Department of Computer Science & Engineering, IGIT, GGSIPU Delhi

Lab manual
For
Algorithm Analysis & Design

(Version 1.0)

Even Semester – CSE Second Year
(January, 2011)

Prepared by:
Vivekanand Jha
Assistant Professor,
CSE Department, IGIT Delhi

Table of Contents

S. No.	Contents	Page No
1	**Objective**	3
2	**Prerequisites and execution mode**	4-5
3	**Schedule of practical course work**	6
4	**Practical assignments** Divide & Conquer Techniques ……... Disjoint Set Data Structure ……... Greedy Method ……... Dynamic Programming ……... String Matching ……...	 7-11 12-13 14-19 20-23 24-26
5	**Submission mode**	27

Course Objectives:

The lab course focuses on the practical implementation of various algorithms based on different algorithmic strategies. Lab work can improve the following skills:

- To design an implementation framework to algorithm,
- Approach to problem solving in context of existing algorithmic techniques,
- Understanding the fundamental & comparison of various algorithms efficiency

Pre-requisites

- **Windows/ Linux** platform operating knowledge.
- **C/C++** programming language.
- Theory course work on **Data Structures.**
- Theory course work on **Discrete Mathematics.**
- Theory course work on **Analysis & Design of Algorithms.**

Execution Mode:

- For the given problem statement design flowchart/Algorithm/Logic
- Define variables and functions, which will show the flow of program
- Write C/C++ code in the file
- Compile code
    - Using gcc compiler for Linux, create a.out executable file.
    - Using Windows platform, compile and run .c/.cpp file.
- Test the program using sample input and write down output
- Generate the graph based on various input versus obtained output
- Derive the efficiency of algorithm based on graph generated

Schedule :

S. No.	Deadlines	Algorithm Strategy/Area
1	Week 1,2,3,4	Divide & Conquer Techniques
2	Week 5	Disjoint Set Data Structure
3	Week 6,7,8,9,10	Greedy Method
4	Week 11,12,13	Dynamic Programming
5	Week 14 onwards	String Matching

Practical Assignment
on
Divide & Conquer Techniques

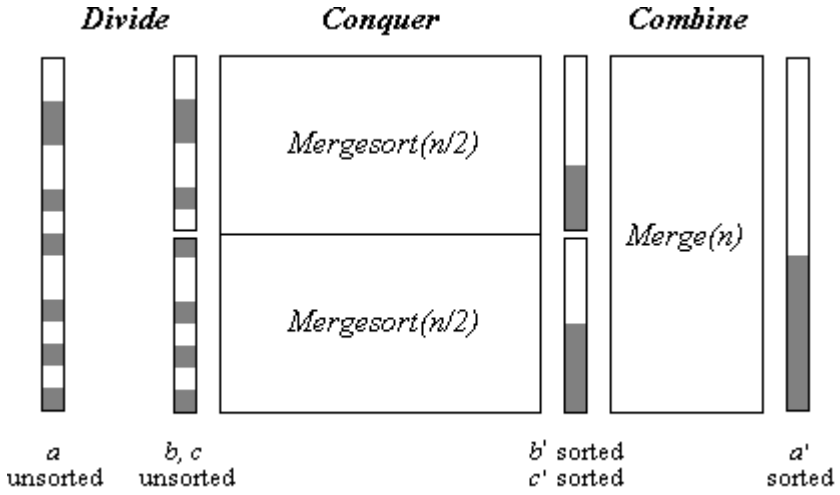
1. Write a program to implement Merge Sort.

The Mergesort algorithm is based on a divide and conquer strategy.

- First, the sequence to be sorted is decomposed into two halves (*Divide*).
- Each half is sorted independently (*Conquer*).
- Then the two sorted halves are merged to a sorted sequence (*Combine*)

To sort *A*[*p . . r*]:

Divide by splitting into two subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$, where q is the halfway point of $A[p \,..\, r]$.
Conquer by recursively sorting the two subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$.
Combine by merging the two sorted subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ to produce a single sorted subarray $A[p \,..\, r]$.



