Maharaja Education Trust (R), Mysuru

Maharaja Institute of Technology Mysore

Belawadi, Sriranga Pattana Taluk, Mandya – 571 477

Approved by AICTE, New Delhi,
Affiliated to VTU, Belagavi & Recognized by Government of Karnataka
Accredited By NBA and NAAC with B++

Lab Manual of

Analysis and Desing of Algorithms Lab

(BCSL404)

Department of Computer Science and Engineering

Program Outcomes

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
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.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
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.
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.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
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.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
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.
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.
12. **Life-long learning**: 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.

Maharaja Institute of Technology Mysore

Department of Computer Science and Engineering

Course Overview

Subject: Analysis and Design of Algorithms Lab (ADAL) **Subject Code: BCSL404**

Analysis and Design of Algorithms is a fundamental aspect of computer science that involves creating efficient solutions to computational problems and evaluating their performance. ADA focuses on designing algorithms that effectively address specific challenges and analyzing their efficiency in terms of time and space complexity. All the algorithms have to be implemented either writing C programs or writing C++ programs.

Course Objectives

- To design and implement various algorithms in C/C++ programming using suitable development tools to address different computational challenges.
- To apply diverse design strategies for effective problem-solving.
- To Measure and compare the performance of different algorithms to determine their efficiency and suitability for specific tasks.

Course Outcomes

COs	Description
C244.1	Analyze and Apply various algorithms design strategies to solve computational problems.
C241.2	Develop a C/C++ program to implement various algorithms to solve problems using Modern tool and document the same with appropriate oral justification.
C241.3	Demonstrate and Evaluate the runtime performance of different algorithm design approaches for the given data.

General Lab Guidelines

General Lab Guidelines:

- Maintain laboratory etiquettes during the laboratory sessions.
- Do not wander around or distract other students or interfere with the conduction of the experiments of other students.
- Keep the laboratory clean, do not eat, drink or chew gum in the laboratory.

DO'S

- Sign the log book when you enter/leave the laboratory.
- Read the handout/procedure before starting the experiment. If you do not understand the procedure, clarify with the concerned staff.
- Report any problem in system (if any) to the person in-charge.
- After the lab session, shut down the computers.
- All students in the laboratory should follow the directions given by staff/lab technical staff.

DON'TS

- Do not insert metal objects such as pins, needle or clips into the computer casing. They may cause fire.
- Do not open any irrelevant websites in labs.
- Do not use flash drive on laboratory computers without the consent of lab instructor.
- Do not upload, delete or alter any software/ system files on laboratory computers.
- Students are not allowed to work in laboratory alone or without presence of the teaching staff/ instructor.
- Do not change the system settings and keyboard keys.
- Do not damage any hardware.

MAHARAJA INSTITUTE OF TECHNOLOGY MYSORE

BELAWADI, SRIRANGAPATNA Tq, MANDYA-571477

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

2023-24 (Even) – 4th Semester

Subject: **Analysis & Design of Algorithms Lab** Subject Code: **BCSL404**

Lab Program list

Exp. No.	Name of the Experiment	Page No.
1	Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm	1-3
2	Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.	4-7
3	a. Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm. b. Design and implement C/C++ Program to find the transitive closure using Warshal's algorithm.	8-12
4	Design and implement C/C++ Program to find shortest paths from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm.	13-15
5	Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.	16-17
6	Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method.	18-19
7	Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.	20-21
8	Design and implement C/C++ Program to find a subset of a given set S = {sl , s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d.	22-23
9	Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.	24-25
10	Design and implement C/C++ Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.	26-28
11	Design and implement C/C++ Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.	29-30
12	Design and implement C/C++ Program for N Queen's problem using Backtracking.	31-33
13	Viva Questions with answer	34-40

Program 1: Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.

Description:

A ***spanning tree*** of a connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.

