MGM's College of Engineering and Technology
Kamothe, Navi Mumbai
Department of Computer Engineering

LAB MANUAL OF

Analysis of Algorithms Lab

Code: CSL401

Class: SE Computer Engineering

Semester: IV (CBCGS)

Dr. Anil Kale
Subject Incharge

Program Structure B.E. Computer Engineering, (Rev. 2016) w.e.f. AY 2017-18

S. E. Computer Engineering (Semester-IV)

CourseCode	CourseNa me	Teaching Scheme (Contact Hours)			Credits Assigned			
		Theory	Pract	Tut	Theory	TW/ Pract	Tut	Total
CSC401	Applied Mathematics- IV	4+1@	-	-	5	-	-	5
CSC402	Analysis of Algorithms	4	-	-	4	-	-	4
CSC403	Computer Organization andArchitecture	4	-	-	4	-	-	4
CSC404	Computer Graphics	4	-	-	4	-	-	4
CSC405	Operating System	4	-	-	4	-	-	4
CSL401	Analysis of Algorithms Lab	-	2	-	-	1	-	1
CSL402	Computer Graphics Lab	-	2	-	-	1	-	1
CSL403	Processor Architecture Lab	-	2	-		1	-	1
CSL404	Operating System Lab	-	2	-	-	1	-	1
CSL405	Open Source Tech Lab	-	2+2*	-	-	2	-	2
	Total	**21**	**12**	-	**21**	**6**	-	**27**

@ 1 hour to be taken tutorial as class wise .

*2 hours shown as Practical"s to be taken class wise and other 2 hours to be taken as batch wise

Course Code	Course Name	Examination Scheme								
		Theory					TW	Oral	Oral & Pract	Total
		Internal Assessment			End Sem. Exam	Exam Duration (in Hrs)				
		Test 1	Test 2	Avg.						
CSC401	Applied Mathematics- IV	20	20	20	80	3	-	-	-	100
CSC402	Analysis of Algorithms	20	20	20	80	3	-	-	-	100
CSC403	Computer Organization and Architecture	20	20	20	80	3	-	-	-	100
CSC404	Computer Graphics	20	20	20	80	3	-	-	-	100
CSC405	Operating System	20	20	20	80	3	--	-	-	100
CSL401	Analysis of Algorithms Lab	-	-	-	-	-	25	--	25	50
CSL402	Computer Graphics Lab	-	-	-	-	-	25	--	25	50
CSL403	Processor Architecture Lab	-	-	-	-	-	25	25	-	50
CSL404	Operating System Lab	-	-	-	-	-	25	-	25	50
CSL405	Open Source Tech Lab	-	-	-	-	-	25	---	25	50
Total		100	100	100	400	-	125	25	100	750

Lab Code	Lab Name	Credit
CSL401	**Analysis of Algorithms Lab**	**1**

Lab outcomes: At the end of the course student will be able to

1. Analyze the complexities of various problems in different domains.
2. Prove the correctness and analyze the running time of the basic algorithms for those classic problems in various domains.
3. Develop the efficient algorithms for the new problem with suitable designing techniques.
4. Implement the algorithms using different strategies.

Prerequisites: Students should be familiar with concepts of Data structure and Discrete structures.

Description:

Minimum 2 experiments should be implemented using any language on each algorithm design strategy (Divide and conquer, dynamic programming, Greedy method, backtracking and branch & bound, string matching).

Suggested Laboratory Experiments:

Sr. No.	Module Name	Suggested Experiment List
1	Introduction to analysis of algorithm Divide and Conquer Approach	Selection sort , insertion sort. Merge sort, Quick sort, Binary search.
2	Dynamic Programming Approach	Multistage graphs, single source shortest path, all pair shortest path, 0/1 knapsack, Travelling salesman problem, Longest common subsequence.
3	Greedy Method Approach	Single source shortest path, Knapsack problem, Job sequencing with deadlines, Minimum cost spanning trees-Kruskal and prim‟s algorithm, Optimal storage on tapes.
4	Backtracking and Branch-and-bound	8 queen problem (N-queen problem), Sum of subsets, Graph coloring, 15 puzzle problem, Travelling salesman problem.

5	String Matching Algorithms	The naïve string matching Algorithms, The Rabin Karp algorithm, String matching with finite automata, The knuth-Morris-Pratt algorithm.
6	Any two Experiments	This will involve implementation of two algorithms for problems beyond the scope of syllabus. The exact set of algorithms to implement is to be decided by the course instructor.

