MEGHNAD SAHA INSTITUTE OF TECHNOLOGY

DESIGN AND ANALYSIS OF ALGORITHM LAB

LABORATORY MANUAL

Subject Code : PCC CS494

Year :2nd Semester :4th

Department of Computer Science &Engineering

Nazirabad Rd. , Uchhepota , Kolkata , South 24 Parganas Pincode-700150 West Bengal, India

DEPARTMENT OFCOMPUTER SCIENCE AND ENGINEERING

Vision

- **To attain a global platform in academics, research and innovation by preparing competent computer engineers to cater for the needs of industry and society at large.**

Mission

- **To address the dynamic & growing needs of the software industry by creating quality professionals with a strong focus on principles of Computer Science and Engineering.**
- **To provide state-of-the-art infrastructure to facilitate the research work for enhancing the knowledge in emerging technologies including machine learning models, data science, cybersecurity, and IoT etc.**
- **To strengthen the industry-academic relationship through collaboration with global IT organizations, healthcare units and relevant institutions for data sharing and developing technology.**
- **To nurture the students by inculcating the spirit of ethical and social values through creating strong foundation in ethical coding and computational design paradigms.**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAMME EDUCATIONAL OBJECTIVES(PEOs)

PEO	PROGRAMME EDUCATIONAL OBJECTIVES
PEO1	To prepare graduates who will be successful software professionals, academicians, researchers related to computational frameworks, and entrepreneurial pursuit.
PEO2	To prepare graduates who will achieve peer recognition and adapt to the new technological environment as an individual or in a team demonstrating good computational, analytical, model designing, and software implementation skills.
PEO3	To prepare graduates who will thrive to pursue life-long learning for keeping pace with dynamic technological changes, through the development of an intuitive computational learning paradigm.
PEO4	To foster the values of professional and societal ethics for generating responsible citizens with proven expertise in computational domain.

PO	PROGRAM OUTCOME DETAILS
PO 1	**Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the Solution of complex engineering problems.
PO 2	**Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics ,natural sciences ,and engineering sciences.
PO 3	**Design / development of solutions**: Design solutions for complex engineering problems and design system components or processes thatmeetthespecifiedneedswithappropriateconsiderationforthepublichealthand safety, and the cultural, societal, and environmental considerations.
PO 4	**Conduct investigations of complex problems :**Use research-based knowledge and research methods including design of experiments ,analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO 5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources ,and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO 6	**The engineer and society:** Apply reasoning in formed by the contextual knowledge to assessorial l, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO 7	**Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of ,and need for sustainable development.
PO 8	**Ethics:** Apply ethical principles and commit to professional ethics and Responsibilities and norms of the engineering practice.

PO 9	**Individual and team work:** Function effectively as an individual, and as a Member or leader in diverse teams ,and in multidisciplinary settings.
PO 10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation ,make effective presentations, and give and receive clear instructions.
PO 11	**Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage Projects and in multidisciplinary environments.
PO 12	**Life-longlearning:** Recognize the need for ,and have the preparation and ability to engage in independent and life-long learning in the broadest Context of technological change.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM SPECIFIC OUTCOMES(PSOs)

After the completion of the course, B.Tech Computer Science and Engineering, the graduates will Have the following Program Specific Outcomes:

PSO	PROGRAMSPECIFICOUTCOMESDETAILS
PSO1	**Apply the skills of basic science, discrete mathematics, probability, principles of electrical and electronics, and programming aptitude to develop the self-learning capabilities for the modelling and designing of computing systems in a way of solving engineering problems.**
PSO2	**Evaluate the solution framework of complex engineering problems by applying algorithms, tools, and techniques related to computer science to identify the optimal solutions for enriching computational research and development.**
PSO3	**Strengthen the industry-institute partnership and socio-economic framework by enhancing the emerging areas of computer science & engineering for the growing needs of society with computational solutions.**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to the starting time), those who come after 5minutes will not be allowed in the lab.
2. Plan your task properly much before to the commencement, come prepared to the lab with the synopsis/program/experiment details.
3. Students should enter in the laboratory with:
   a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm procedure, Program Expected Output, etc.,) filled in for the lab session.
   b. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.
   c. Proper Dress code and Identity card.
4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.
5. Execute your task in the laboratory, and record the results/output in the lab Observation notebook, and get certified by the concerned faculty.
6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.
7. Students/Faculty must keep the mobile phones in SWITCHED OFF mode During the lab sessions. Misuse of the equipment, misbehavior With the staff and systems etc. will act severe punishment.
8. Students must take the permission of the faculty in case of any urgency to go out ;if anybody found out side the lab / class without permission during working hours will be treated seriously and punished appropriately.
9. Students should LOGOFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects.

Head of the Department **Principal**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DO's

1. Follow the rules and guidelines set by the department for computer lab usage.
2. Follow the instructions provided by the Technical Assistant and Faculty members.
3. Use the equipment and software appropriately for educational purposes only.
4. Respect the equipment and take care of it. Do not tamper with or damage the computers, keyboards, mouse, and other equipment.
5. Keep the computer and the lab clean and organized.
6. Be respectful of other users and their privacy while inside the lab.
7. Log out of your computer account and shut down the computer properly before leaving the lab.
8. Ask for help if you need it. Don't be afraid to seek assistance from Technical Assistants or Faculty.
9. Report any technical issues to the technical staff or System Administrator.

DON'Ts

1. Don't use the lab for non-educational purposes such as playing games or browsing social media.
2. Don't enter the lab with your Bag &amp; wearing shoes (only mobile and money purse allowed)
3. Don't download or install any unauthorized software or files on the computer.
4. Don't alter or modify the computer hardware or software in any way.
5. Don't eat or drink inside the computer lab to avoid spilling or damaging equipment.
6. Don't leave your personal belongings unattended in the lab.
7. Don't modify, delete or copy files that belong to other students or the lab.
8. Do not engage in any form of cyberbullying or harassment.
9. Do not use your personal devices in the lab without permission from the lab administrator or instructor.

9. Don't disturb other students by making excessive noise or talking loudly.

10. Do not share or copy copyrighted material.

11. Don't insert any kind of external device onto the computer.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

University Syllabus

Maulana Abul Kalam Azad University of Technology, West Bengal
(Formerly West Bengal University of Technology)
Syllabus for B.Tech in Computer Science &Engineering
(Applicable from the academic session 2020-2021)

Course Name: Design & Analysis of Algorithm Lab	
Code: PCC-CS494	Credit Points: 2
Contacts : 12P (4 hrs./week)	
Teaching Scheme : Continuous Assessment	
Full Marks : 100	
Internal Assessment : 40	External Assessment :60

Laboratory Experiments :

1.1	Write a program to implement Binary Search using Divide and Conquer approach.
1.2	Write a program to implement Merge Sort using Divide and Conquer approach..
1.3	Write a program to Implement Quick Sort using Divide and Conquer approach.
1.4	Write a program to find Maximum and Minimum element from an array of integer using Divide and Conquer approach.
2.1	Write a program to find the minimum number of scalar multiplication needed for chain of matrix using Dynamic Programming approach.
2.2	Write a program to implement all pair of Shortest path for a graph (Floyed- Warshall Algorithm) using Dynamic Programming approach.
2.3	Write a program to implement Traveling Salesman Problem using Dynamic Programming approach.
2.4	Write a program to implement Single Source shortest Path for a graph (Dijkstra , Bellman Ford Algorithm) using Dynamic Programming approach.
3.1	Write a program to implement 15 Puzzle Problem
4.1	Write a program to implement 8 Queen problem.
4.2	Write a program to implement Graph Coloring Problem.
4.3	Write a program to implement Hamiltonian Problem.
5.1	Write a program to implement Knapsack Problem.
5.2	Write a program to implement Job sequencing with deadlines.
5.3	Write a program to implement Minimum Cost Spanning Tree by Prim's Algorithm.
5.4	Write a program to implement Minimum Cost Spanning Tree by Kruskal's Algorithm.
6.1	Write a program to implement Breadth First Search (BFS).
6.2	Write a program to implement Depth First Search (DFS).

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

RECOMMENDEDSYSTEM/SOFTWAREREQUIREMENTS:

1. Intel based desktop PC of 166MHz or faster processor with at least 64 MB RAM and100MB free disk space.
2. Turbo C compiler or GCC compilers or Online GDB Compiler

USEFULTEXTBOOKS/REFERECES:

1. Introduction to Algorithms, 4TH Edition, Thomas H Cormen, Charles E Lieserson, Ronald L Rivest and Clifford Stein, MIT Press/McGraw-Hill.

2. Fundamentals of Algorithms – E. Horowitz et al.

3. Algorithm Design, 1ST Edition, Jon Kleinberg and Éva Tardos, Pearson.

4. Algorithm Design: Foundations, Analysis, and Internet Examples, Second Edition, Michael T Goodrich and Roberto Tamassia, Wiley.