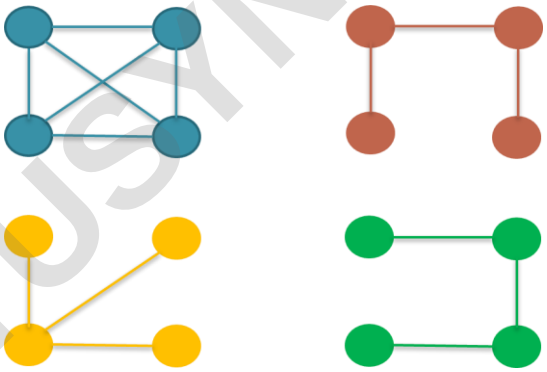
A ***minimum spanning*** *tree* of a weighted connected graph is its spanning tree of the smallest weight, where the *weight* of a tree is defined as the sum of the weights on all its edges.

The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

Introduction to Kruskal's Algorithm:

In Kruskal's algorithm, **sort all edges of the given graph in increasing order**. Then it keeps on adding new edges and nodes in the MST (***minimum spanning tree***) if the newly added edge **does not form a cycle**. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it **makes a locally optimal choice in each step in order to find the optimal solution**. Hence this is a Greedy Algorithm.

Example: Figure below shows the complete **graph on four nodes** together with **three of its spanning trees**



Algorithm:

Steps

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 tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

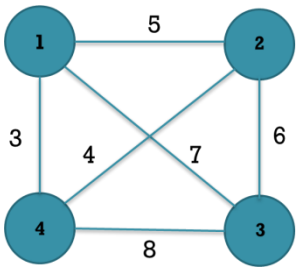
Program:

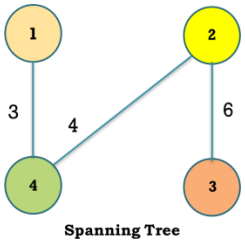
/* Program to implement Kruskal's Algorithm */

```
#include<stdio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
    printf("\n\t Implementation of Kruskal's algorithm\n");
    printf("\nEnter the no. of vertices:");
    scanf("%d",&n);
```

```
    printf("\nEnter the cost adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cost[i][j]);
            if(cost[i][j]==0)
                cost[i][j]=999;
        }
    }
    printf("The edges of Minimum Cost Spanning Tree are\n");
    while(ne < n)
    {
        for(i=1,min=999;i<=n;i++)
        {
            for(j=1;j <= n;j++)
            {
                if(cost[i][j] < min)
                {
                    min=cost[i][j];
                    a=u=i;
                    b=v=j;
                }
            }
        }
        u=find(u);
        v=find(v);
        if(uni(u,v))
        {
            printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
            mincost +=min;
        }
        cost[a][b]=cost[b][a]=999;
    }
    printf("\n\tMinimum cost = %d\n",mincost);
}
int find(int i)
{
    while(parent[i])
        i=parent[i];
    return i;
}
int uni(int i,int j)
{
    if(i!=j)
    {
        parent[j]=i;
        return 1;
    }
    return 0;
}
```

Input Graph:





Spanning Tree

Output 1:

Implementation of Kruskal's algorithm
Enter the no. of vertices: 4
Enter the cost adjacency matrix:
999 5 7 3
5 999 6 4
7 6 999 8
3 4 6 999

The edges of Minimum Cost Spanning Tree are
1 edge (1,4) =3
2 edge (2,4) =4
3 edge (2,3) =6
Minimum cost = 13

Output 2:
Implementation of Kruskal's algorithm
Enter the no. of vertices:7
Enter the cost adjacency matrix:
0 28 0 0 0 10 0
28 0 16 0 0 0 14
0 16 0 12 0 0 0
0 0 12 0 22 0 18
0 0 0 22 0 25 24
10 0 0 0 25 0 0
0 14 0 0 24 0 0
The edges of Minimum Cost Spanning Tree are
1 edge (1,6) =10
2 edge (3,4) =12
3 edge (2,7) =14
4 edge (2,3) =16
5 edge (4,5) =22
6 edge (5,6) =25

Minimum cost = 99

Performance Analysis

The Kruskal's method has an O(E log E) or O(V log V) time complexity, where E is the number of edges and V is the number of vertices.