Text Books:

1. T.H.Coreman , C.E. Leiserson,R.L. Rivest, and C. Stein, "Introduction to algorithms", 2nd edition , PHI publication 2005.
2. Ellis horowitz , sartaj Sahni , s. Rajsekaran. "Fundamentals of computer algorithms" University Press

Reference Books:

1. Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani, "Algorithms", Tata McGraw- Hill Edition.
2. S. K. Basu, "Design Methods and Analysis of Algorithm", PHI.
3. Dana Vrajittoru and William Knight, "Practical Analysis of Algorithms", Springer 2014th Edition.

Term Work:

Laboratory work must contain implementation of minimum 10 experiments. The final certification and acceptance of term work ensures the satisfactory performance of laboratory work and minimum passing marks in term work. The 25 marks of the term work should be divided as below:
25 Marks (total marks) = 15 Marks Lab. Experiments + 05 Marks Assignments (based on theory syllabus) + 05 (Attendance: theory + practical)

Oral & Practical Exam will be based on the experiments implemented in the Laboratory.

List of Experiments

Sr. No.	Title
1	Implement selection and insertion sort and perform comparative analysis in terms time complexity
2	Write a program to implement Quick sort and Merge sort and study their complexities.
3	Write a program to implement Single pair shortest path using Dijkstra's Shortest Path Algorithm and its analysis.
4	Write a program to implement Fractional Knapsack problem using greedy approach and its analysis.
5	Write a program to implement Minimum cost Spanning tree using Prim‟s algorithm and its analysis.
6	Write a program to implement Minimum cost Spanning tree using Kruskal's algorithm and its analysis.
7	Write a program to implement LCS problem using dynamic programming
8	Write a program to implement All Pair Shortest Path Floyd-Warshall Algorithm
9	Write a program to implement N-Queen Problem using Backtracking and its analysis.
10	Write a program to implement the Rabin Karp string matching algorithms and its analysis.
11	Write a program to implement Graph Coloring Algorithm

Experiment No: 01

Aim: Program to implement selection sort & Insertion sort.

Theory:

Selection Sort

A selection sort is one in which successive elements are selected in order and placed in their proper sorted positions. The elements of the input array may have to be preprocessed to make the ordered selection possible. This algorithm is called the general selection sort.

Straight Selection Sort or push down sort implements the descending priority queueas an unordered array. Therefore, the straight selection sort consists entirely of a selection phase in which the largest of the remaining elements large is repeatedly placed in its proper position i.e. the end of the array. To do so, large is interchanged with the elementx[i].

Algorithm:

```
Algorithm: Selection-Sort (A)
fori ← 1 to n-1 do
   min j ← i;
   min x ← A[i]
   for j ←i + 1 to n do
      if A[j] < min x then
         min j ← j
         min x ← A[j]
   A[min j] ← A [i]
   A[i] ← min x
```

Analysis of Selection Sort:

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takes n-1comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n1elements and so on,
For (n-1) + (n-2) + ... + 2 + 1 = n(n-1) / 2 ∈ $\Theta(n^2)$ comparisons. Each of these scans requires one swap for n-1 elements (the final element is already in place).

Insertion Sort:

An insertion sort is one that sorts a set of records by inserting records into an existing sorted file. The simplest way to insert next element into the sorted part is to sift it down, until it occupies correct position. Initially the element stays right after the sorted part. At each step algorithm compares the element with one before it and, if they stay in reversed order, swap them.

Algorithm:

```
Algorithm: Insertion-Sort(A)
for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i -1
    A[i + 1] = key
```

Analysis of Insertion Sort:

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., Θ(n)), During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e.,$O(n^2)$). The average case is also quadratic, which makes insertion sort impractical for sorting large arrays.

Conclusion:

Insertion sort is very similar in that after the kth iteration, the first k elements in the array are in sorted order. Insertion sort's advantage is that it only scans as many elements as it needs in order to place the k + 1st element, while selection sort must scan all remaining elements to find the k + 1stelement. Insertion sort is one of the fastest algorithms for sorting very small arrays. Insertion sort typically makes fewer comparisons than selection sort.

Experiment No: 02

Aim: Implementation of Merge Sort using Divide –and-Conquer

Theory:

Merge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with sub problems, we state each sub problem as sorting a sub array $A[p .. r]$. Initially, $p = 1$ and $r = n$, but these values change as we recourse through sub problems.