5. Algorithms -- A Creative Approach, 3RD Edition, UdiManber, Addison-Wesley, Reading, MA

6. Design & Analysis of Algorithms, Gajendra Sharma, Khanna Publishing House (AICTE Recommended Textbook – 2018)

7. Algorithms Design and Analysis, Udit Agarwal, Dhanpat Rai

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

<u>Assessment Methods of University Lab Examination</u>

Internal Assessment	**40Marks**
Practical Assessment	**60marks**
Total	**100Marks**

Internal Assessment(40Marks)

PCA1	**PCA2**
Lab Report and Viva (25)	**Lab Report and Viva (25)**
Lab Conduct Session (15)	**Lab Conduct Session (15)**

Practical Assessment(60Marks)

Viva	**20Marks**
Experiment and Result	**40Marks**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ASSIGNMENT LIST

EXPERIMENT / ASSIGNMENT NO (WITH NAME)

Introduction:

An algorithm is a set of steps defined for solving a problem in finite time and space. Generally an algorithm is written to express the method of problem solving and it may include mathematical notations.
As an algorithm is expressed in a language, developers can implement the logic in any programming language. If you are familiar with Data Structure, you know that algorithm is very important in implementing Data Structures.

Algorithmic Strategies:

Divide and Conquer	There are many problems, which can be divided into sub-problems and the sub-problems can be solved individually and finally the solutions of individual problems can be merged together to form the final result. Examples include the merge sort, quicksort algorithms etc.
Dynamic Programming	This is an optimization technique. When sub-problems need to be solved repetitively, the solutions of the sub-problems are stored in a table to reduce time consumption. In future, when the result of the subproblems are needed, instead of solving the sub-problem the stored solutions are used.
Branch and Bound	This is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores *branches* of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated *bounds* on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm. The algorithm depends on efficient estimation of the lower and upper bounds of regions/branches of the search space. If no bounds are available, the algorithm degenerates to an exhaustive search.
Backtracking	**This** is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.
Greedy Approach	A greedy algorithm is useful when sufficient information is known about possible choices. The best choice can be determined without considering all possible choices. Thus this approach may not give best solution. But, the advantage of this approach is that the result can be obtained in less time.
Graph Traversal	Graph traversal is a technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops.

Prerequisites:

The readers should have a basic understanding of C programming language and Data Structure.

1.1 Write a program to implement Binary Search using Divide and Conquer approach.

I) OBJECTIVE

Search an element from a sorted array.

II) THEORY

Binary Search is the simplest example of **Divide and Conquer approach**. If a set of n elements is given where the numbers are sorted either in ascending or descending order, this algorithm can be used to search an element in the given set.

The idea behind this algorithm is that to find an element in the given array, first we find the element at the middle of the array. If the array is in ascending order and the middle element of the array is lesser than the element we want to search. The element can not be at the left-hand side of the middle element. So the search space is reduced by nearly 50%.

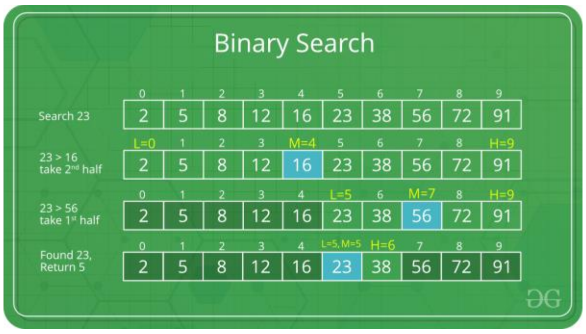
In the next step, we try to find the element in one half of the array and the other half is rejected. In this step we find the middle element of the respective half of the array and the element is compared with the key, we want to search.

This process is followed, until we find the element or size of the searching space becomes one.

III) Algorithm for the Experiment / Assignment

```
Input:  A, array of elements n,
        size of the array x,
        the key to search
Output: ind, index of element x
        ind = 0
        low = 1
        high = n
        while low <= high do
           mid = (low + high) / 2
           if x < A[mid] then
              high = mid - 1
           else if
              low = mid + 1
           else
              ind = mid
              return
           end
        end while
```

IV) RESULT / OUTPUT / OBSERVATION



V) DISCUSSION

Time Complexity:

The time complexity of Binary Search can be written as

$T(n) = T(n/2) + c$

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

1.2 Write a program to implement Merge Sort using Divide and Conquer approach.

I) OBJECTIVE

Sorting a set of elements of an array.

II) THEORY

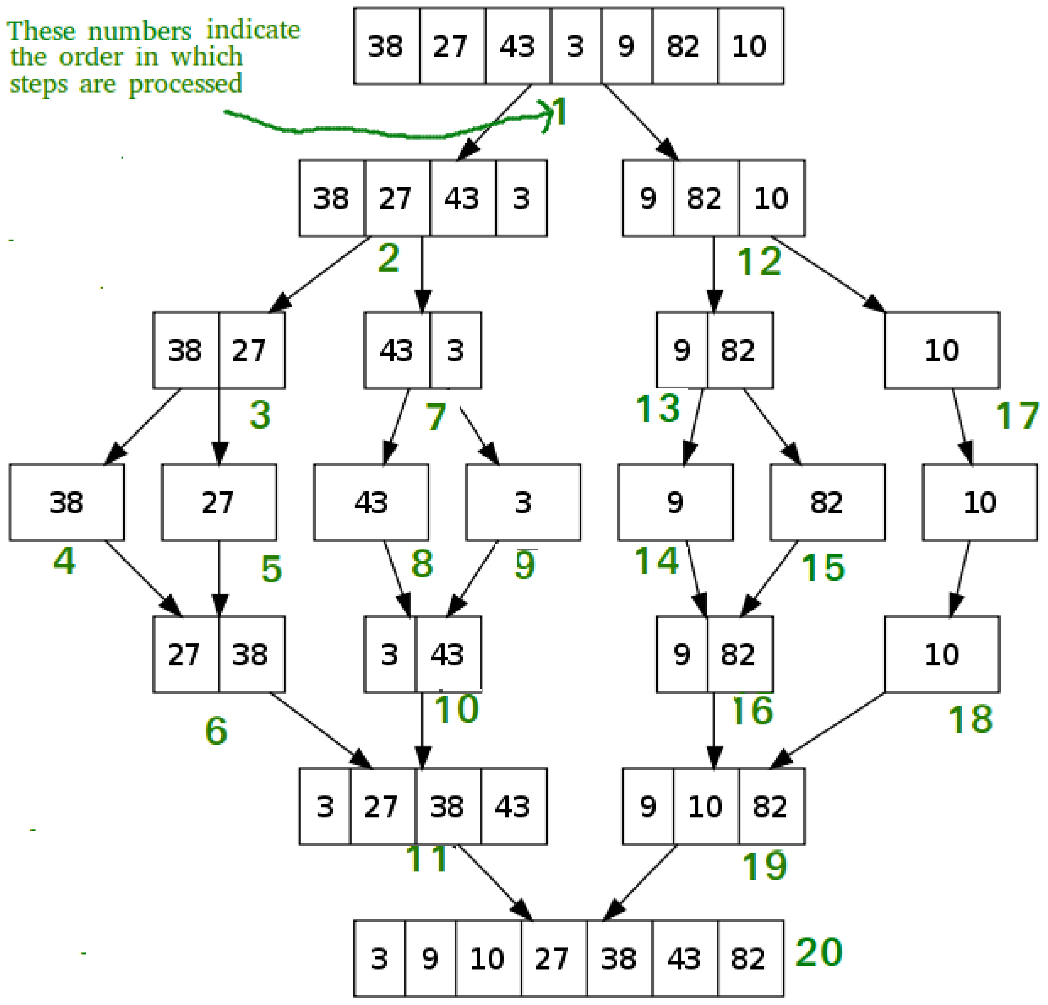
Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(A, p, q, r) is key process that assumes that arr[p..q] and arr[q+1..r] are sorted and merges the two sorted sub-arrays into one. See following for details.

III) Algorithm for the Experiment / Assignment

```
MergeSort(array A, int p, int r) {
if (p < r) { // we have at least 2 items
q = (p + r)/2
MergeSort(A, p, q) // sort A[p..q]
MergeSort(A, q+1, r) // sort A[q+1..r]
Merge(A, p, q, r) // merge everything together
}
}
Merge(array A, int p, int q, int r) { // merges A[p..q] with A[q+1..r]  in array B[p..r]
i = k = p // initialize pointers
j = q+1
while (i <= q and j <= r) { // while both subarrays are nonempty
if (A[i] <= A[j]) B[k++] = A[i++] // copy from left subarray
else B[k++] = A[j++] // copy from right subarray
}
while (i <= q) B[k++] = A[i++] // copy any leftover to B
while (j <= r) B[k++] = A[j++]
for i = p to r do A[i] = B[i] // copy B back to A
}
```

IV) RESULT / OUTPUT / OBSERVATION

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



V) DISCUSSION

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

T(n) = 2T(n/2) + T(n)

Time complexity of Merge Sort is in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

Auxiliary Space: O(n)

Applications of Merge Sort

Merge Sort is useful for sorting linked lists in **O(n Logn)** time. In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike an array, in the linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

1.3 Write a program to implement Quick Sort using Divide and Conquer approach.

I) OBJECTIVE

Sorting a set of elements of an array.

II) THEORY

Quicksort is a divide-and-conquer sorting algorithm in which division is dynamically carried out (as opposed to static division in Mergesort).

