

**GHARDA FOUNDATION'S**

**GHARDA INSTITUTE OF TECHNOLOGY**

**A/P:-LAVEL, TALUKA: KHED, DIST. RATNAGIRI, STATE:MAHARASTRA,**

# ***LABORATORY MANUAL***

***Department  
Of  
Computer Engineering***

***Analysis of Algorithms***

***Semester :- IV***

Prepared By

Prof. K.M.Gajmal



## GHARDA FOUNDATION'S

### GHARDA INSTITUTE OF TECHNOLOGY

A/P:-LAVEL, TALUKA: KHED,DIST. RATNAGIRI, STATE:MAHARASTRA

University of Mumbai				
Class: S.E.	Branch: Computer Engineering	Semester: IV		
Subject: Analysis Of Algorithms (Abbreviated as AOA)				
Periods per Week (each 60 min)	Lecture	05		
	Practical	02		
	Tutorial	--		
		Hours	Marks	
Evaluation System	Theory	03	80	
	Practical and Oral	02	25	
	Oral	---	--	
	Term Work	---	25	
	Total	05	150	



## GHARDA FOUNDATION'S

### GHARDA INSTITUTE OF TECHNOLOGY

A/P:-LAVEL, TALUKA: KHED,DIST. RATNAGIRI, STATE:MAHARASTRA

#### List of Experiments

Sr. No.	Name of the Experiment
01.	To Analyze and implement the insertion sort algorithm.
02	Implementation and analysis of Binary Search Algorithm
03	Implementation and analysis of Single Source Shortest Path using Dynamic Programming
04	Write a program to find Longest Common Subsequence from the given two sequences.
05	Implementation and analysis of Knapsack Problem using greedy Approach.
06	Implementation and analysis of Optimal Storage on tapes
07	Implementation and analysis of sum of subsets problem
08	Implementation and analysis of Graph Coloring using backtracking
09	Implementation and analysis of the naïve string matching Algorithms
10	Implementation and analysis of Rabin Karp Algorithm
11	
12	



GHARDA FOUNDATION'S



## GHARDA INSTITUTE OF TECHNOLOGY

A/P:-LAVEL, TALUKA: KHED,DIST. RATNAGIRI, STATE:MAHARASTRA

---

### Experiment No.1

**Title:** To Analyze and implement the insertion sort algorithm.

#### Theory:

Insertion sort belongs to the  $O(n^2)$  sorting algorithms. Unlike many sorting algorithms with quadratic complexity, it is actually applied in practice for sorting small arrays of data. For instance, it is used to improve quick sort routine. Some sources notice, that people use same algorithm ordering items, for example, hand of cards.

Insertion sort algorithm somewhat resembles selection sort. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part contains **first element** of the array and unsorted one contains the rest. At every step, algorithm takes **first element** in the unsorted part and **inserts** it to the right place of the sorted one. When unsorted part becomes **empty**, algorithm *stops*.

Algorithm:

```
INSERTION-SORT(A)
  for i = 1 to n
    key ← A[i]
    j ← i - 1
    while j >= 0 and A[j] > key
      A[j+1] ← A[j]
      j ← j - 1
    End while
    A[j+1] ← key
  End for
```

#### *Complexity analysis*

##### Worst Case/Average Case:

Insertion sort's overall complexity is  $O(n^2)$ , regardless of the method of insertion.

##### Best Case:

On the almost sorted arrays insertion sort shows better performance, up to  $O(n)$  in case of applying insertion sort to a sorted array.  $N-1$  Comparisons but no moves of elements.

So complexity is  $O(n)$

Space Complexity:  $O(1)$

### ***Insertion sort properties***

- adaptive (performance adapts to the initial order of elements);
- stable (insertion sort retains relative order of the same elements);
- in-place (requires constant amount of additional space);
- online (new elements can be added during the sort).

## **Experiment No.2**

**Title:** Implementation and analysis of Binary Search Algorithm.

### **Theory:**

A binary search algorithm is a technique for finding a particular value in a linear array, by ruling out half of the data at each step, widely but not exclusively used in computer science.

A binary search finds the median, makes a comparison to determine whether the desired value comes before or after it, and then searches the remaining half in the same manner. Another explanation would be: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half.