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

Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p .. r]$ into two sub arrays $A[p .. q]$ and $A[q + 1 .. r]$, each containing about half of the elements of $A[p .. r]$. That is, q is the halfway point of $A[p .. r]$.

Conquer Step

Conquer by recursively sorting the two sub arrays $A[p .. q]$ and $A[q + 1 .. r]$.

Combine Step

Combine the elements back in $A[p .. r]$ by merging the two sorted sub arrays $A[p .. q]$ and $A[q + 1 .. r]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).
Note that the recursion bottoms out when the sub array has just one element, so that it is triviallysorted.

Alg.: MERGE-SORT(A, p, r)

```
if p < r                          Check for base case
  then q ← ⌊(p + r)/2⌋             Divide
       MERGE-SORT(A, p, q)         Conquer
       MERGE-SORT(A, q + 1, r)     Conquer
       MERGE(A, p, q, r)           Combine
```

- Initial call: MERGE-SORT(A, 1, n)

Alg.: MERGE(A, p, q, r)

1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1 \ldots n_1 + 1]$ and the next n_2 elements into $R[1 \ldots n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[i] \leq R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$

ANALYSIS

The straightforward version of function *merge* requires at most $2n$ steps (n steps for copying the sequence to the intermediate array *b*, and at most n steps for copying it back to array *a*). The time complexity of *mergesort* is therefore

$$T(n) \leq 2n + 2\,T(n/2) \text{ and}$$
$$T(1) = 0$$

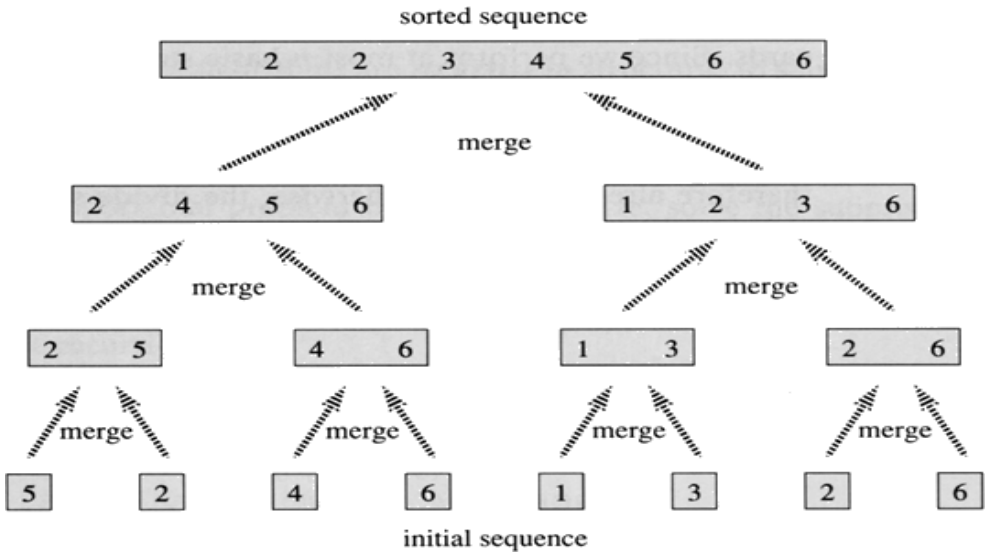
The solution of this recursion yields

$$T(n) \leq 2n \log(n) \in O(n \log(n))$$

Thus, the Mergesort algorithm is optimal, since the lower bound for the sorting problem of $\Omega(n\log(n))$ is attained.

In the more efficient variant, function *merge* requires at most $1.5n$ steps ($n/2$ steps for copying the first half of the sequence to the intermediate array *b*, $n/2$ steps for copying it back to array *a*, and at most $n/2$ steps for processing the second half). This yields a running time of mergesort of at most $1.5n \log(n)$ steps. Algorithm Mergesort has a time complexity of $\Theta(n\log(n))$ which isoptimal.
A drawback of Mergesort is that it needs an additional space of $\Theta(n)$ for the temporary array *b*.

Example: Bottom-up view of the above procedure for $n = 8$.



Merging

What remains is the MERGE procedure. The following is the input and output of the ERGE procedure.