The three steps of Quicksort are as follows:

Divide: Rearrange the elements and split the array into two subarrays and an element in between such that so that each element in the left subarray is less than or equal the middle element and each element in the right subarray is greater than the middle element.
Conquer: Recursively sort the two subarrays.
Combine: None.

III) Algorithm for the Experiment / Assignment

```
Quicksort(A, n)
1: Quicksort0(A, 1, n)
Quicksort0(A, p, r)
1: if p ≥r then return
2: q = Partition(A, p, r)
3: Quicksort0(A, p, q - 1)
4: Quicksort0(A, q +1, r)
```

<u>The subroutine Partition</u>

Given a subarray A[p .. r] such that $p \le r - 1$, this subroutine rearranges the input subarray into two subarrays, A[p .. q - 1] and A[q +1 .. r], so that

- each element in A[p .. q - 1] is less than or equal to A[q] and
- each element in A[q +1 .. r] is greater than or equal to A[q]

Then the subroutine outputs the value of q.
Use the initial value of A[r] as the “pivot,” in the sense that the keys are compared against it. Scan the keys A[p .. r - 1] from left to right and flush to the left all the keys that are greater than or equal to the pivot.

<u>The Algorithm</u>

```
Partition(A, p, r)
    1: x = A[r]
    2: i ← p - 1
    3: for j ← p to r - 1 do
    4: if A[j] ≤ x then {
    5:              i ← i+1
    6:              Exchange A[i] and A[j] }
    7: Exchange A[i+1] and A[r]
    8: return i+1
```

During the for-loop i+1 is the position at which the next key that is greater than or equal to the pivot should go to.

<u>Proving Correctness of Partition</u>

Let (A, p, r) be any input to Partition and let q be the output of Partition on this input.
Suppose $1 \le p < r$. Let x = A[r]. We will prove the correctness using loop invariant. The loop invariant we use is: at the beginning of the for-loop, for all k, $p \le k \le r$, the following properties hold:
1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i+1 \le k \le j - 1$, then $A[k] > x$.
3. If k = r, then A[k] = x.

IV) RESULT / OUTPUT / OBSERVATION

Initialization

The initial value of i is p - 1 and the initial value of j is p. So, there is no k such $p \le k \le i$ and there is no k such that $i+1 \le k \le j-1$.

Thus, the first conditions are met. The initial value of A[r] = x, is so the last one is met.

Maintenance

Suppose that the three conditions are met at the beginning and that $j \le r - 1$.

Suppose that A[j] > x. The value of i will not be changed, so (1) holds.

The value of j becomes j +1. Since A[j] > x, (2) will for the new value of j.

Also, A[r] is unchanged so (3) holds.

Suppose that $A[j] \le x$. Then A[i+1] and A[j] will be exchanged. By (2), A[i+1] > x.

So, after exchange $A[i+1] \le x$ and A[j] > x. Both i and j will be incremented by 1, so (1) and (2) will be preserved. Again (3) still holds.

Termination

At the end, j = r. So, for all k, $1 \le k \le i$, $A[k] \le x$ and for all k, $i+1 \le k \le r - 1$, A[k] > x.

V)DISCUSSION

Running Time Analysis

The running time of quicksort is a linear function of the array size, r - p+1, and the distance of q from p, q - p. This is Θ(r - p+1).

What are the worst cases of this algorithm?

Worst-case analysis

Let T be the worst-case running time of Quicksort. Then

$$T(n) = T(1)+T(n - 1)+ \Omega(n).$$

By unrolling the recursion we have

$$T(n) = nT(1)+ \Omega(\textstyle\sum_{i=2}^{n} n).$$

Since T(1) = O(1), we have

$$T(n) = \Omega(n^2):$$

Thus, we have:

Theorem A The worst-case running time of Quicksort is $\Omega(n^2)$.

Since each element belongs to a region in which Partition is carried out at most n times, we have:

Theorem B The worst-case running time of Quicksort is $O(n^2)$.

The Best Cases

The best cases are when the array is split half and half. Then each element belongs to a region in which Partition is carried out at most $\lceil \log n \rceil$ times, so it's O(n log n).

1.4 Write a program to find Maximum and Minimum element from an array of integer using Divide and Conquer approach.

I) OBJECTIVE

Find Maximum and Minimum element from an array of integers.

II) THEORY

MaxMin is a recursive algorithm to find the largest and smallest elements from a set of elements. Let us consider an array of n numbers {a[0], a[1],...,a[n-1]}. Hence, at time of finding the largest and the smallest element of the array, the largest and the smallest elements are kept in max and min variables respectively. The algorithm is shown below:

III) Algorithm for the Experiment / Assignment

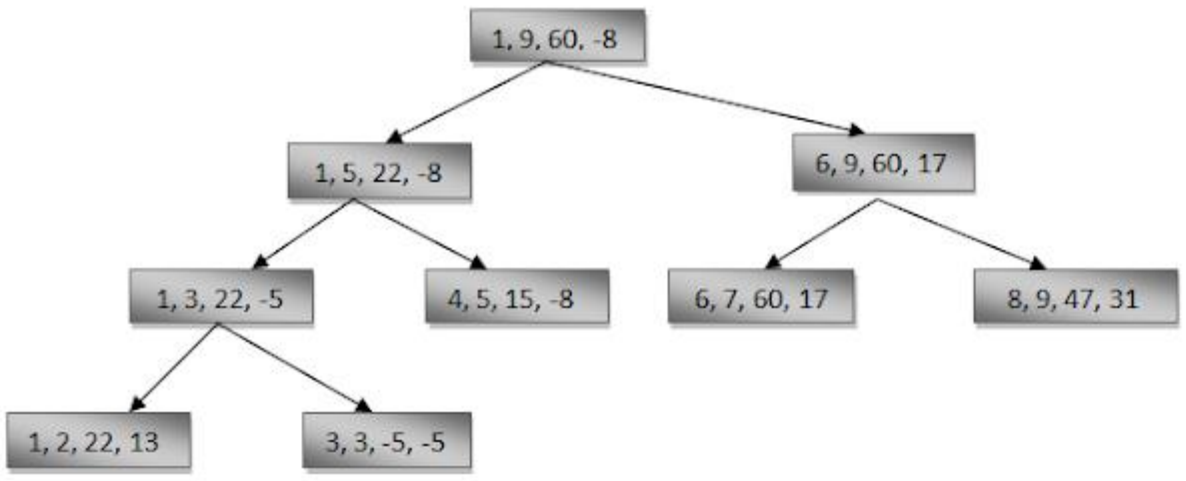
```
MaxMin(i, j, largest, smallest)
{
    if (i=j) then
        largest := smallest := a[i];
    else if (i=j-1) then
    {
        if (a[i] < a[j]) then
            largest := a[j];
            smallest := a[i];
        else
            largest := a[i];
            smallest := a[j];
    }
    else
    {
        mid := ( i + j )/2;
        MaxMin( i, mid, largest, smallest);
        MaxMin( mid+1, j, largest1, smallest1);

        if (largest < largest1) then
            largest := largest1;
        if (smallest > smallest1) then
            smallest := smallest1;
    }
}
```

IV) RESULT / OUTPUT / OBSERVATION

Example:

A	1	2	3	4	5	6	7	8	9
Values	22	13	-5	-8	15	60	17	31	47

Tree Diagram:



i. As shown in above figure, in this Algorithm, each node has 4 items of information: i, j, max & min.

ii. In figure 4, root node contains 1 & 9 as the values of i& j corresponding to the initial call to MaxMin.

iii. This execution produces 2 new calls to MaxMin, where i& j have the values 1, 5 & 6, 9 respectively & thus split the set into 2 subsets of approximately the same size.

iv. Maximum depth of recursion is 4.

V) DISCUSSION

Time Complexity: O(n)

2.1 Write a program to find the minimum number of scalar multiplication needed for chain of matrix using Dynamic Programming.

I) OBJECTIVE

Determine the optimal sequence of multiplication operations for a series of matrices.

II) THEORY

In this problem we determine the optimal sequence of multiplication operations for a series of matrices. This technique is also used for code optimization in complier design.

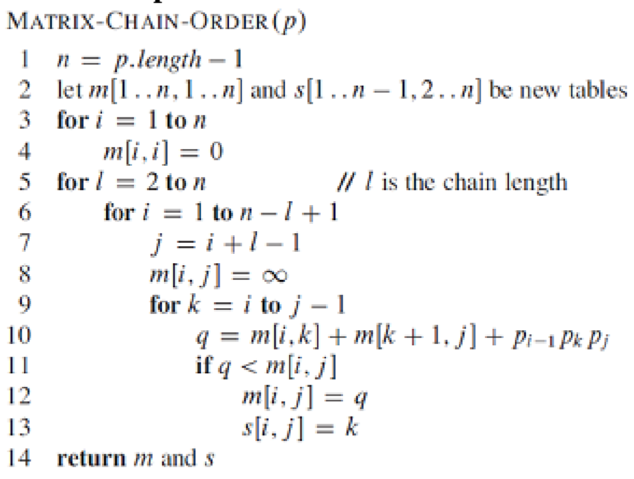
Let us consider a chain of n matrices $A_1\ A_2\ \ldots\ A_N$. As matrix multiplication is associative operation, we can parenthesize them without performing any rearrangement. Moreover, first we need to check the order of the matrices, to verify whether the operations can be performed on that sequence or not.