Program 2: Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.

Description:

Like Kruskal's algorithm, **Prim's algorithm** is also a Greedy algorithm. This algorithm always starts with a **single node and moves through several adjacent** nodes, in order to explore all of the connected edges along the way.

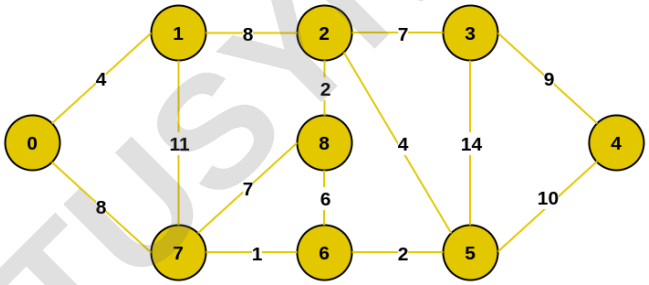
The algorithm starts with an **empty spanning tree**. The idea is to maintain **two sets of vertices**. The first set contains the **vertices already included** in the MST, and the other set contains the vertices **not yet included**. At every step, it considers **all the edges that connect the two sets and picks the minimum weight edge** from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing Minimum Cost Spanning Tree.

Algorithm:

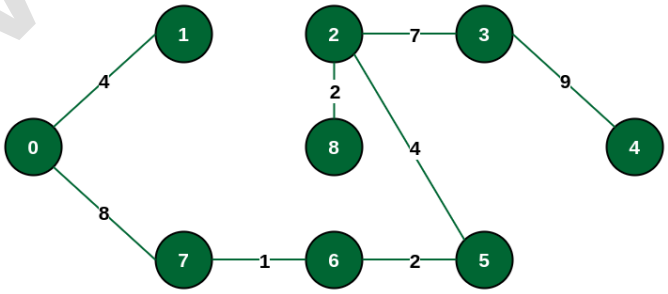
Step 1: Determine an arbitrary vertex as the starting vertex of the Minimum Cost Spanning Tree.
Step 2: Follow steps 3 to 5 till there are vertices that are not included in the Minimum Cost Spanning Tree (known as fringe vertex).
Step 3: Find edges connecting any tree vertex with the fringe vertices.
Step 4: Find the minimum among these edges.
Step 5: Add the chosen edge to the Minimum Cost Spanning Tree if it does not form any cycle.
Step 6: Return the Minimum Cost Spanning Tree and exit.

Example:

Input Graph:



Output Graph:



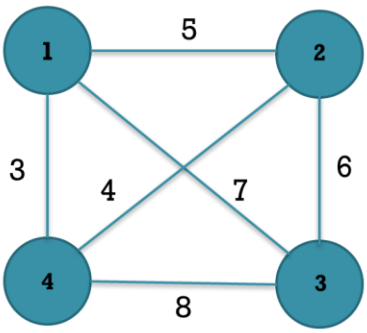
Program:

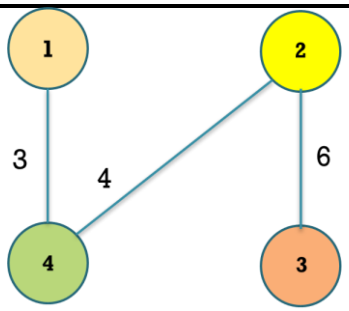
```
#include<stdio.h>
int n,cost[10][10],temp,nears[10];
void readv();
void primsalg();
void readv()
{
    int i,j;
    printf("\n Enter the No of nodes or vertices:");
    scanf("%d",&n);
    printf("\n Enter the Cost Adjacency matrix of the given graph: \n");
```

```
    for(i=1;i<=n;i++)
        {
            for(j=1;j<=n;j++)
            {
                scanf("%d",&cost[i][j]);
                if((cost[i][j]==0) && (i!=j))
                {
                   cost[i][j]=999;
                }
            }
        }
}
void primsalg()
{
    int k,l,min,a,t[10][10],u,i,j,mincost=0;
    min=999;
    for(i=1;i<=n;i++) //To Find the Minimum Edge E(k,l)
    {
        for(u=1;u<=n;u++)
        {
            if(i!=u)
            {
                if(cost[i][u]<min)
                {
                   min=cost[i][u];
                   k=i;
                   l=u;
                }
            }
        }
    }
    t[1][1]=k;
    t[1][2]=l;
    printf("\n The Minimum Cost Spanning tree is...");
    printf("\n(%d,%d)-->%d",k,l,min);
    for(i=1;i<=n;i++)
    {
        if(i!=k)
        {
            if(cost[i][l]<cost[i][k])
            {
                nears[i]=l;
            }
            else
            {
               nears[i]=k;
            }
        }
    }
    nears[k]=nears[l]=0;
   mincost=min;
   for(i=2;i<=n-1;i++)
   {
```

```
        j = findnextindex(cost,nears);
        t[i][1]=j;
        t[i][2]=nears[j];
        printf("\n(%d,%d)-->%d",t[i][1],t[i][2],cost[j][nears[j]]);
        mincost=mincost+cost[j][nears[j]];
        nears[j]=0;
        for(k=1;k<=n;k++)
        {
            if(nears[k]!=0 && cost[k][nears[k]]>cost[k][j])
            {
                  nears[k]=j;
            }
        }
    }
    printf("\n The Required Mincost of the Spanning Tree is:%d",mincost);
}
int findnextindex(int cost[10][10],int nears[10])
{
    int min=999,a,k,p;
    for(a=1;a<=n;a++)
    {
        p=nears[a];
        if(p!=0)
         {
            if(cost[a][p]<min)
            {
               min=cost[a][p];
               k=a;
            }
        }
    }
 return k;
}
void main()
{
    readv();
    primsalg();
}
```

Input Graph:





Spanning Tree

Output:

Enter the No of nodes or vertices:4

Enter the Cost Adjacency matrix of the given graph:

999 5 7 3
5 999 6 4
7 6 999 8
3 4 6 999

The Minimum Cost Spanning tree is...

(1,4)-->3
(2,4)-->4
(3,2)-->6

The Required Mincost of the Spanning Tree is:13

Performance Analysis:

Time Complexity: $O(V^2)$, If the input graph is represented using an adjacency list, then the time complexity of Prim's algorithm can be reduced to O(E * log V) with the help of a binary heap.

Program 3.a: Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm.

Description:

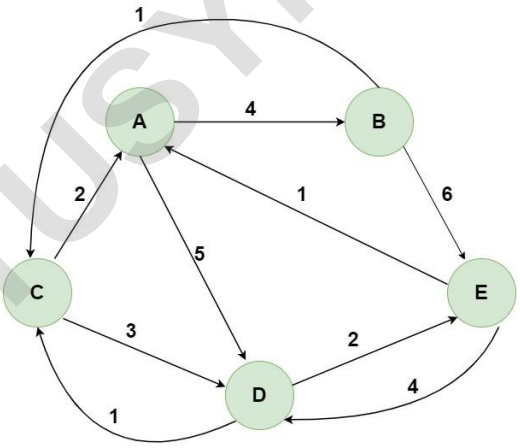
The Floyd Warshall Algorithm is an **all pair shortest path algorithm** unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. This algorithm works for both the **directed and undirected weighted graphs**. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative). It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

Algorithm:

ALGORITHM *Floyd(W[1..n, 1..n])*
//Implements **Floyd's algorithm** for the **all-pairs shortest -paths** problem
//**Input**: The weight **matrix *W*** of a graph with **no negative-length cycle.**
//**Output**: The distance matrix of the **shortest paths' lengths.**
D ← ***W*** // is not necessary if ***W*** can be **overwritten**
for *k* ← 1 to *n* **do**
  for *i* ← 1 **to** *n* do
    for *j* ← 1 to *n* **do**
      D[i, j] ← **min{ D[i, j],** ***D[i, k]*** **+** ***D[k, j]*****}**
return *D*

Example:

Input Graph



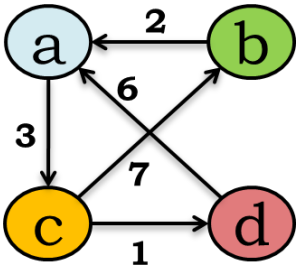
Output Matrix: Distance Matrix:

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Program:

```
/* Program to find all pair shortest path. */
#include<stdio.h>
void readf();
void amin();
int cost[20][20],a[20][20];
int i,j,k,n;
    void readf()
    {
     printf("\n Enter the number of vertices :");
     scanf("%d",&n);
     printf("\n Enter the weighted matrix - 999 for infinity:");
     for(i=0;i<n;i++)
     {
         for(j=0;j<n;j++)
         {
            scanf("%d",&cost[i][j]);
            if(cost[i][j]==0 && (i!=j))
                    cost[i][j]=999;
            a[i][j]=cost[i][j];
         }
     }
    }
    void amin()
    {
         for(k=0;k<n;k++)
         {
             for(i=0;i<n;i++)
             {
                for(j=0;j<n;j++)
                {
                     if(a[i][j]>a[i][k]+a[k][j])
                     {
                        a[i][j]=a[i][k]+a[k][j];
                     }
                }
             }
         }
         printf("\n The All pair shortest path is:");
         for(i=0;i<n;i++)
         {
             printf("\n");
             for(j=0;j<n;j++)
             {
                printf("%d\t",a[i][j]);
             }
         }
    }
    void main()
    {
         readf();
         amin();
     }
```

INPUT GRAPH:



OUTPUT:

Enter the number of vertices:
4
Enter the weighted matrix - 999 for infinity :

0	999	3	999
2	0	999	999
999	7	0	1
6	999	999	0

The All pair shortest path is:

0	**10**	**3**	**4**
2	**0**	**5**	**6**
7	**7**	**0**	**1**
6	**16**	**9**	**0**

Performance:

Time Complexity: O(V^3), where V is the number of vertices in the graph and we run three nested loops each of size V

Program 3.b: Design and implement C/C++ Program to find the transitive closure using Warshal's algorithm.

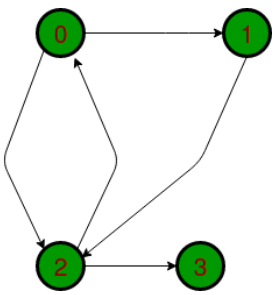
Description:

Given a directed graph, determine if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable means that there is a path from vertex i to j. The reach-ability matrix is called the transitive closure of a graph.

The graph is given in the form of adjacency matrix say 'graph[V][V]' where **graph[i][j] is 1** if there is an edge from vertex i to vertex j or i is equal to j, otherwise **graph[i][j] is 0**.

Floyd Warshall Algorithm can be used, we can calculate the distance matrix dist[V][V] using Floyd Warshall, if dist[i][j] is infinite, then j is not reachable from i. Otherwise, j is reachable and the value of **dist[i][j]** will be less than V.