> **INPUT**: Array A and indices p, q, r such that $p \le q \le$ r and sub array $A[p \,..\, q]$ is sorted and sub array $A[q + 1 \,..\, r]$ is sorted. By restrictions on p, q, r, neither subarray is empty.
> **OUTPUT**: The two sub arrays are merged into a single sorted sub array in $A[p \,..\, r]$.

We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1$, which is the number of elements being merged.

Conclusion: Thus we implemented Merge Sort which is having a time complexity is O(n log n).

Aim: Implementation of Quick Sort using Divide-and-Conquer Strategy

Theory:

Quick sort is a divide and conquer algorithm which relies on a partition operation: to partition an array, an element, called a pivot is chosen, all smaller elements are moved before the pivot, and all greater elements are moved after it. This can be done efficiently in linear time and in-place. Then recursively sorting can be done for the lesser and greater sub lists. Efficientimplementations of quick sort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, this makes quick sort one of the most popular sorting algorithms, available in many standard libraries. The most complex issue in quick sort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower ($O(n^2)$) performance, but if at each step we choose the median as the pivot then it works in O(n log n).
Quick sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

Pick an element, called a pivot, from the list.
Reorder the list so that all elements which are less than pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

ANALYSIS

The partition routine examines every item in the array at most once, so complexity is clearly $O(n)$. Usually, the partition routine will divide the problem into two roughly equal sized partitions. We know that we can divide n items in half $\log_2 n$ times. This makes quicksort a $O(n \log n)$ algorithm - equivalent to heapsort.

Conclusion: Thus we implemented quick sort which is fastest algorithm and worst case time complexity is $O(N^2)$

Experiment No: 03

Aim: Implementation of Single source shortest path using Dijkstra algorithm and its analysis.

Theory:

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

DIJKSTRA (G, w, s)

```
1.  INITIALIZE SINGLE-SOURCE (G, s)
2.  S ← { }     // S will ultimately contains vertices of final shortest-
     path weights from s
3.  Initialize priority queue Q i.e., Q ←  V[G]
4.  while priority queue Q is not empty do
5.      u  ←  EXTRACT_MIN(Q)    // Pull out new  vertex
6.      S  ←  S È {u}                    // Perform relaxation for each vertex  v adjacent to u
7.      for each vertex v in Adj[u] do
8.          Relax (u, v, w)
```

ANALYSIS

Like Prim's algorithm, Dijkstra's algorithm runs in O(|E|lg|V|) time.

Conclusion: Thus we implemented the Single source shortest path using Dijkstra algorithm.

Experiment No: 04

Aim: To implement Fractional knapsack algorithm using dynamic programming.

Theory: Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.
Problem Scenario
A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of ith item is wi and its profit is pi. What items should the thief take?
In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.
Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack
In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.
According to the problem statement,

- There are n items in the store
- Weight of $\mathbf{i^{th}}$ item wi>0wi>0
- Profit for $\mathbf{i^{th}}$ item pi>0pi>0 and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction xi of $\mathbf{i^{th}}$ item.

$$0 \leqslant xi \leqslant 10 \leqslant xi \leqslant 1$$

The $\mathbf{i^{th}}$ item contributes the weight xi.wixi.wi to the total weight in the knapsack and profit xi.pixi.pi to the total profit.
Hence, the objective of this algorithm is to subject to constraint,

$$\sum n=1n(xi.wi) \leqslant W \sum n=1n(xi.wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.
Thus, an optimal solution can be obtained by

$$\sum n=1n(xi.wi)=W \sum n=1n(xi.wi)=W$$

In this context, first we need to sort those items according to the value of piwipiwi, so that pi+1wi+1pi+1wi+1 ≤ piwipiwi . Here, x is an array to store the fraction of items.