### **Algorithm:**

**Binsearch(a,l,h,x)**

```
{
    If(l==x)
    {
        If(x==a[l]) return l else return 0;
    }
    Else
    {
        Mid=(l+h)/2;
        If(x==a[mid]) then return mid
        Else
        if(x<a[mid]) then return Binsearch(a,l,mid-1,x)
        else
        return Binsearch(a,mid+1,h,x)
    }
}
```

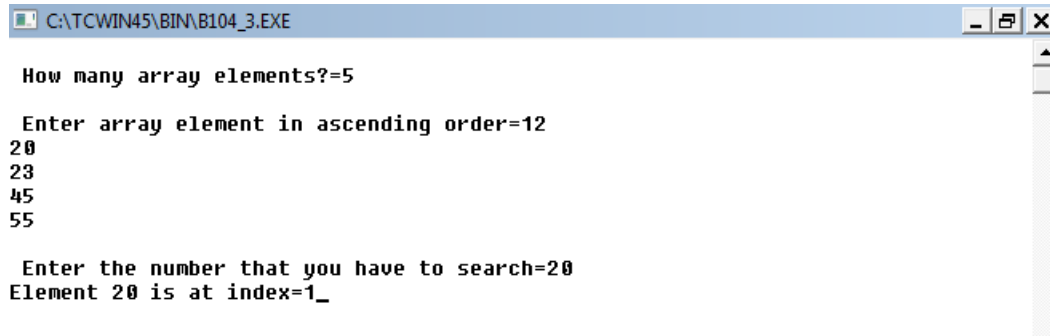
**Complexity Analysis:**

For Successful Search;

Best case:  $\Omega(1)$     Average Case:  $\theta(\log n)$     Worst Case:  $O(\log n)$

For Un Successful Search(All Cases):  $O(\log n)$

### Output:



```
C:\TCWIN45\BIN\B104_3.EXE

How many array elements?=5

Enter array element in ascending order=12
20
23
45
55

Enter the number that you have to search=20
Element 20 is at index=1_
```

**Conclusion:** We implemented the binary search algorithm to search the element present in the given list and the element not in the list.

## Experiment No.3

**Title:** Implementation and analysis of Single Source Shortest Path using Dynamic Programming

### Theory:

This is also called a Bellman-Ford algorithm where Given a graph and a source vertex *src* in graph, we have to find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges

### Algorithm

```
Algorithm Bellmanford(v,cost,dist,n)
{
for i=1 to n do
dist[i]=cost[v,i];
for k=2 to n-1 do
    for each u such that u <> v and u has at least one incoming edge do
        for each <i,u> in the graph do
            if dist[u]>dist[i]+cost[i,u] then
                dist[u]= dist[i]+cost[i,u]
}
```

### Complexity Annalysis:

**IEI \*IVI-1**

**Best case Time complexity:**  $O(|V|-1 * |E|) = O(n-1 * n) = O(n^2)$

**Worst Case for complete graph:**

$|E|=n(n-1)/2$  so :  $O(|V| |E|) = n(n-1)/2 * (n-1) = O(n^3)$

### Output

**BELLMAN FORD**

**Enter no. of vertices: 4**

**Enter graph in matrix form:**

**0 4 0 5**

**0 0 0 0**

**0 -10 0 0**

**0 0 3 0**

**Enter source: 1**

**Vertex 1 -> cost = 0 parent = 0**

**Vertex 2 -> cost = -2 parent = 3**

**Vertex 3 -> cost = 8 parent = 4**

**Vertex 4 -> cost = 5 parent = 1**

**No negative weight cycle**

## Experiment No.4

**Title:** Write a program to find Longest Common Subsequence from the given two sequences.

### Theory:

A Longest Common Subsequence is a common subsequence of maximum length. In the longest common subsequence problem we are given two sequences  $X = \langle x_1, x_2 \dots x_m \rangle$  and  $Y = \langle y_1, y_2 \dots y_n \rangle$  and wish to find a maximum-length common subsequence of  $x$  and  $y$ .