Input: **List having unsorted data.**

Output: **Input list in sorted form.**

2. Write a program to implement Quick Sort.

Quicksort is based on the three-step process of divide-and-conquer and problem is to sort the subarray $A[p \,..\, r]$:
Divide: Partition $A[p \,..\, r]$, into two (possibly empty) subarrays $A[p \,..\, q - 1]$ and $A[q + 1 \,..\, r]$, such that each element in the first subarray $A[p \,..\, q - 1]$ is $\leq A[q]$ and $A[q]$ is $\leq$ each element in the second subarray $A[q + 1 \,..\, r]$.

Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the subarrays, because they are sorted in place.

Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

Input: **List having unsorted data.**

Output: **Input list in sorted form.**

3. Write a program to implement Randomized Quick Sort.

It is assumed that all input permutations are equally likely in the case of quick sort. This is not always true.. To correct this, randomization to quick sort is added. Here randomly an element picked from the subarray that is being sorted.

Divide: Partition $A[p\,..\,r\,]$, into two (possibly empty) sub arrays $A[p\,..\,q-1]$ and $A[q+1\,..\,r\,]$, such that each element in the first subarray $A[p\,..\,q-1]$ is $\leq A[q]$ and $A[q]$ is $\leq$ each element in the second subarray $A[q+1\,..\,r\,]$.

Conquer: Sort the two sub arrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the sub arrays, because they are sorted in place

Perform the divide step by a procedure RANDOMIZED PARTITION, which returns the index q that marks the position separating the sub arrays

RANDOMIZED-PARTITION*(A, p, r)*

$i \leftarrow$RANDOM*(p, r)*

exchange $A[r\,] \leftrightarrow A[i\,]$

return PARTITION*(A, p, r)*

Randomization of quick sort stops any specific type of array from causing worst-case behavior. For example, an already-sorted array causes worst-case behavior in non-randomized QUICKSORT, but not in RANDOMIZED-QUICKSORT.

Input:	**List having unsorted data.**
Process:	**Must include random position of pivot element.**
Output:	**Input list in sorted form.**

4. Write a program to implement Strassen's matrix multiplication for NxN Matrix

Let *A*, *B* be two square matrix and problem is to calculate the matrix product *C* as

$$\mathbf{C} = \mathbf{AB} \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

If the matrices *A*, *B* are not of type 2^n x 2^n then fill the missing rows and columns with zeros.

If,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

with

$$\mathbf{A}_{i,j}, \mathbf{B}_{i,j}, \mathbf{C}_{i,j} \in R^{2^{n-1} \times 2^{n-1}}$$

then Strassens algorithm works like:

$$\begin{aligned}
\mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}$$

which are then used to express the $C_{i,j}$ in terms of M_k. Because of M_k it can eliminate one matrix multiplication and reduce the number of multiplications and express the $C_{i,j}$ as

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}$$

Input:	**Two matrices.**
Process:	**Strassen's Algorithm.**
Output:	**Product of Input matrices.**

Practical Assignment
on
Disjoint Set Data Structure

1.Write a program to implement Disjoint Set Data Structure

Properties:

- Maintain collection S = {*S*1, . . . , *Sk*} of disjoint dynamic (changing over time) sets.
- A representative identifies each set, which is some member of the set.

Basic operations:

- MAKE-SET*(x)*: make a new set *Si* = {*x*}, and add *Si* to S.
- UNION*(x, y)*: if $x \in Sx$, $y \in Sy$, then S ← S − *Sx* − *Sy* ∪ {*Sx* ∪ *Sy*}.
    - Representative of new set is any member of *Sx* ∪ *Sy*, often the representative of one of *Sx* and *Sy*.
    - Destroys *Sx* and *Sy* (since sets must be disjoint).

- FIND-SET*(x)*: return representative of set containing *x*.

Implementation by Linked List :

- Each set is a singly linked list.
- Each list node has fields for
    - the set member
    - pointer to the representative
    - next
- List has *head* (pointer to representative) and *tail*.
- MAKE-SET: create a singleton list.
- FIND-SET: return pointer to representative.
- UNION: a couple of ways to do it.

UNION*(x, y)*: append *x*.s list onto end of *y*.s list. Use *y*.s tail pointer to find the end. Need to update the representative pointer for every node on *x*.s list.

Input:	**Element (data) to create Set.**
Process:	**Perform basic operations by using link list.**
Output:	**Primitive operations result.**

Practical Assignment
on
Greedy Method

1. Write a program that implements Knapsack Problem.

The classic Knapsack problem is:

A thief breaks into a store and wants to fill his knapsack with as much value in goods as possible before making his escape. Given the following list of items available, what should he take?