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
```

Analysis

If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn).***

Conclusion: The fractional knapsack problem is implemented using dynamic programming, which is used to fill the knapsack with items to get maximum benefit (value or profit) without crossing the weight capacity of the knapsack. And we are also allowed to take an item in fractional part.

Experiment No: 05

Aim: Implementation of Minimum Spanning Tree using Prims Algorithm.

Theory:

Prim‟s algorithm is a greedy algorithm that is used to find a minimum spanning tree (MST) of a given connected weighted graph. This algorithm is preferred when the graph is dense. The dense graph is a graph in which there is a large number of edges in the graph. This algorithm can be applied to only undirected connected graph and there should not be any negative weighted edge. In this case, the algorithm is quite efficient. Since there are no nonnegative weight cycles, there will be a shortest path whenever there is a path.

The steps to find minimum spanning tree using Prim‟s algorithm are as follows:

1. If graph has loops and parallel edges than remove loops and parallel edges of that graph.
2. Randomly choose any node, labelling it with a distance of 0 and all other nodes as ∞. The chosen node is treated as current node and considered as visited. All other nodes are considered as unvisited.
3. Identify all the unvisited nodes that are presently connected to the current node. Calculate the distance from the unvisited nodes to the current node.
4. Label each of the vertices with their corresponding weight to the current node, but relabel of a node, if it is less than the previous value of the label. Each time, the nodes are labelled with its weights; keep track of the path with the smallest weight.
5. Mark the current node as visited by colouring over it. Once a vertex is visited, we not need to look at it again.
6. From all the unvisited nodes, find out the node which has minimum weight to the current node, consider this node as visited and treat it as the current working node.
7. Repeat steps 3, 4 and 5 until all nodes are visited.
8. After completed all steps get desired MST.

Algorithm:

```
/* S= set of visited node, Q= Queue, G=Graph, w=Weight */
Prim_MST (G, w, S)
{
   Initialization (G,S)
   S ←Ø // the set of visited nodes is initially empty
  Q← v[G] // The queue is initially contain all nodes
  while (Q= Ø) // Queue not empty
       do u← extract_min(Q) // select the minimum distance of Q
       S ←SU{u} // the u is add in visited set S
  For each vertex v ϵ adj(u)
  do relax(u, v, w)
}
```

```
Initialization (G, S)
{
     For each vertex v ϵ adj (u)
     d [v] ←∞ //Unknown Distance from source node
    π[v] ←nil // Predecessor node initially nil.
    d[s] ←0 //Distance of source nodes is zero
}

Relax (u, v,w)
{
    If d[v]>w[u,v] // comparing new distance with existing value
    {
       d[v] ← w[u,v]
       π[u] ←u
    }
}
```

Analysis of Prim's Algorithm:

Finding the minimum distance is O (V) and overall complexity with adjacency list representation is O (V2).
If queue is kept as a binary heap, relax will need a decrease-key operation which is O (logV) and the overall complexity is O (V log V + E log V) i.e. O (E log V).
If queue is kept as a Fibonacci heap, decrease-key has an amortized complexity O (1) and the overall complexity is O (E+V log V).

Conclusion: Thus we implemented the minimum spanning tree using Prim‟s algorithm.

Experiment No: 06

Aim: Implementation of Minimum Spanning Tree using Kruskal Algorithm.

Theory: Kruskal's Algorithm

Kruskal‟s algorithm is the following: first, sort the edges in increasing (or rather non decreasing)order of costs, so that c(e1) ≤ c(e2) ≤ . . . ≤ c(em); then, starting with an initially empty tree T, go through the edges one at a time, putting an edge in T if it will not cause a cycle, but throwing the edge out if it would cause a cycle.

Below are the steps for finding MST using Kruskal‟s algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning treeformed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Algorithm:

MST-Kruskal(G, w)

1. A ← ∅ // initially A is empty
2. for each vertex v ∈ V[G] // line 2-3 takes O(V) time
3. do Create-Set(v) // create set for each vertex
4. sort the edges of E by non decreasing weight w
5. for each edge (u, v) ∈ E, in order by non decreasing weight
6. do if Find-Set(u) ≠ Find-Set(v) // u&v on different trees
7. then A ← A ∪ {(u, v)}
8. Union(u,v)
9. return A

Analysis:

We can easily make each of them run in time O(E lgV) using ordinary binary heaps.
By using Fibonacci heaps, This algorithm runs in time O(E+ V lgV), which improves over thebinary-heap implementation if |V| is much smaller than |E|.

Conclusion: Thus we implemented the minimum spanning tree using Kruskal‟salgorithm.

Experiment No: 07

Aim: To implement LCS problem using dynamic programming approach.

Theory:

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.