#### Computing the length of an LCS :


Procedure LCS-LENGTH takes two sequences  $X = \langle x_1, x_2 \dots x_m \rangle$  and  $Y = \langle y_1, y_2 \dots y_n \rangle$  as inputs. It stores the  $c[i,j]$  values in a table  $c[0 \dots m, 0 \dots n]$  whose entries are computed in row-major order i.e. the first row of  $c$  is filled in from left to right, then second row and so on. It also maintains the table  $b[1 \dots m, 1 \dots n]$  to simplify construction of an optimal solution. Initially,  $b[i,j]$  points to the table entry corresponding to the optimal sub problem solution. The procedure returns the  $b$  and  $c$  tables;  $c[m,n]$  contains the length of an LCS of  $X$  and  $Y$ .

#### Algorithm

```
LCS(A[m],B[n])
{
  for i=0 to m-1
    for j=0 to n-1
      if(A[i]=B[j])
        LCS[i,j]=1+LCS[i-1,j-1]
      Else
        LCS[i,j]=max(LCS[i-1,j], LCS[i,j-1])
}
```

#### Constructing an LCS : Bottom Up approach

The  $b$  table returned by LCS\_LENGTH can be used to quickly construct an LCS of  $X = \langle x_1, x_2 \dots x_m \rangle$  and  $Y = \langle y_1, y_2 \dots y_n \rangle$ . We begin at  $b[m, n]$  and trace through the table following the arrows.

Whenever we encounter a “” in entry  $b[i, j]$  it implies that  $x_i = y_j$  is an element of the LCS.

The elements of the LCS are encountered in reverse order by this method.

**Complexity:**  $O(m*n)$  WHERE  $m$ = Length of first string ,  $n$ = Length of second string



**Output:**

Enter 1st sequence:longest

Enter 2nd sequence:song

The Longest Common Subsequence is ong

-----

**Conclusion:**

In compilation of program, in software design or in system design text processing is a vital activity. While processing the text, string matching is an important activity which is needed most of the time. Least common subsequences is one of the pattern matching algorithm. This algorithm is more efficient than other algorithm and hence implemented successfully.

## Experiment No.5

**Title:** Implementation and analysis of Knapsack Problem using greedy Approach.

### Theory:

#### Problem Definition:

We are given a empty knapsack of capacity „W” and we are given „n” different objects from  $i=1,2,\dots,n$ . Each object „i” has some positive weight „ $w_i$ ” and some profit value is associated with each object which is denoted as „ $p_i$ ”.

We want to fill the knapsack with total capacity „W” such that profit earned is maximum. When we solve this problem main goal is :

1. Choose only those objects that give maximum profit.
2. The total weight of selected objects should be  $\leq W$

Most problem have n input and require us to obtain a subset that satisfies some constraints is called as a feasible solution. We are required to find a feasible solution that optimize (minimize or maximizes ) a given objective function. The feasible solution that does this is called an optimal solution.

#### Algorithm :

- 1 Let „W” be the maximum weight of the knapsack
- 2 Let „ $w_i$ ” and „ $p_i$ ” be the weight and profit of individual items i.e. for  $i=1,\dots,n$
- 3 Calculate  $p_i / w_i$  ratio and arrange that in decreasing order.
- 4 initially weight=0 and profit = 0
- 5 for  $i=1$  to  $n$   
{  
    add item in knapsack  
    till weight  $\leq W$   
    profit= profit +  $p_i$   
}  
6 Stop

#### Time Complexity

As main time taking step is sorting, the whole problem can be solved in  $O(n \log n)$  only.

## Algorithm

**Algorithm: Greedy-Fractional-Knapsack** ( $w[1..n]$ ,  $p[1..n]$ ,  $W$ )

```
for i = 1 to n
    do x[i] = 0
weight = 0
for i = 1 to n
    if weight + w[i] ≤ W then
        x[i] = 1
        weight = weight + w[i]
    else
        x[i] = (W - weight) / w[i]
        weight = W
        break
return x
```

## Program:

### Output:

Enter no of objets:

7

Enter profits of each object:

10 5 15 7 6 18 3

Enter weights of each object:

2 3 5 7 1 4 1

Enter capacity of knapsack:

15

Ratios of Profit and Weigh are:

Ratio1 1=5.000000

Ratio1 1=1.666667

Ratio1 1=3.000000

Ratio1 1=1.000000

Ratio1 1=6.000000

Ratio1 1=4.500000

Ratio1 1=3.000000

Maximum profit is :55.333332

-----

## Experiment No.6

**Title:** Implementation and analysis of Optimal Storage on tapes

### Theory:

Given  $n$  programs stored on a computer tape and length of each program  $i$  is  $L_i$  where  $1 \leq i \leq n$ , find the order in which the programs should be stored in the tape for which the Mean Retrieval Time (MRT given as  $\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^i L_j$ ) is minimized.