Suppose, matrix A has dimensions $P\ X\ Q$ which is being multiplied with another matrix B of dimensions $Q\ X\ R$, and the result will be a matrix C with dimensions $P\ X\ R$.

We know that the matrix multiplication is associative, so four matrices ABCD, we can multiply A(BCD), (AB)(CD), (ABC)D, A(BC)D, in these sequences. Like these sequences, our task is to find which ordering is efficient to multiply.

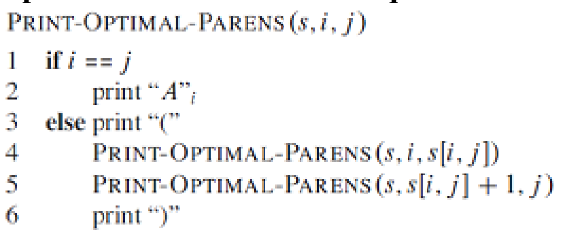
III) Algorithm for the Experiment / Assignment

Algorithm to calculate optimal cost



```
MATRIX-CHAIN-ORDER($p$)
$n = p.length - 1$
let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
for $i = 1$ to $n$
    $m[i,i] = 0$
for $l = 2$ to $n$                    // $l$ is the chain length
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $m[i,j] = \infty$
        for $k = i$ to $j - 1$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
            if $q < m[i,j]$
                $m[i,j] = q$
                $s[i,j] = k$
return $m$ and $s$
```

Algorithm to parenthesize the matrix sequnce



```
PRINT-OPTIMAL-PARENS($s, i, j$)
if $i == j$
    print "$A$"$_i$
else print "("
    PRINT-OPTIMAL-PARENS($s, i, s[i,j]$)
    PRINT-OPTIMAL-PARENS($s, s[i,j] + 1, j$)
    print ")"
```

IV) RESULT / OUTPUT / OBSERVATION

Input:

The orders of the input matrices. {1, 2, 3, 4}. It means the matrices are {(1 x 2), (2 x 3), (3 x 4)}.

Output:

Minimum number of operations need multiply these three matrices. Here the result is 18.

V) DISCUSSION

As before, if we have ***n*** matrices to multiply, it will take ***O(n)*** time to generate each of the ***$O(n^2)$*** costs and entries in the BEST matrix for an overall complexity of ***$O(n^3)$*** time at a cost of ***$O(n^2)$*** space.

2.2 Write a program to implement all pair of Shortest path for a graph (Floyed-Warshall Algorithm) using Dynamic Programming .

I) OBJECTIVE

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.

II) THEORY

Initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

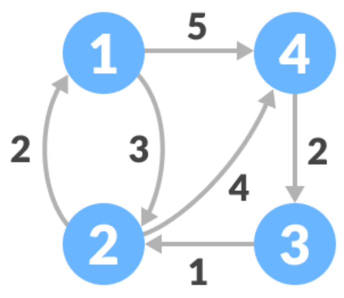
The following figure shows the above optimal substructure property in the all-pairs shortest path problem.

III) Algorithm for the Experiment / Assignment

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u, v) do
   dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
   dist[v][v] ← 0
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
         if dist[i][j] > dist[i][k] + dist[k][j]
            dist[i][j] ← dist[i][k] + dist[k][j]
         end if
```

IV) RESULT / OUTPUT / OBSERVATION

Consider the following directed weighted graph-



Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A^0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from ith vertex to jth vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & \infty & 1 & 0 & \infty \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

2. Now, create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the freezing vertex in the shortest path from source to destination. In this step, k is the first vertex.A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).

That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].

In this step, k is vertex 1. We calculate the distance from the source vertex to destination vertex through this vertex k.

$$A^1 = \begin{array}{ccccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & & \\ 3 & \infty & & 0 & \\ 4 & \infty & & & 0 \end{array} \rightarrow \begin{array}{ccccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & \infty & 1 & 0 & 8 \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (i.e. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since $4 < 7$, $A^0[2, 4]$ is filled with 4.

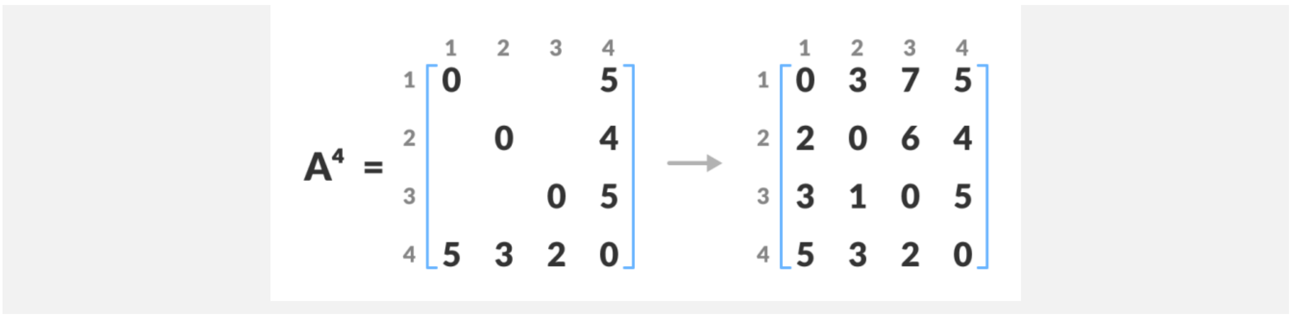
3. In a similar way, A^2 is created using A^1. The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2**.

$$A^2 = \begin{array}{ccccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & & \\ 2 & 2 & 0 & 9 & 4 \\ 3 & & 1 & 0 & \\ 4 & & \infty & & 0 \end{array} \rightarrow \begin{array}{ccccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

4. Similarly, A^3 and A^4 is also created.

$$A^3 = \begin{array}{ccccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & & \infty & \\ 2 & & 0 & 9 & \\ 3 & \infty & 1 & 0 & 8 \\ 4 & & & 2 & 0 \end{array} \rightarrow \begin{array}{ccccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$



$$A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & & 5 \\ 2 & & 0 & & 4 \\ 3 & & & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array} \rightarrow \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

5. A^4 gives the shortest path between each pair of vertices.

V) DISCUSSION

The time complexity of this algorithm is O(V^3), and the space complexity is: O(V^2).
where V is the number of vertices in the graph.

2.3 Write a program to implement Traveling Salesman Problem (TSP) using Dynamic Programming .

I) OBJECTIVE

Given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points.

II) THEORY

In the Euclidean traveling-salesman problem, we are given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time.

J. L. B entley has suggested that we simplify the problem by restricting our attention to bitonic tours, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. In this case, a polynomial-time algorithm is possible.

Describe an O(n^2)-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate and that all operations on real numbers take unit time.

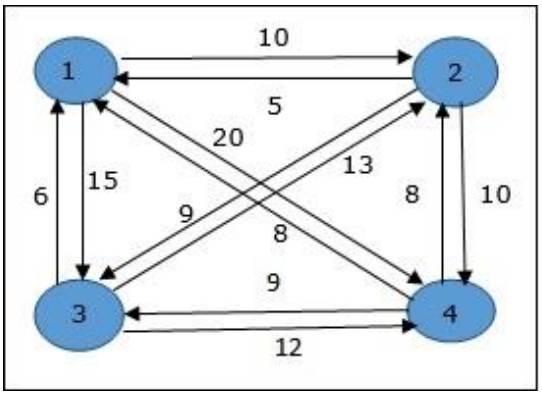
III) Algorithm for the Experiment / Assignment

```
C ({1}, 1) = 0
for s = 2 to n do
  for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
    C (S, 1) = ∞
  for all j Є S and j ≠ 1
    C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
```

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

IV) RESULT / OUTPUT / OBSERVATION

In the following example, we will illustrate the steps to solve the travelling salesman problem.



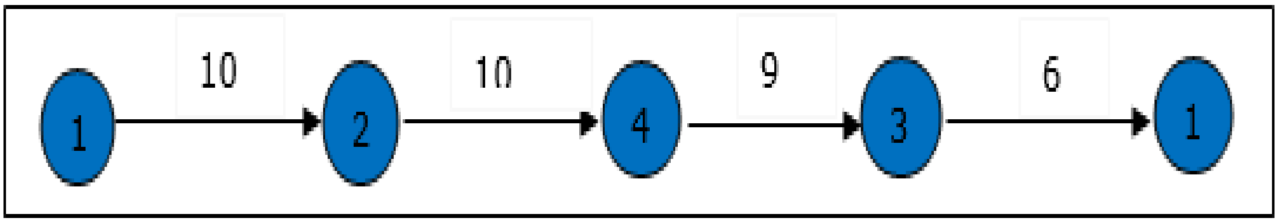
From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

The minimum cost path is 35.

Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.

When **s = 1**, we get the minimum value for **d [4, 3]**. Selecting path 4 to 3 (cost is 9), then we shall go to then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).



V) DISCUSSION

There are at the most $2^n.n$ sub-problems and each one takes linear time to solve. Therefore, the total running time is $O(2^n x n^2)$.

2.4 Write a program to implement Single Source shortest Path for a graph (Dijkstra Algorithm) using Dynamic Programming.

I) OBJECTIVE

Solve the single-source shortest-path problem for a graph having non-negative weights of edges.

II) THEORY

Dijkstra's algorithm is used to solve the single-source shortest-path problem for a graph having non-negative weights of edges. This algorithm starts at the source vertex. In every iteration, it expands with the neighboring vertices those are reachable from the source. The distances of the vertices are updated by minimum distance between the source and corresponding vertex.

III) Algorithm for the Experiment / Assignment

