Experiment #4			
Date		Student ID	
		Student Name	

- **Procedure/Program:**

```
#include <stdio.h>
int main(){
    char *txt[] ={"welcome EVERYONE", "HAVE
                                A NICE DAY"};
char *Pattern[] = {"ONE", "NICE"};
    for(int i=0; i<2; i++){
    for(int k=0; k<txt[i][k]; k++){
        int j=0;
    while(Pattern[i][j] && txt[i][k+j]==
                        Pattern[i][j]) j++;
    if(!Pattern[i][j]){
Printf("Pattern found at index %d\n", k);
        break;
        }
    }
    }
    return 0;
}
```

- **Data and Results:**

- **Analysis and Inferences:**

Data: Texts are "welcome EVERYONE"
and "HAVE A NICE DAY".

Results: Pattern "ONE" found at index 14;
"NICE" at index 8

Analysis: Patterns found in texts indicate
search efficiency and relevance.

Inferences: Both patterns effectively
located, demonstrating string matching
success.

Experiment #4			
Date		Student ID	
		Student Name	

In-Lab:

1. Write a program to implement naive string matching algorithm for the below example. Consider txt[] = "AAAAABBAABAAAACC", pat[] = "AAAA"

- **Procedure/Program:**

```
#include <stdio.h>
int main(){
char txt[]= "AAAAABBAABAAAACC",
Pat[] = "AAAA";

for(int i=0; txt[i]; i++){
int j=0;
while (Pat[j] && txt[i+j] == Pat[j])
j++;
if(!Pat[j]){
Printf("Pattern found at index %d\n",i);
return 0;
}
}
Printf("Pattern not found\n");
return 0;
}
```

- **Data and Results:**

Data: Text: "AAAAABBAABAAAACC", Pattern: "AAAA".
2 2

Result: Pattern: "AAAA" found at index 0
2 2 2 in the text.

Analysis: Pattern: "AAAA" successfully located at
2 2

the start of text.

inferences: Navie algorithm efficiently
2 2

matches patterns in simple

cases.

- **Analysis and Inferences:**

2. Naive method and KMP are two string comparison methods. Write a program for Naïve method and KMP to check whether a pattern is present in a string or not. Using clock function find execution time for both and compare the time complexities of both the programs (for larger inputs) and discuss which one is more efficient and why? Sample program with function which calculate execution time:

```
#include<stdio.h>
#include<time.h>
void fun() {
//some statements here
}
int main() {
//calculate time taken by fun()
clock_t t;
t=clock();
fun();
t=clock()-t;
double time_taken=((double)t)/CLOCK_PER_SEC; //in seconds
printf("fun() took %f seconds to execute \n",time_taken);
return 0;
}
```

- **Procedure/Program:**

```
#include <stdio.h>
#include <string.h>
#include <time.h>
int main() {
```

```
char text[] = "ABABDABACDABABCABAB";
char Pattern[] = "ABABCABAB";
int n = strlen(text), m = strlen(Pattern);
clock_t start = clock();
    int naiveResult = -1;
for(int i=0; i<=n-m; i++){
    int j;
  for(j=0; j<m; j++)
if(text[i+j] != Pattern[j]) break;
if(j==m) {naiveResult = i; break;}
}
double naiveTime = (double)((clock() - start)/
                            CLOCKS_PER_SEC;
int lps[m], j=0;
for(int i=1; i<m; i++){
while(j>0 && Pattern[i] != Pattern[j])
                        j = lps[j-1];
if(Pattern[i] == Pattern[j]) j++;
    lps[i] = j;
}
    start = clock();
        j=0;
```

```
int kmpResult = -1;
for(int i =0; i<n; i++){
  while (j>0 && text[i] != Pattern[j])
    j = lps[j-1];
  if (text[i] == Pattern[j]) j++;
  if (j == m) { kmpResult = i - m + 1;
                break; }
}
double kmpTime = (double)(clock() - start) /
                 CLOCKS_PER_SEC;
printf("Naive : %s at index %d, time : %.6f\n",
       naiveResult != -1 ? "found" : "not found",
       naiveResult, naiveTime);
return 0;
}
```

- **Data and Results:**

Data:

The input consists of a text and a search pattern.

- **Analysis and Inferences:**

Result:

Pattern found at specific index with execution time displayed

Experiment #4		Student ID	
Date		Student Name	

Post-Lab:

Given a pattern of length- 5 window, find the valid match in the given text by step-by-step process using Robin-Karp algorithm

Pattern: 2 1 9 3 6

Modulus: 21

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Text: 9 2 7 2 1 8 3 0 5 7 1 2 1 2 1 9 3 6 2 3 9 7

- **Procedure/Program:**

Pattern: 2 1 9 3 6

Modulus: 21

Text : 9 2 7 2 1 8 3 0 5 7 1 2 1 2 1

9 3 6 2 3 9 7

1) Hash of Pattern:

Hash(P) = 12

2) initial Hash of Text window (9 2 7 2 1):

Hash(T[0:4]) = 3

3) Sliding windows:

- window 1 : Hash = 3 (no match)
- window 2 : Hash = 2 (no match)
- window 3 : Hash = 6 (no match)
- window 4 : Hash = 18 (no match)
- window 5: Hash = 6 (no match)

window 6 : Hash = 3 (no match)

window 7 : Hash = 18 (no match)

window 8 : Hash = 12 (match found)

Experiment #4		Student ID	
Date		Student Name	

- **Data and Results:**

Data: Pattern: 2 1 9 3 6 Text: 9 2 7 2 1
2

Result: Pattern found at index 13 in the given
= 2 text.

- **Analysis and Inferences:**

Inferences: Rabin-Karp efficiently found the
2 2
Pattern using hash matching
Analysis: Hashing simplifies
- **Sample VIVA-VOCE Questions (In-Lab):** matching, sliding through text
2

1. What is the main advantage of using the Knuth-Morris-Pratt (KMP) algorithm over the naive string matching algorithm? with reduced
2. How does the naive algorithm search for a pattern in a text? complexity
3. What are the time complexities of the naive pattern matching algorithm in the best, worst, and average cases?
4. What is the prefix function (or partial match table) in the context of the KMP algorithm?
5. What are the time complexities of the KMP algorithm for preprocessing and searching?

Evaluator Remark (if Any):	**Marks Secured:** 49 **out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.

1) KMP is faster due to avoiding redundant comparisons

2) It checks each positions for a matching pattern.

3) Best: o(m), worst : o(n*m), Average: o(n)

4) Prefix function indicates longest prefix matching a suffix.

5) Preprocessing : o(m), searching : o(n)

for total o(n+m)