### Example:

Input :  $n = 3$

$L[] = \{ 5, 3, 10 \}$

Output : Order should be  $\{ 3, 5, 10 \}$  with  $MRT = 29/3$

**If there are  $n$  programs on tape as  $i=1$  to  $n$  then**

Let's suppose that the retrieval time of program  $i$  is  $T_i$ . Therefore,  $T_i = \sum_{j=1}^i L_j$

MRT is the average of all such  $T_i$ . Therefore  $MRT = \frac{1}{n} \sum_{i=1}^n T_i$ , or

$$MRT = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^i L_j$$

For e.g. Suppose there are 3 programs of lengths 2, 5 and 4 respectively. So there are total  $3! = 6$  possible orders of storage.

	ORDER	TOTAL RETRIEVAL TIME	MEAN RETRIEVAL TIME
1	1 2 3	$2 + (2 + 5) + (2 + 5 + 4) = 20$	$20/3$
2	1 3 2	$2 + (2 + 4) + (2 + 4 + 5) = 19$	$19/3$
3	2 1 3	$5 + (5 + 2) + (5 + 2 + 4) = 23$	$23/3$
4	2 3 1	$5 + (5 + 4) + (5 + 4 + 2) = 25$	$25/3$
5	3 1 2	$4 + (4 + 2) + (4 + 2 + 5) = 21$	$21/3$
6	3 2 1	$4 + (4 + 5) + (4 + 5 + 2) = 24$	$24/3$

It's clear that by following the second order in storing the programs, the mean retrieval time is least.

So, careful analysis suggests that in order to minimize the MRT, programs having greater lengths should be put towards the end so that the summation is reduced. Or, the lengths of the programs should be sorted in increasing order. That's the **Greedy Algorithm** in use.

**Time complexity is complexity of sorting elements which is  $O(n \log n)$ . If we sort with bubble sort complexity can be  $O(n^2)$**

## Algorithm

```
MRT(L[],n)
{
    Sort array of length in increasing order
    //print order
    For i=0 to n
        Printf L[i]

    //calculate MRT
    Mrt=0, sum=0
    float sum=0; //to save sum of tape

    for(i=0;i<n;i++)
    {
        sum=sum+l[i];
        mrt=mrt+sum;
    }
    mrt=mrt/n;
    print mrt.
}
```

## Program:

### Output

```
Enter no of programs
4
Enter length of program 1
3
Enter length of program 2
2
Enter length of program 3
8
Enter length of program 4
5
program 1    length 2
program 2    length 3
program 3    length 5
program 4    length 8
MRT of tape is 8.750000
```

## Experiment No.7

**Title:** Implementation and analysis of Sum Of Subsets problem

**Theory:**

**The subset-sum problem is to find a subset of a set of integers that sums to a given value. The decision problem of finding out if such a subset exists is NP-complete. One way of solving the problem is to use backtracking.**

### Backtracking

In Backtracking algorithm as we go down along depth of tree we add elements so far, and if the added sum is satisfying explicit constraints, we will continue to generate child nodes further. Whenever the constraints are not met, we stop further generation of sub-trees of that node, and backtrack to previous node to explore the nodes not yet explored. We need to explore the nodes along the breadth and depth of the tree. Generating nodes along breadth is controlled by loop and nodes along the depth are generated using recursion (post order traversal).

