case time complexity of O(n*m), it can perform well in practice when the pattern string is short and the text string is relatively small. The Rabin-Karp algorithm is a more sophisticated algorithm that uses hash values to compare the pattern string with substrings of the text string. It has a worst-case time complexity of O(n+m) and can perform much faster than the naive approach for larger text strings and longer pattern strings. In general, the Rabin-Karp algorithm is a more efficient and powerful algorithm than the naive approach. However, it requires additional computational overhead to calculate hash values, which can be a limiting factor for very large text strings or patterns.