```
DIJKSTRA'S ALGORITHM (G, W, S)

INITIALIZE SINGLE-SOURCE (G, S)
S <- { }
INITIALIZE PRIORITY QUEUE Q BY ALL THE VERTICES OF THE GRAPH
WHILE PRIORITY QUEUE Q  IS NOT EMPTY DO
    U  <-  EXTRACT-MIN(Q)
    S  <-  S U {U}
    FOR EACH VERTEX V IN ADJ[U] DO
        RELAX (U, V, W)

```

IV) RESULT / OUTPUT / OBSERVATION

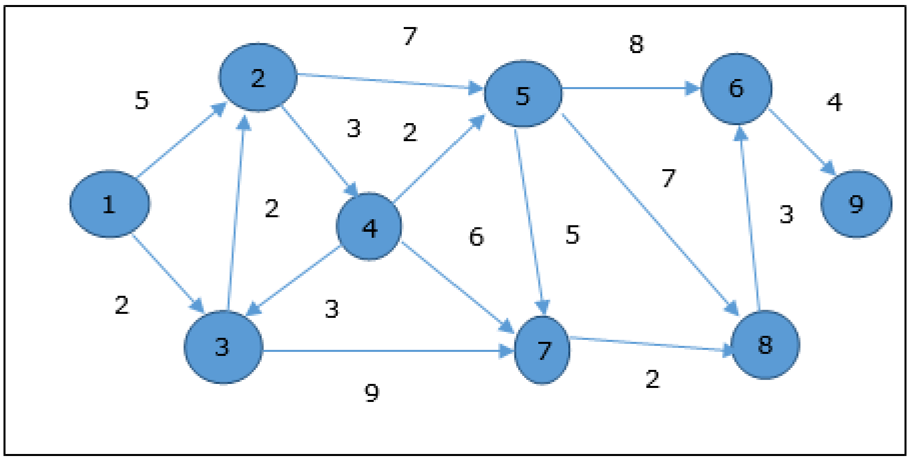
Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

Vertex	**Initial**	**Step1 V_1**	**Step2 V_3**	**Step3 V_2**	**Step4 V_4**	**Step5 V_5**	**Step6 V_7**	**Step7 V_8**	**Step8 V_6**
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is

1→ 3→ 7→ 8→ 6→ 9

This path is determined based on predecessor information.



V) DISCUSSION

The complexity of this algorithm is fully dependent on the implementation of Extract-Min function. If extract min function is implemented using linear search, the complexity of this algorithm is ***O(V^2 + E)***.

In this algorithm, if we use min-heap on which ***Extract-Min()*** function works to return the node from ***Q*** with the smallest key, the complexity of this algorithm can be reduced further.

2.4 Write a program to implement Single Source shortest Path for a graph (Bellman Ford Algorithm) using Dynamic Programming.

I) OBJECTIVE

The Bellman-Ford algorithm is a way to find single source shortest paths in a graph with negative edge weights (but no negative cycles).

II) THEORY

Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

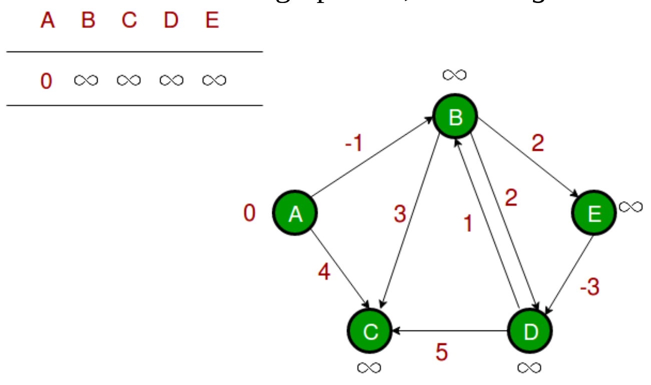
III) Algorithm for the Experiment / Assignment

Input: Graph and a source vertex *src*
Output: Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.
1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
…..**a)** Do following for each edge u-v
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]
………………….dist[v] = dist[u] + weight of edge uv

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

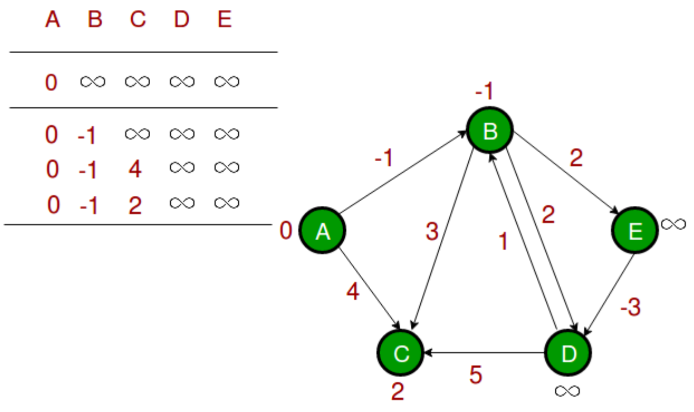
IV) RESULT / OUTPUT / OBSERVATION

Let us understand the algorithm with following example graph. The images are taken from this source.
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*



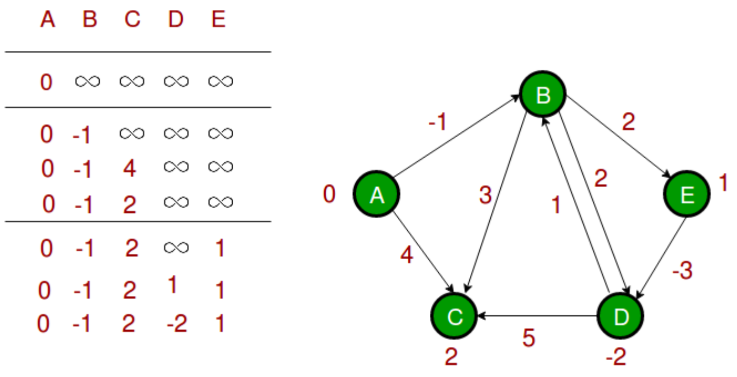
A	B	C	D	E
0	∞	∞	∞	∞

Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values).



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Output:

Vertex	Distance from Source
0	0
1	-1
2	2
3	-2
4	1

V) DISCUSSION

Time complexity of Bellman-Ford is O(VE).

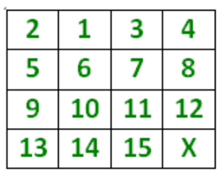
3.1 Write a program to implement 15 Puzzle Problem

I) OBJECTIVE

This game is played on a 4×4 board. On this board there are 15 playing tiles numbered from 1 to 15. One cell is left empty (denoted by 0). You need to get the board to the position presented below by repeatedly moving one of the tiles to the free space:

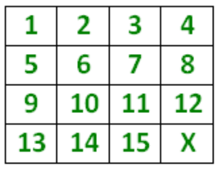
II) THEORY

Initial Configration



2	1	3	4
5	6	7	8
9	10	11	12
13	14	15	X

Final Configuration



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	X

Here X marks the spot to where the elements can be shifted and the final configuration always remains the same the puzzle is solvable.
In general, for a given grid of width N, we can find out check if a N*N – 1 puzzle is solvable or not by following below simple rules :

III) Algorithm for the Experiment / Assignment

1. If N is odd, then puzzle instance is solvable if number of inversions is even in the input state.
2. If N is even, puzzle instance is solvable if
   - the blank is on an even row counting from the bottom (second-last, fourth-last, etc.) and number of inversions is odd.
   - the blank is on an odd row counting from the bottom (last, third-last, fifth-last, etc.) and number of inversions is even.
3. For all other cases, the puzzle instance is not solvable.

IV) RESULT / OUTPUT / OBSERVATION

Let's consider this problem: given a position on the board, determine whether a sequence of moves which leads to a solution exists.

Suppose we have some position on the board:

a_1	a_2	a_3	a_4
a_5	a_6	a_7	a_8
a_9	a_{10}	a_{11}	a_{12}
a_{13}	a_{14}	a_{15}	a_{16}

where one of the elements equals zero and indicates an empty cell az=0

Let’s consider the permutation:

$$a_1a_2...a_{z-1}a_{z+1}...a_{15}a_{16}$$

i.e. the permutation of numbers corresponding to the position on the board without a zero element

Let NN be the number of inversions in this permutation (i.e. the number of such elements ai and aj that i<j, but ai>aj).

Suppose KK is an index of a row where the empty element is located (i.e. using our convention, K=(z−1)÷ 4+1K=(z−1)÷ 4+1).

Then, **the solution exists iff** N+K **is even**.

V) DISCUSSION

Time Complexity : O(n^2)
Space Complexity: O(n)

4.1 Write a program to implement 8 Queen Problem.

I) OBJECTIVE

This problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on the board.

II) THEORY

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

A binary matrix is used to display the positions of N Queens, where no queens can attack other queens.

III) Algorithm for the Experiment / Assignment

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.
Using place, we give a precise solution to then n- queens problem.