```
Algorithm: LCS-Length-Table-Formulation (X, Y)
m :=
length(X
)n :=
length(Y
)
for i = 1
   to m do
   C[i, 0]
   := 0
for j = 1
   to n do
   C[0, j]
   := 0
for i = 1 to m
   do for j =
   1 to n do
      if xi = yj
         C[i, j] := C[i - 1, j -
         1] + 1B[i, j] := 'D'
      else
         if C[i -1, j] ≥ C[i, j -1]
            C[i, j] := C[i - 1, j] + 1
            B[i, j] := 'U'
         else
         C[i, j] := C[i, j - 1]
         B[i, j] := 'L'
return C and B
Algorithm: Print-LCS (B, X, i, j)
if i = 0 and
   j = 0
   return
if B[i, j] = 'D'
   Print-LCS(B, X, i-1, j-1)
   Print(xi)
else if B[i, j] = 'U'
   Print-LCS(B, X,
i-1, j)else
   Print-LCS(B, X, i, j-1)
```

Conclusion:

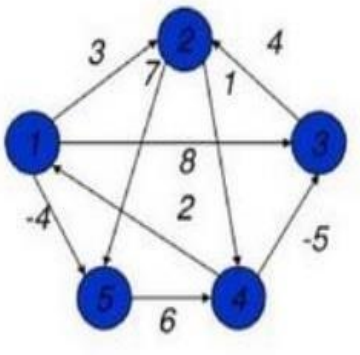
To populate the table, the outer **for** loop iterates ***m*** times and the inner **for** loop iterates ***n*** times.Hence, the complexity of the algorithm is *O(m, n),* where ***m*** and ***n*** are the length of two strings.

Experiment No: 08

Aim: Implement All Pair Shortest Path Algorithm

Theory:

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.

Input − The cost matrix of the graph.

```
0 3 6 ∞ ∞ ∞ ∞
3 0 2 1 ∞ ∞ ∞
6 2 0 1 4 2 ∞
∞ 1 1 0 2 ∞ 4
∞ ∞ 4 2 0 2 1
∞ ∞ 2 ∞ 2 0 1
∞ ∞ ∞ 4 1 1 0
```

Output − Matrix of all pair shortest path.

```
0 3 4 5 6 7 7
3 0 2 1 3 4 4
4 2 0 1 3 2 3
5 1 1 0 2 3 3
6 3 3 2 0 2 1
7 4 2 3 2 0 1
7 4 3 3 1 1 0
```

Algorithm:

floydWarshal(cost)
Input − The cost matrix of given Graph.
Output − Matrix to for shortest path between any vertex to any vertex.

```
Begin
   for k := 0 to n, do
      for i := 0 to n, do
         for j := 0 to n, do
            if cost[i,k] + cost[k,j] < cost[i,j], then
               cost[i,j] := cost[i,k] + cost[k,j]
            done
         done
      done
      display the current cost matrix
End
```

Analysis:
The time complexity of this algorithm is $O(V^3)$, here V is the number of vertices in the graph.

Conclusion: Thus we implemented all pair shortest path algorithm.

Experiment No: 09

Aim: To implement N queen problems using backtracking

Theory:

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example;

The **eight queens puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general ***n*-queens problem** of placing *n* queens on an *n*×*n*chessboard, where solutions exist for all natural numbers *n* with the exception of *n*=2 and*n*=3

A way to place all n queens on the board such that no queens are attacking another queen. Using backtracking, this algorithm prints all possible placements of n queens on an n*n chessboard so that they are non-attacking. Place (k, i) – Returns true if a queen can be placed in k^{th} row and i^{th} column. Otherwise it returns false. X [] is a Global array whose first (k –1) values have been set. Abs(r) returnsthe absolute value of r.

Algorithm:

```
Algorithm NQueens( k, n) //Prints all Solution to the n-queens problem
{
    For i := 1 to n do
    {   if Place (k, i) then
              {      x[k] := i;
                       if ( k = n) then write ( x [1 : n] else NQueens(k+1,n);
               }
         }
}
Algorithm Place (k,i)
{
for j := 1 to k-1 do
if (( x[ j ] = // in the same column
or (Abs( x [ j ] - i) =Abs ( j – k ))) // or in the same
diagonalthen return false;
return true;
}
```

Conclusion:

- Backtracking provides the hope to solve some problem instances of nontrivial sizes bypruning non-promising branches of the state-spacetree.
- The success of backtracking varies from problem to problem and from instance toinstance.
- Backtracking possibly generates all possible candidates in an exponentially growingstate-spacetree.

Experiment No: 10

Aim: Implement Rabin Karp string matching algorithm

Theory:

Rabin Karp string matching algorithm makes use of elementary number-theoretic notion of modulo arithmetic.
Given a pattern P[1..m], let p denote its corresponding decimal value. Similarly, given a text T[1..n],let ts denote the decimal value of the length m substring T[s+1..s+m], for s = 0,1,..,n-m.
Certaintly, ts = p if and only if T[s+1].s+m] = P[1..m]; thus, is a valid shift if and only if ts=p.

Algorithm

RABIN-KARP-MATCHER(*T, P, d, q*)