**Steps:**

1. Start with an empty set
2. Add the next element from the list to the set
3. If the subset is having sum M, then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set, then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible (sum of subset < M) then go to step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

### Algorithm

**sumOfSub (s,k,r)**

**//Find All subsets of w[1:n] that sum to m. The value of x[j], 1<=j<=k have already determined.**

$$S = \sum_{j=1}^{k-1} w[j] * x[j] \text{ and } r = \sum_{j=k}^n w[j]$$

```

{
    //Genarate Left Child
    S+w[k]<=m
    X[k]=1

    if(s+w[k]==m)then
    write (x[1:k])//subset found

    else
    if((s+w[k]+w[k+1]) <=m)
    {
        sumOfSub(s+w[k],k+1,r-w[k]);
    }
    //Generate right child and evaluate BK.
    if((s+r-w[k]) >=m && (s+w[k+1]) <=m)
    {
        x[k]=0;
        sumOfSub(s,k+1,r-w[k]);
    }
}

```

### Output:

Enter size of array: 6

Enter 6 elements:

5 10 12 13 15 18

Enter the sum to be obtained: 30

Possible Subsets are( 0 indicates exclusion and 1 indicates inclusion) :

1	1	0	0	1	
1	0	1	1		
0	0	1	0	0	1

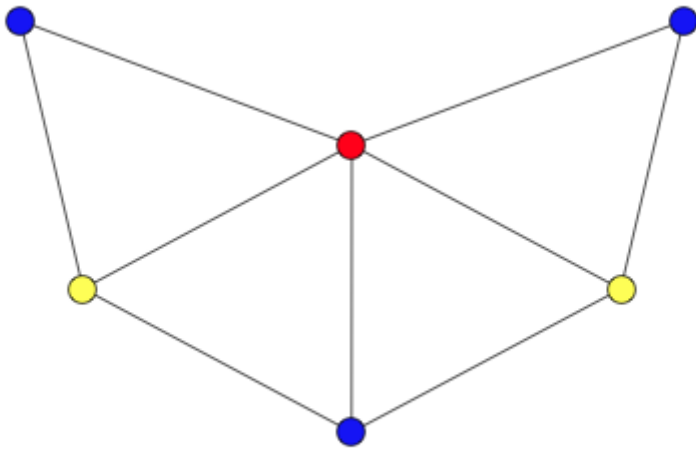
---

## Experiment No.8

Title: **Implementation and analysis of Graph coloring problem with Backtracking in C**

### Theory:

A **graph coloring** is an assignment of labels, called colors, to the vertices of a [graph](#) such that no two adjacent vertices share the same color. The **chromatic number** of a graph  $G$  is the *minimal* number of colors for which such an assignment is possible. Other types of colorings on graphs also exist, most notably **edge colorings** that may be subject to various constraints.



*A graph coloring for a graph with 6 vertices. It is impossible to color the graph with 2 colors, so the graph has chromatic number 3.*

### Applications of Graph Coloring

Some applications of graph coloring include –

- [Register Allocation](#)
- Map Coloring
- Bipartite Graph Checking
- [Mobile Radio Frequency Assignment](#)
- Making time table, etc.

### Algorithm

The steps required to color a graph  $G$  with  $n$  number of vertices are as follows –

**Step 1** – Arrange the vertices of the graph in some order.



**Step 2** – Choose the first vertex and color it with the first color.

**Step 3** – Choose the next vertex and color it with the lowest numbered color that has not been colored on any vertices adjacent to it. If all the adjacent vertices are colored with this color, assign a new color to it. Repeat this step until all the vertices are colored.

### Complexity

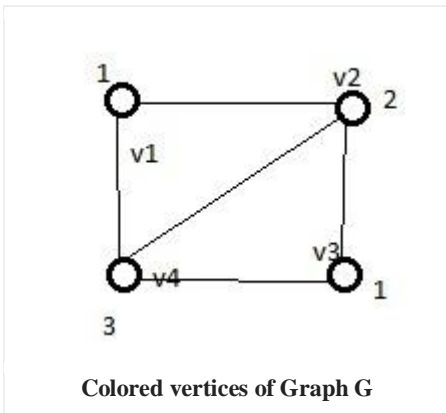
$O(V^2 + E)$  in worst case.(where V is the vertex and E is the edge).