```
Place (k, i)
{
For j ← 1 to k - 1
do if (x [j] = i)
or (Abs x [j]) - i) = (Abs (j - k))
then return false;
return true;
}
```

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.
x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

```
N - Queens (k, n)
{
  For i ← 1 to n
        do if Place (k, i) then
  {
      x [k] ← i;
      if (k ==n) then
        write (x [1....n));
      else
      N - Queens (k + 1, n);
  }

}
```

IV) RESULT / OUTPUT / OBSERVATION

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.

Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."
Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

1 2 3 4

1			**Q1**	
	Q2			
2 3				**Q3**
4		**Q4**		

1. Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.
4. The first equation implies that j - l = i - k.
5. The second equation implies that j - l = k - i.
6. Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

V) DISCUSSION

The worst case “brute force” solution for the N-queens puzzle has an O(n^n) time complexity.

However, if it is found that N number of queens cannot be placed on that board, it will backtrack and try another safe position. This is over 100 times as fast as brute force and has a time complexity of O(2^n).

4.2 Write a program to implement Graph coloring Problem.

I) OBJECTIVE

Graph coloring is a technique to assign colors to the nodes in such a way that either none of the adjacent nodes share same color or none of the adjacent edges share same color.

The objective is to minimize the number of colors while coloring a graph.

II) THEORY

In a graph if the adjacent vertices do not share same color, the problem is known as vertex coloring. Similarly, if adjacent edges of a graph do not share same color, the problem is known as edge coloring.

III) Algorithm for the Experiment / Assignment

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

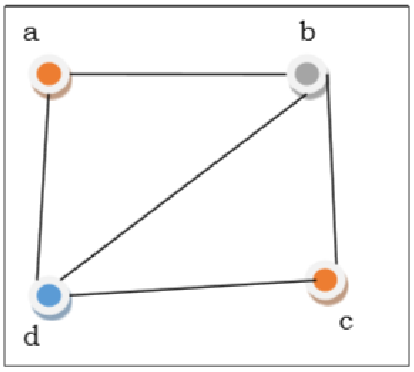
Step 1 − Arrange the vertices of the graph in some order.

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.

IV) RESULT / OUTPUT / OBSERVATION

Example



In the above figure, at first vertex a is colored red. As the adjacent vertices of vertex a are again adjacent, vertex b and vertex d are colored with different color, green and blue respectively. Then vertex c is colored as red as no adjacent vertex of c is colored red. Hence, we could color the graph by 3 colors. Hence, the chromatic number of the graph is 3.

V) DISCUSSION

The smallest number of colors required to color a graph G is called its chromatic number of that graph. Graph coloring problem is a NP Complete problem.

4.3 Write a program to implement Hamiltonian Problem.

I) OBJECTIVE

Determine whether a graph contains a Hamiltonian cycle or not.

II) THEORY

In an undirected graph, the Hamiltonian path is a path, that visits each vertex exactly once, and the Hamiltonian cycle or circuit is a Hamiltonian path, that there is an edge from the last vertex to the first vertex.

III) Algorithm for the Experiment / Assignment

isValid(v, k)

Input − Vertex v and position k.

Output − Checks whether placing v in the position k is valid or not.

```
Begin
   if there is no edge between node(k-1) to v, then
      return false
   if v is already taken, then
      return false
   return true; //otherwise it is valid
End
```

cycleFound(node k)

Input − node of the graph.

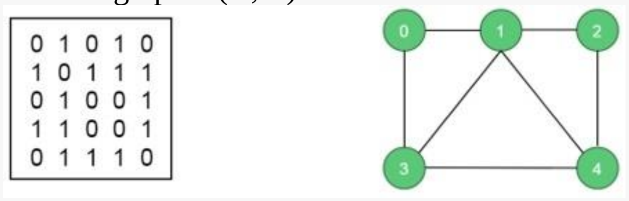
Output − True when there is a Hamiltonian Cycle, otherwise false.

```
Begin
   if all nodes are included, then
      if there is an edge between nodes k and 0, then
         return true
      else
         return false;

   for all vertex v except starting point, do
      if isValid(v, k), then //when v is a valid edge
         add v into the path
         if cycleFound(k+1) is true, then
            return true
         otherwise remove v from the path
   done
   return false
End
```

IV)RESULT / OUTPUT / OBSERVATION

Input:

The adjacency matrix of a graph G(V, E).



$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Output:

The algorithm finds the Hamiltonian path of the given graph. For this case it is (0, 1, 2, 4, 3, 0). This graph has some other Hamiltonian paths.
If one graph has no Hamiltonian path, the algorithm should return false.
Cycle: 0 1 2 4 3 0

V) DISCUSSION

Note that the above code always prints cycle starting from 0. The starting point should not matter as the cycle can be started from any point. If you want to change the starting point, you should make two changes to the above code.

5.1 Write a program to implement Knapsack Problem.

I) OBJECTIVE

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.

II) THEORY

Let us consider that there are N items in a store. Weight of item I is W_I *(WHERE* $W_I > 0$*)* and worth of the item is V_I *(WHERE* $V_I > 0$*)* . A thief has a knapsack by which he can carry some items of maximum weight W pounds. In this problem the thief can carry a fraction X_I of item I , where $0 \leq X_I \leq 1$. So, item I contributes $X_I.W_I$ to the total weight in the knapsack, and $X_I.\ V_I$ to the profit.

Hence, the objective function of fraction knapsack problem can be written as follow:

maximize $\Sigma^n_{i=1}\ x_i.v_i$ subject to constraint $\Sigma^n_{i=1}\ x_i.w_i \leq W$

In this problem, the optimal solution can be achieved if the knapsack is full when cumulative weight of the items is greater than size of the knapsack. Thus in an optimal solution $\Sigma^n_{i=1}\ x_i.w_i = W$.

III) Algorithm for the Experiment / Assignment

```
GREEDY-FRACTIONAL-KNAPSACK (W, V, W)

FOR I = 1 TO N
    DO X[I] = 0
WEIGHT = 0
WHILE WEIGHT < W
    DO
        I = BEST AMONG REMAINING ITEMS
        IF WEIGHT + W[I] <= W
        THEN
            X[I] = 1
            WEIGHT = WEIGHT + W[I]
        ELSE
            X[I] = (W - WEIGHT) / W[I]
            WEIGHT = W
RETURN X
```

IV) RESULT / OUTPUT / OBSERVATION

Example

Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table −

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (p_i / w_i)	7	10	6	5

As the provided items are not sorted based on p_i / w_i. After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio (p_i / w_i)	10	7	6	5

Solution

After sorting all the items according to $p_i w_i$. First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

V) DISCUSSION

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

5.2 Write a program to implement Job sequencing with deadlines.

I) OBJECTIVE

In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

II) THEORY

Let us consider, a set of ***n*** given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines. Assume, deadline of **ith** job J_i is d_i and the profit received from this job is p_i. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Thus, D(i)>0*D(i)>0* for 1⩽i⩽n*1⩽i⩽n*.

Initially, these jobs are ordered according to profit,

i.e. p1⩾p2⩾p3⩾...⩾pn*p1⩾p2⩾p3⩾...⩾pn*.

III) Algorithm for the Experiment / Assignment

Job-Sequencing-With-Deadline (D, J, n, k)