- **Item A, weighing w_A pounds and valued at v_A**
- **Item B, weighing w_B pounds and valued at v_B**
- **Item C, weighing w_C pounds and valued at v_C**
- **.....**

Input

- Capacity K
- 'n' items with weights w_i and values v_i

Goal

- Output a set of items S such that
  - the sum of weights of items in S is at most K
  - and the sum of values of items in S is maximized

Procedure:

- P[1:n] and w[1:n] contain the profits and weights respectively f the n objects .
- N items are ordered such that p[i]/w[i]>=p[i+1]/w[i+1].
- M is the knapsack size and x[1:n] is the solution vector.

Input:	**N items having weights and profits.**
Process:	**Perform above procedure.**
Output:	**Selected item with associated fraction.**

2. Write a program that implements Huffman Coding.

Huffman coding is a technique used to compress files for transmission. Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix-free code that expresses the most common characters using shorter strings of bits than are used for less common source symbols

Problem description:

Given. A set of symbols and their costs.
Find. A prefix free binary character code (a sets of codewords) with minimum weighted path length.
Note-1. A code wherein each character is represented by a unique binary string (codeword) is called a binary character code.
Note-2. A prefix free code is a code having the property that no codeword is a prefix of any other codeword. Prefix code is generated by using Huffman Tree. A fast way to create a Huffman tree is to use the heap data structure.

Creating the tree:

1. Start with as many leaves as there are symbols.
2. Push all leaf nodes into the heap.
3. While there is more than one node in the heap:
    1. Remove two nodes with the lowest weight from the heap.
    2. If the heap was storing copies of node data rather than pointers to nodes in final storage for the tree, move these nodes to final storage.
    3. Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
    4. Update the parent links in the two just-removed nodes to point to the just-created parent node.
    5. Push the new node into the heap.
4. The remaining node is the root node; the tree has now been generated.

Input: Text file to be compressed

Process:

- Scan text to be compressed and tally occurrence of all characters.
- Sort or prioritize characters based on number of occurrences in text.
- Build Huffman code tree based on prioritized list.
- Perform a traversal of tree to determine all code words.
- Scan text again and create new file using the Huffman codes.

Output: Compressed text file and display compression factor

3. Write a program to implement Kruskal's Algorithm.

Kruskal's algorithm finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

Problem description:

- create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
- create a set *S* containing all the edges in the graph
- while *S* is nonempty and F is not yet spanning
    - remove an edge with minimum weight from *S*
    - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
    - otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

Input:	**Weighted graph (G).**
Process:	**Kruskal's Algorithm.**
Output:	**Minimum Cost Spanning Tree of G.**

4. Write a program to implement Prim's Algorithm.

Prim's algorithm finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Problem description:

The only spanning tree of the empty graph (with an empty vertex set) is again the empty graph. The following description assumes that this special case is handled separately.

The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices.

- Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).
- Initialize: $V_{new} = \{x\}$, where x is an arbitrary node (starting point) from V, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
    - Choose an edge (u, v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, any of them may be picked)
    - Add v to V_{new}, and (u, v) to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree

Input:	**Weighted connected graph (G).**
Process:	**Prim's Algorithm.**
Output:	**Minimum Cost Spanning Tree of G.**

5. Write a program to implement Bellman Ford Algorithm.

Bellman Ford algorithm works even in the case where edge weights may be negative, given a weighted, directed graph $G = (V;E)$ with source s. Returns TRUE if no negative-weight cycles reachable from *source*, FALSE otherwise.

Algorithm:

I. BELLMAN-FORD*(V, E,w, s)*

INIT-SINGLE-SOURCE*(V, s)*
for $i \leftarrow 1$ to $|V| - 1$
do for each edge $(u, v) \in E$
do RELAX*(u, v,w)*
for each edge $(u, v) \in E$
do if $d[v] > d[u] + w(u, v)$
then return FALSE
return TRUE

II. RELAX*(u, v,w)*

if $d[v] > d[u] + w(u, v)$
then $d[v] \leftarrow d[u] + w(u, v)$
$\pi[v] \leftarrow u$

III. INIT-SINGLE-SOURCE*(V, s)*

for each $v \in V$
do $d[v] \leftarrow \infty$
$\pi[v] \leftarrow$ NIL
$d[s] \leftarrow 0$

Input: **Weighted directed graph (G) & Source(S).**

Process: **Bellman Ford Algorithm.**

Output: **Shortest Path from S if no negative cycle.**

Practical Assignment
on
Dynamic Programming

1. Write a program to implement Matrix Chain Multiplication Algorithm.

Dynamic programming is a method of solving problems that exhibit the properties of overlapping subproblems and optimal substructure (described below). The method takes much less time than naive methods.