---

### *OUTPUT*

---

Enter no. of vertices : 4



Enter no. of edges : 5

Enter indexes where value is 1-->

0 1

1 2

1 3

2 3

3 0

Colors of vertices -->

Vertex[1] : 1

Vertex[2] : 2

Vertex[3] : 1

Vertex[4] : 3

## Experiment No.9

### Title: Implementation and analysis of Naive algorithm for Pattern Searching

#### Theory:

Given a text  $txt[0..n-1]$  and a pattern  $pat[0..m-1]$ , write a function  $search(char\ pat[], char\ txt[])$  that prints all occurrences of  $pat[]$  in  $txt[]$ . You may assume that  $n > m$ .

```
Input:  txt[] = "THIS IS A TEST TEXT"
```

```
        pat[] = "TEST"
```

```
Output: Pattern found at index 10
```

```
Input:  txt[] = "AABAACAADAABAABA"
```

```
        pat[] = "AABA"
```

```
Output: Pattern found at index 0
```

```
        Pattern found at index 9
```

```
        Pattern found at index 12
```

#### Algorithm

```
naivePatternSearch(pattern, text)
```

**Input:** The text and the pattern

**Output:** location, where patterns are present in the text

Begin

```
    patLen := pattern Size
```

```
    strLen := string size
```

```
    for i := 0 to (strLen - patLen), do
```

```
        for j := 0 to patLen, do
```

```
            if text[i+j] ≠ pattern[j], then
```

```
                break the loop
```

```
        done
```

```
    if j == patLen, then
```

```
        display the position i, as there pattern found
```

```
    done
```

End

#### Complexity:

##### best case?

The best case occurs when the first character of the pattern is not present in text at all.

```
txt[] = "AABCCAADDEE";
```

```
pat[] = "FAA";
```

The number of comparisons in best case is  $O(n)$ .

### **worst case ?**

The worst case of Naive Pattern Searching occurs in following scenarios.

1) When all characters of the text and pattern are same.

```
txt[] = "AAAAAAAAAAAAAAAAAAAA";
```

```
pat[] = "AAAAA";
```

2) Worst case also occurs when only the last character is different.

```
txt[] = "AAAAAAAAAAAAAAAAAAB";
```

```
pat[] = "AAAAB";
```

The number of comparisons in the worst case is  $O(m*(n-m+1))$ . Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts).

### **Program:**

#### **Output:**

**Pattern found at index 0**

**Pattern found at index 9**

**Pattern found at index 13**

## Experiment No.10

### Title: Implementation and analysis of Rabin Carp for Pattern Searching

#### Theory:

**Rabin-Karp** is another pattern searching algorithm. It is the string matching algorithm that was proposed by Rabin and Karp to find the pattern in a more efficient way. Like the Naive Algorithm, it also checks the pattern by moving the window one by one, but without checking all characters for all cases, it finds the hash value. When the hash value is matched, then only it proceeds to check each character. In this way, there is only one comparison per text subsequence making it a more efficient algorithm for pattern searching.

Given a text  $txt[0..n-1]$  and a pattern  $pat[0..m-1]$ , write a function  $search(char\ pat[], char\ txt[])$  that prints all occurrences of  $pat[]$  in  $txt[]$ . You may assume that  $n > m$ .

#### Algorithm

`rabinkarp_algo(text, pattern, prime)`

**Input** – The main text and the pattern. Another prime number of find hash location

**Output** – locations, where the pattern is found

```
Start
    pat_len := pattern Length
    str_len := string Length
    patHash := 0 and strHash := 0, h := 1
    maxChar := total number of characters in character set
for index i of all character in the pattern, do
    h := (h*maxChar) mod prime
for all character index i of pattern, do
    patHash := (maxChar*patHash + pattern[i]) mod prime
    strHash := (maxChar*strHash + text[i]) mod prime
for i := 0 to (str_len - pat_len), do
    if patHash = strHash, then
        for charIndex := 0 to pat_len -1, do
            if text[i+charIndex] ≠ pattern[charIndex], then
                break
if charIndex = pat_len, then
    print the location i as pattern found at i position.
if i < (str_len - pat_len), then
    strHash := (maxChar*(strHash - text[i]*h)+text[i+patLen]) mod
prime, then
    if strHash < 0, then
        strHash := strHash + prime
End
```

**Complexity Analysis:**

*Preprocessing time-  $O(m)$*

The time complexity of the Rabin-Karp Algorithm is  **$O(m+n)$** , but for the worst case, it is  **$O(mn)$** .

**Program**

} **Output**

```
Pattern found at index 12
```

**Experiment No.11**

**Title: Implementation and analysis of Hamiltonian and Euler's Graph (Beyond Syllabus)**

**Theory:**