```
D(0) := J(0) := 0
k := 1
J(1) := 1   // means first job is selected
for i = 2 … n do
   r := k
   while D(J(r)) > D(i) and D(J(r)) ≠ r do
      r := r – 1
   if D(J(r)) ≤ D(i) and D(i) > r then
      for l = k … r + 1 by -1 do
         J(l + 1) := J(l)
         J(r + 1) := i
         k := k + 1
```

IV) RESULT / OUTPUT / OBSERVATION

Example

Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	**J_1**	**J_2**	**J_3**	**J_4**	**J_5**
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	**J_2**	**J_1**	**J_4**	**J_3**	**J_5**
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select ***J_2***, as it can be completed within its deadline and contributes maximum profit.

- Next, ***J_1*** is selected as it gives more profit compared to ***J_4***.
- In the next clock, ***J_4*** cannot be selected as its deadline is over, hence ***J_3*** is selected as it executes within its deadline.
- The job ***J_5*** is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (***J_2, J_1, J_3***), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is **100 + 60 + 20 = 180**.

V) DISCUSSION

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $O(n^2)$.

5.3 Write a program to implement Minimal Spanning Tree by using Prim's Algorithm.

I) OBJECTIVE

This Algorithm is used to find the minimum cost spanning tree for a connected weighted graph using greedy approach.

II) THEORY

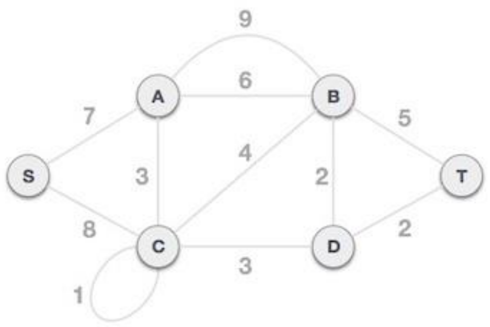
Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph. This algorithm has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. Each step adds to the tree A a light edge that connects A to an isolated vertex-one on which no edge of A is incident. This rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.

III) Algorithm for the Experiment / Assignment

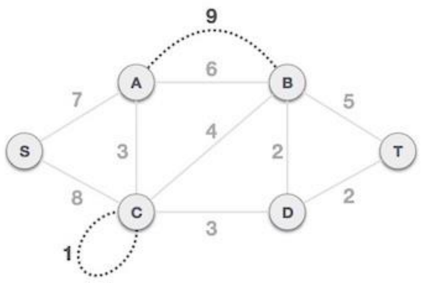
```
MST-PRIM($G, w, r$)
  for each $u \in G.V$
      $u.key = \infty$
      $u.\pi$ = NIL
  $r.key = 0$
  $Q = G.V$
  while $Q \neq \emptyset$
      $u$ = EXTRACT-MIN($Q$)
      for each $v \in G.Adj[u]$
          if $v \in Q$ and $w(u, v) < v.key$
              $v.\pi = u$
              $v.key = w(u, v)$
```

IV) RESULT / OUTPUT / OBSERVATION

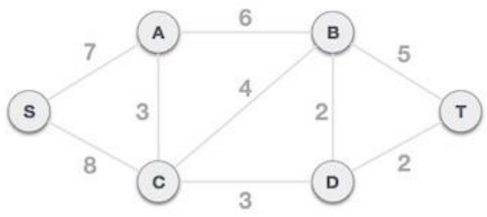
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

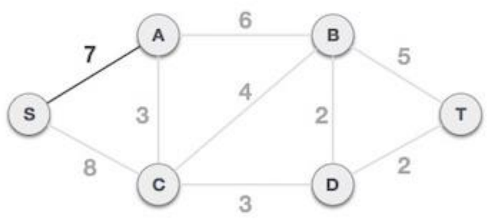


Step 2 - Choose any arbitrary node as root node

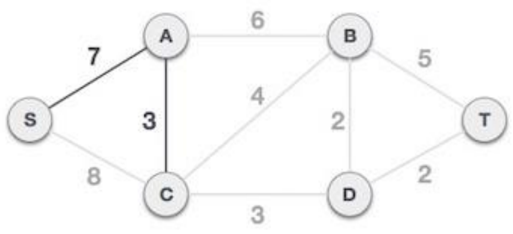
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

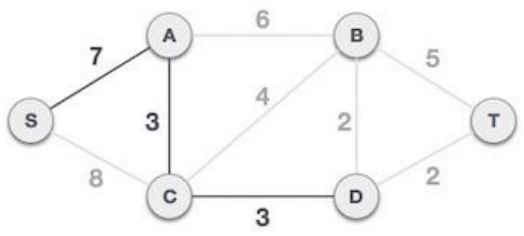
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



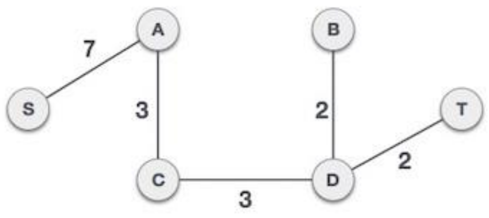
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

Thus the cost of MST is 2+2+3+3+7=17

V) DISCUSSION

The running time of Prim's algorithm depends on how we implement the min priority queue Q. If we implement Q as a binary min-heap , we can use the BUILD-MIN-HEAP procedure to perform lines 1-5 in O(V) time. The body of the while loop executes |V| times, and since each EXTRACT-MIN operation takes O(lg V) time, the total time for all calls to EXTRACT-MIN is O(V lg V). The for loop in lines 8-11 executes O(E) times altogether, since the sum of the lengths of all adjacency lists is 2|E|. Within the for loop, we can implement the test for membership in Q in line 9 in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q. The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which a binary min-heap supports in O(lg V) time. Thus, the total time for Prim's algorithm is O(V lg V) + E lg V) = O(E lg V), which is asymptotically the same as for our implementation of Kruskal's algorithm.

5.4 Write a program to implement Minimal Spanning Tree by using Kruskal's Algorithm.

I) OBJECTIVE

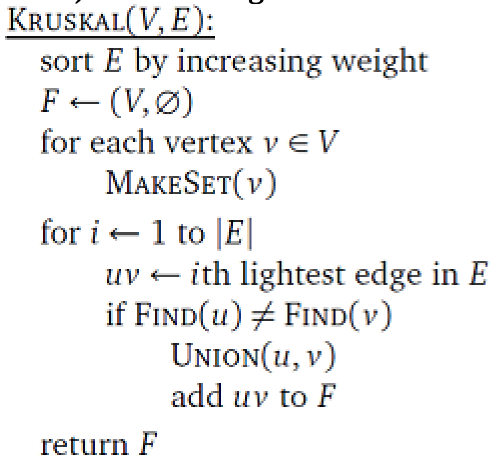
This Algorithm is used to find the minimum cost spanning tree for a connected weighted graph using greedy approach.

II) THEORY

This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

Scan all edges in increasing weight order; if an edge is safe, add it to forest F.

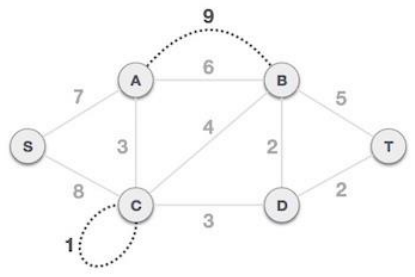
III) Algorithm for the Experiment / Assignment



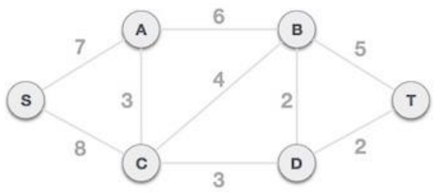
```
KRUSKAL($V, E$):
  sort $E$ by increasing weight
  $F \leftarrow (V, \emptyset)$
  for each vertex $v \in V$
      MAKESET($v$)
  for $i \leftarrow 1$ to $|E|$
      $uv \leftarrow i$th lightest edge in $E$
      if FIND($u$) $\neq$ FIND($v$)
          UNION($u, v$)
          add $uv$ to $F$
  return $F$
```

IV) RESULT / OUTPUT / OBSERVATION

Step 1 -Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



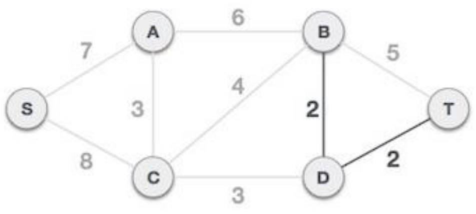
Step 2 - Arrange all edges in their increasing order of weight.

Create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

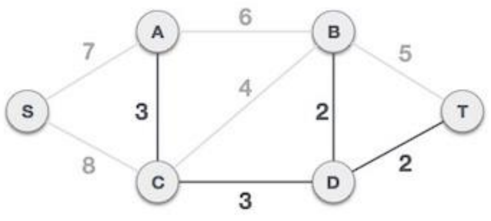
Step 3 - Add the edge which has the least weightage.

Start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

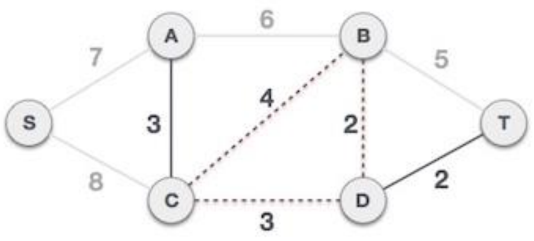


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

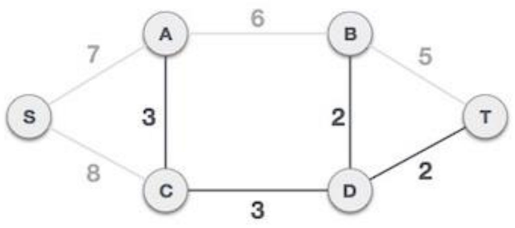
Next cost is 3, and associated edges are A,C and C,D. We add them again −



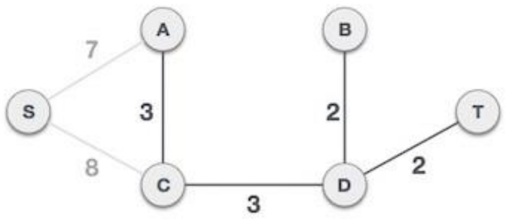
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



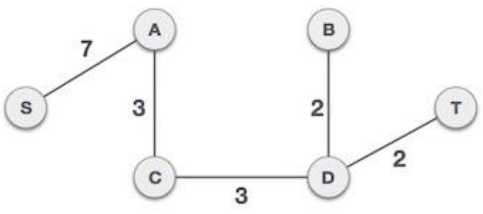
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Thus the cost of MST is 2+2+3+3+7=17

V) DISCUSSION

The sets are components of F, and n = V. Kruskal's algorithm performs O(E) FIND operations, two for each edge in the graph, and O(V) UNION operations, one for each edge in the minimum spanning tree. Using union-by-rank and path compression allows us to perform each UNION or FIND in O(α(E, V)) time, where α is the not-quite-constant inverse-Ackerman function. So ignoring the cost of sorting the edges, the running time of this algorithm is O(E α(E, V)). We need O(E log E) = O(E log V) additional time just to sort the edges. Since this is bigger than the time for the UNION-FIND data structure, the overall running time of Kruskal's algorithm is O(E log V).
Traversing is nothing but visiting each node/vertex of a graph in a systematic way. Graph is represented by its nodes and edges, so, visiting all the nodes of a graph is known as traversal of