Matrix chain multiplication is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, we want to find the most efficient way to multiply these matrices together. The problem is not actually to *perform* the multiplications, but merely to decide in which order to perform the multiplications. We have many options because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same.
For example, if we had four matrices *A*, *B*, *C*, and *D*, we would have:

$(ABC)D = (AB)(CD) = A(BCD) = A(BC)D = ...$

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the *efficiency*. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

$(AB)C = (10\times30\times5) + (10\times5\times60) = 1500 + 3000 = 4500$ operations
$A(BC) = (30\times5\times60) + (10\times30\times60) = 9000 + 18000 = 27000$ operations

Clearly the first method is the more efficient. Now that we have identified the problem, how do we determine the optimal parenthesization of a product of *n* matrices? We could go through each possible parenthesization (brute force), but this would require time $O(2n)$, which is very slow and impractical for large *n*. The solution, as we will see, is to break up the problem into a set of related subproblems. By solving subproblems one time and reusing these solutions many times, we can drastically reduce the time required. This is known as dynamic programming.

In general, we can find the minimum cost using the following recursive algorithm:

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out each subsequence.
- Add these costs together, and add in the cost of multiplying the two result matrices.
- Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

Input:	**Chain of N matrices with dimensions.**
Process:	**Use dynamic programming.**
Output:	**Optimal parenthesization on chain of N input matrices.**

2. Write a program to implement Longest Common Subsequence Algorithm.

A string is a sequence of symbols over an alphabet set Σ. A subsequence of a string s is obtained by deleting zero or more symbols from s. The longest common subsequence (LCS) problem for strings is to find a common subsequence having maximum length. For example, if $S1$ = abcacba and $S2$ = aabbccbbaa,abccba is a LCS for these two strings. The *LCS* can be found by dynamic programming formulation.

LCS Procedure

Let two sequences be defined as follows: $X = (x_1, x_2...x_m)$ and $Y = (y_1, y_2...y_n)$. The prefixes of X are $X_{1, 2,...m}$; the prefixes of Y are $Y_{1, 2,...n}$. Let $LCS(X_i, Y_j)$ represent the set of longest common subsequence of prefixes X_i and Y_j. This set of sequences is given by the following.

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ (LCS(X_{i-1}, Y_{j-1}), x_i) & \text{if } x_i = y_j \\ \text{longest}(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j \end{cases}$$

To find the longest subsequences common to X_i and Y_j, compare the elements x_i and y_i. If they are equal, then the sequence $LCS(X_{i-1}, Y_{j-1})$ is extended by that element, x_i. If they are not equal, then the longer of the two sequences, $LCS(X_i, Y_{j-1})$, and $LCS(X_{i-1}, Y_j)$, is retained. (If they are both the same length, but not identical, then both are retained.) Notice that the subscripts are reduced by 1 in these formulas. That can result in a subscript of 0. Since the sequence elements are defined to start at 1, it was necessary to add the requirement that the LCS is empty when a subscript is zero.

Input:	**Two strings P and T.**
Process:	**Use dynamic programming approach.**
Output:	**LCS of P and T**

3. Write a program to implement All Pair Shortest Path Algorithm.

Given a weighted digraph G = (V,E), determine the length of the shortest path (i.e., distance) between all pairs of vertices in G. Assume that there are no cycles with zero or negative cost. In the *all pairs shortest path* problem, we want to find the shortest path from *every* possible source to *every* possible destination.
As usual for dynamic programming algorithms, we first need to come up with a recursive formulation of the problem. So define *dist*(*u*, *v*, *k*) to be the length of the shortest path from *u* to *v* that uses *at most k edges*. Therefore, it can be defined shortestPath(*i*, *j*, *k*) in terms of the following recursive formula:

$$\text{shortestPath}(i,j,k) = \begin{cases} \text{edgeCost}(i,j) & \text{if } k = 0 \\ \min(\text{shortestPath}(i,j,k-1), & \\ \quad \text{shortestPath}(i,k,k-1) + \text{shortestPath}(k,j,k-1)) & \text{else} \end{cases}$$