```
1   n   length[T]
2   m   length[P] 3h        d^m -1 mod q
3   p   0
4   t0  0
5   for i   1 to m
6       do p   (dp+ P[i]) mod q
7       t0     (dt0 + T[i]) mod q
8   for s   0 to n -m
9       do if p =ts
10      then if P[1 . . m] = T[s + 1 . . s +m]
11      then "Pattern occurs with shift"s
12      if s <n -m
13      then ts+1  (d(ts- T[s + 1]h) + T[s + m + 1]) modq
```

Running time: Preprocessing time: O(m), matching time: worst case is O((n-m+1)m), while averagewill be lesser.

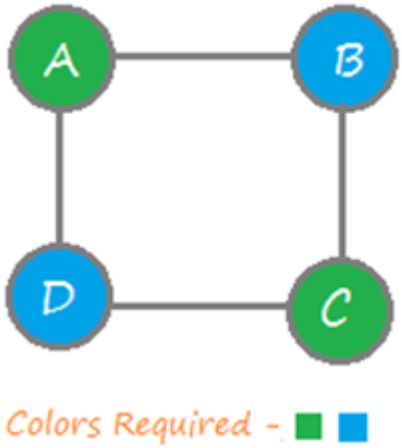
Conclusion:
The Rabin karp works on arithmetic operations to match pattern in text and is much faster than Naïvestring matching algorithm.

Experiment No: 11

Aim: Implement Graph Coloring Algorithm
What is Graph Coloring Problem?

Theory

For a given graph if it is asked to color all vertices with the 'M' number of given colors, insuch a way that no two adjacent vertices should have the same color.



It it is possible to color all the vertices with the given colors then we have to output the colored result, otherwise output 'no solution possible'.

The least possible value of 'm' required to color the graph successfully is known asthe ***chromatic number of the given graph****.*

It it is possible to color all the vertices with the given colors then we have to output the colored result, otherwise output 'no solution possible'.

The least possible value of 'm' required to color the graph successfully is known asthe ***chromatic number of the given graph****.*

Graph Coloring Solution
Using Naive Algorithm

In this approach using the brute force method, we find all permutations of color combinations that can color the graph.

If any of the permutations is valid for the given graph and colors, we output the resultotherwise not. This method is not efficient in terms of time complexity because it finds all colors combinations rather than a single solution.

Using Backtracking Algorithm

The backtracking algorithm makes the process efficient by avoiding many bad decisions made in naïve approaches.

In this approach, we color a single vertex and then move to its adjacent (connected) vertex to color it with different color.

After coloring, we again move to another adjacent vertex that is uncolored and repeat the process until all vertices of the given graph are colored.

In case, we find a vertex that has all adjacent vertices colored and no color is left to make it color different, we backtrack and change the color of the last colored vertices and again proceed further.

If by backtracking, we come back to the same vertex from where we started and all colors were tried on it, then it means the given number of colors (i.e. 'm') is insufficient to color the given graph and we require more colors (i.e. a bigger chromatic number).

Steps To color graph using the Backtracking Algorithm:

1. Different colors:
   A. Confirm whether it is valid to color the current vertex with the current color (bychecking whether any of its adjacent vertices are colored with the same color).
   B. If yes then color it and otherwise try a different color.
   C. Check if all vertices are colored or not.
   D. If not then move to the next adjacent uncolored vertex.
2. If no other color is available then backtrack (i.e. un-color last colored vertex). *Here **backtracking means to stop further recursive calls** on adjacent vertices by returning false. In this algorithm Step-1.2 (Continue) and Step-2 (backtracking) iscausing the program to try different color option.*
   ***Continue** – try a different color for current vertex.*
   ***Backtrack** – try a different color for last colored vertex.*

Analysis:

- **Time Complexity:** O(m^V).
  There are total O(m^V) combination of colors. So time complexity is O(m^V). The upperboundtime complexity remains the same but the average time taken will be less.
- **Space Complexity:** O(V).
  Recursive Stack of graphColoring(…) function will require O(V) space.

Conclusion:
Understood the application of Graph Coloring Problem in many practical areas such as pattern matching, designing seating plans, scheduling exam time table, solving sudoku puzzles etc.