the graph.
There are two approaches for graph traversal, namely Breadth First Search (BFS) and Depth First Search (DFS). In this tutorial we will discuss BFS approach.

6.1 Write a program to implement Breadth First Search Algorithm.

I) OBJECTIVE

This algorithm is used for traversing or searching a graph.

II) THEORY

In this technique, first we take any node as a starting node and all the adjacent nodes are chosen from that starting node. In next step, same approach is applied for all the newly visited nodes and continued until all the nodes are visited. We maintain the status of visited node in one array so that no node can be traversed again.

III) Algorithm for the Experiment / Assignment

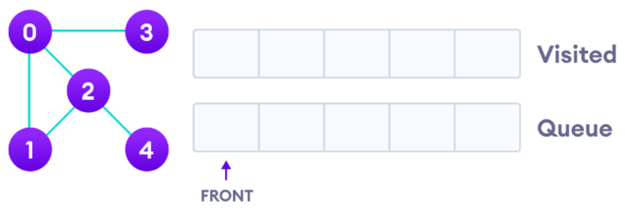
This graph traversal technique uses queue for traversing all the nodes of the graph.

Algorithm :

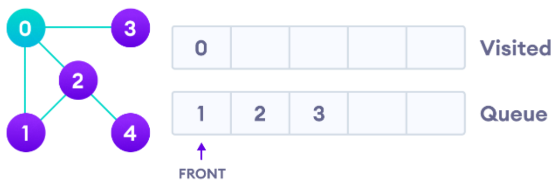
1. Insert starting node into the queue.
2. Delete front element from the queue and insert all its unvisited neigh bours into the queue at the end and traverse them. Also make the value of visited array true for these nodes.
3. Repeat step 2 until the queue is empty.

IV) RESULT / OUTPUT / OBSERVATION

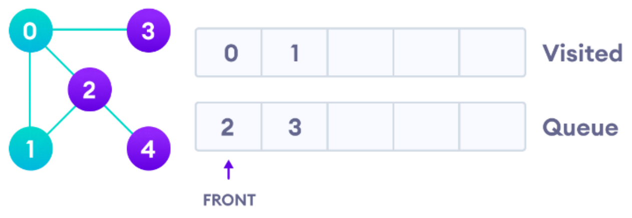
Example



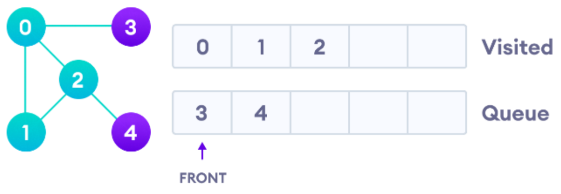
Undirected graph with 5 vertices



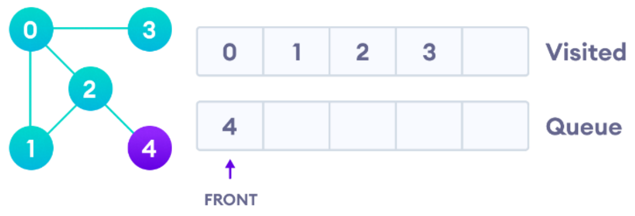
Visit start vertex and add its adjacent vertices to queue



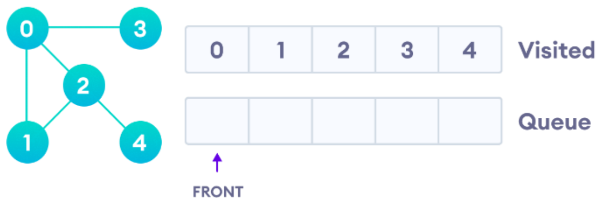
Visit the first neighbour of start node 0, which is 1



Visit 2 which was added to queue earlier to add its neighbours



4 remains in the queue



Visit last remaining item in the stack to check if it has unvisited neighbors

Since the queue is empty, we have completed the Breadth First Traversal of the graph..

V) DISCUSSION

The time complexity of the BFS algorithm is represented in the form of $O(V + E)$, where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is $O(V)$.

6.2 Write a program to implement Depth First Search Algorithm.

I) OBJECTIVE

This algorithm is used for traversing or searching a graph.

II) THEORY

As it name suggest, is to search deeper in the graph, whenever possible. Given as input graph G = (V, E) and a source vertex S, from where the searching starts. First we visit the starting node, then we travel through each node along a path, which begins at S. We visit a neighbour vertex of S and again a neighbour of a neighbour of S and so on.

III) Algorithm for the Experiment / Assignment

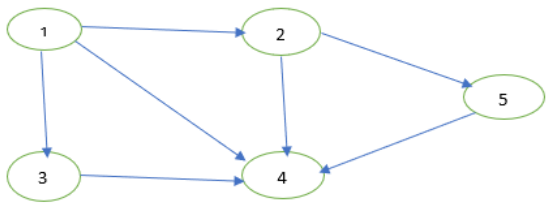
In this method, stack is used to keep the track of unvisited neighbours of the node. But question is that how can we distinguish the visited and unvisited node? Take a another Boolean array VISITED, which will show the visited and unvisited node. Let's see the procedure how it works?

Algorithm :

1. Push starting node into the stack.
2. Pop an element from the stack, if it has not been traversed then traverse it. If it has already been traversed then just ignore it. after traversing make the value of visited array true for this node.
3. Push all the unvisited adjacent nodes of the popped element on stack. No need to Push the element even if it is already on the stack.
4. Repeat step 2 and 3 until stack is empty.

IV) RESULT / OUTPUT / OBSERVATION

Example



.

Suppose, the starting node is 1.
Push(1)
Stack content:1
Pop()
Stack content:empty
visited[1] = TRUE
Push all the unvisited adjacent nodes 2, 5, 4 into the stack.
Stack content:2, 5, 4 and Top = 2
Pop()
Stack content:2, 5 and Top = 1

visited[4] = TRUE
Push adjacent nodes of 4. But there is no adjacent node.
Stack content:2,5 and Top = 1
Pop()
Stack content:2 and Top = 0
visited[5] = TRUE
Push unvisited adjacent nodes of 5.Adjacent node of 5 is 4 which is already visited.
Stack content:2 and Top = 0
Pop()
Stack content:empty and Top = -1
visited[2] = TRUE
Push unvisited node of 2.
Stack content:3 and Top = 0
Pop()
Stack content:empty and Top = -1
visited[3] = TRUE

Stack is empty and there is no adjacent node of 3.So we stop the process.

Finally, the DFS traversal is:

1, 4, 5, 2, 3

V) DISCUSSION

Time Complexity of DFS is O(V+E) where V is vertices and E is edges.

LAB EXERCISE / HOME ASSIGMENTS

A) List of home assignments

1. Write a program to implement Binary Search using Divide and Conquer approach.
2. Write a program to implement Merge Sort using Divide and Conquer approach.
3. Write a program to Implement Quick Sort using Divide and Conquer approach.
4. Write a program to find Maximum and Minimum element from an array of integer using Divide and Conquer approach.
5. Write a program to find the minimum number of scalar multiplication needed for chain of matrix using Dynamic Programming approach.
6. Write a program to implement all pair of Shortest path for a graph (Floyed- Warshall Algorithm) using Dynamic Programming approach.
7. Write a program to implement Traveling Salesman Problem using Dynamic Programming approach.
8. Write a program to implement Single Source shortest Path for a graph (Dijkstra , Bellman Ford Algorithm) using Dynamic Programming approach.
9. Write a program to implement 15 Puzzle Problem
10. Write a program to implement 8 Queen problem.
11. Write a program to implement Graph Coloring Problem.
12. Write a program to implement Hamiltonian Problem.
13. Write a program to implement Knapsack Problem.
14. Write a program to implement Job sequencing with deadlines.
15. Write a program to implement Minimum Cost Spanning Tree by Prim's Algorithm.
16. Write a program to implement Minimum Cost Spanning Tree by Kruskal's Algorithm.
17. Write a program to implement Breadth First Search (BFS).
18. Write a program to implement Depth First Search (DFS).

SAMPLE VIVA-VOCE QUESTIONS

1. What is Algorithm?
2. What is the time complexity of Algorithm?
3. What is NP-Complete?
4. Differentiate Time Efficiency and Space Efficiency?
5. What is Dynamic Programming?
6. What is the Knapsack Problem?
7. What is a Greedy Algorithm?
8. What is Minimum Spanning Trees?
9. What is Kruskal?s Algorithm?
10. What is backtracking?

EXAMINATIONS

I) Conduction of Viva-voce
II) Evaluation and marking system

REFERENCES:

https://www.tutorialspoint.com