DYNAMICPROGRAMMINGAPSP(*V*, *E*,*w*):
    for all vertices *u from V*
        for all vertices *v from V*
            if *u* = *v*
                dist[*u*, *v*, 0] 0
            else
                dist[*u*, *v*, 0] 1
    for *k* 1 to *V*-1
        for all vertices *u from V*
            for all vertices *v from V*
                dist[*u*, *v*, *k*] = infinite
                for all vertices *x from V*
                dist[*u*, *v*, *k*] = min {*dist*[*u*, *v*, *k*], *dist*[*u*, *x*, *k*-1]+ *w*(*x*->*v*)}

Input: **Weighted Graph G in N X N matrix.**

Process: **Floyd–Warshall algorithm.**

Output: **N Matrices for each intermediate node.**

Practical Assignment
on
String Matching

1. Write a program to implement Naïve String Matching Algorithm.

Naïve algorithm

- Match string character by character.
- When there is a mismatch, shift the whole string down by one character against the text, and start again at the beginning of the string

The naïve approach simply test all the possible placement of Pattern $P[1 . . m]$ relative to text $T[1 . . n]$. Specifically, we try shift $s = 0, 1, . . . , n - m$, successively and for each shift, s. Compare $T[s +1 . . s + m]$ to $P[1 . . m]$.

```
NAÏVE_STRING_MATCHER (T, P)
        n ← length [T]
        m ← length [P]
        for s ← 0 to n - m do
                if P[1 . . m] = T[s +1 . . s + m]
                then return valid shift s
```

Input:	**Text String (T) and Pattern String (P).**
Process:	**Successive search.**
Output:	**Shift (S), the position(s) of the pattern string.**

2. Write a program to implement Rabin-Karp String Matching Algorithm.

The Rabin–Karp algorithm seeks to speed up the testing of equality of the pattern to the substrings in the text by using a hash function. A hash function is a function which converts every string into a numeric value, called its hash value; for example, we might have hash("hello")=5. Rabin–Karp exploits the fact that if two strings are equal, their hash values are also equal. Thus compute the hash value of the substring we're searching for, and then look for a substring with the same hash value.

```
RabinKarp:
Input  an array of characters, S, length n
       an array of characters sub, length m
   hsub := hash(sub[1..m])
   hs := hash(s[1..m])
   for i from 1 to n-m+1
      if hs = hsub
         if s[i..i+m-1] = sub
            return i
      hs := hash(s[i+1..i+m])
   return not found
```

To be helpful for the string matching problem an hashing function *hash* should have the following properties:

- efficiently computable
- highly discriminating for strings
- *hash*(*y*[*j*+1 .. *j*+*m*]) must be easily computable from *hash*(*y*[*j* .. *j*+*m*-1]) and *y*[*j*+*m*]:
  hash(*y*[*j*+1 .. *j*+*m*])= *rehash*(*y*[*j*], *y*[*j*+*m*], *hash*(*y*[*j* .. *j*+*m*-1]).

For a word *w* of length *m* let *hash*(*w*) be defined as follows:
hash(*w*[0 .. *m*-1])=(*w*[0]*2^{m-1}+ *w*[1]*2^{m-2}+⋯+ *w*[*m*-1]*2^{0}) mod *q*, *q* is a large number.

Then, *rehash*(*a*,*b*,*h*)= ((*h*-*a**2^{m-1})*2+*b*) mod *q*

The preprocessing phase of the Karp-Rabin algorithm consists in computing *hash*(*x*). During searching phase, it is enough to compare *hash*(*x*) with *hash*(*y*[*j* .. *j*+*m*-1]) for 0 ⩽ *j* < *n*-*m*. If an equality is found, it is still necessary to check the equality *x*=*y*[*j* .. *j*+*m*-1] character by character.

Input: **Text String (T) and Pattern String (P).**

Process: **Rabin-karp algorithm.**

Output: **All successful hits position**

Submission Mode

1. Handwritten file.

2. One side of page to be used for writing.

3. File content format is:
    (i) File heading
    (ii) Table of Contents
    (iii) Algorithm
    (iv) Program
    (v) Input
    (vi) Output
    (vii) Graph (if any)