Example:



Transitive closure of above graphs is

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

Algorithm:

ALGORITHM ***Warshall****(A[1..n, 1..n])*
//Implements Warshall's **algorithm** for computing **the transitive closure**
//**Input:** The **adjacency matrix *A*** of a digraph with *n* vertices
//**Output:** The **transitive closure** of the digraph

$R^{(0)} \leftarrow A$
for k ← 1 **to** *n* **do**
  for i ←1 **to** n **do**
    for *j* ← 1 **to** *n* **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Program:

```
/* Program to find the transitive closure using Warshal's algorithm.*/
#include<stdio.h>
#include<math.h>
void warshal(int p[10][10], int n)
{
   int i, j, k;
   for (k = 1; k <= n; k++)
      for (i = 1; i <= n; i++)
         for (j = 1; j <= n; j++)
            p[i][j] = p[i][j] || (p[i][k] && p[k][j]);
}

void main()
{
   int p[10][10] = { 0 }, n, e, u, v, i, j;
   printf("\n Enter the number of vertices:");
   scanf("%d", &n);
   printf("\n Enter the number of edges:");
   scanf("%d", &e);
   printf("Enter the edges: (u,v)\n");
   for (i = 1; i <= e; i++)
   {
      scanf("%d%d",  &u,&v);
      p[u][v] = 1;
   }
   printf("\n Matrix of input data: \n");
   for (i = 1; i <= n; i++)
   {
      for (j = 1; j <= n; j++)
         printf("%d\t", p[i][j]);
      printf("\n");
   }
   warshal(p, n);
   printf("\n Transitive closure: \n");
   for (i = 1; i <= n; i++)
```

```
    {
        for (j = 1; j <= n; j++)
            printf("%d\t", p[i][j]);
        printf("\n");
    }
}
```

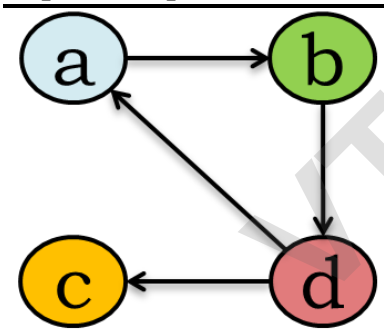
Output:

```
Enter the number of vertices: 4
Enter the number of edges: 4
Enter the edges: (u,v)
1 2
2 4
4 1
4 3

 Matrix of input data:
0       1       0       0
0       0       0       1
0       0       0       0
1       0       1       0

 Transitive closure:
1       1       1       1
1       1       1       1
0       0       0       0
1       1       1       1
```

Input Graph:



Performance:

Time Complexity: O(V³), where V is the number of vertices in the graph and we run three nested loops each of size V.

Program 4: Design and implement C/C++ Program to find shortest paths from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm.

Description:

Dijkstra's algorithm is often considered to be the most straightforward algorithm for solving the shortest path problem.

Dijkstra's algorithm is used for solving **single-source shortest path problems for directed or undirected paths**. Single-source means that **one vertex is chosen to be the start, and the algorithm will find the shortest path from that vertex to all other vertices**.

Dijkstra's algorithm does not work for graphs with negative edges. For graphs with negative edges, the Bellman-Ford algorithm that is described on the next page, can be used instead.

To find the shortest path, Dijkstra's algorithm needs to know which vertex is the source, it needs a way to mark vertices as visited.

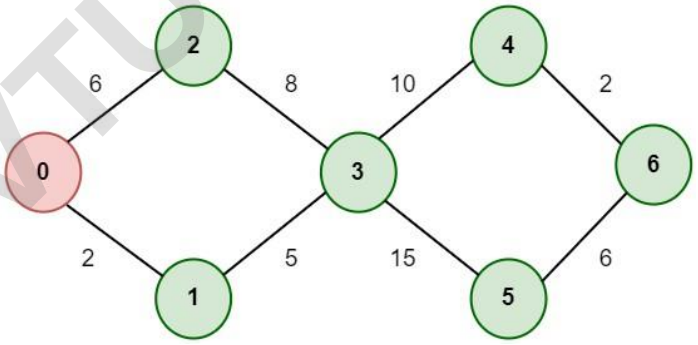
Algorithm for Dijkstra's Algorithm:

Steps:

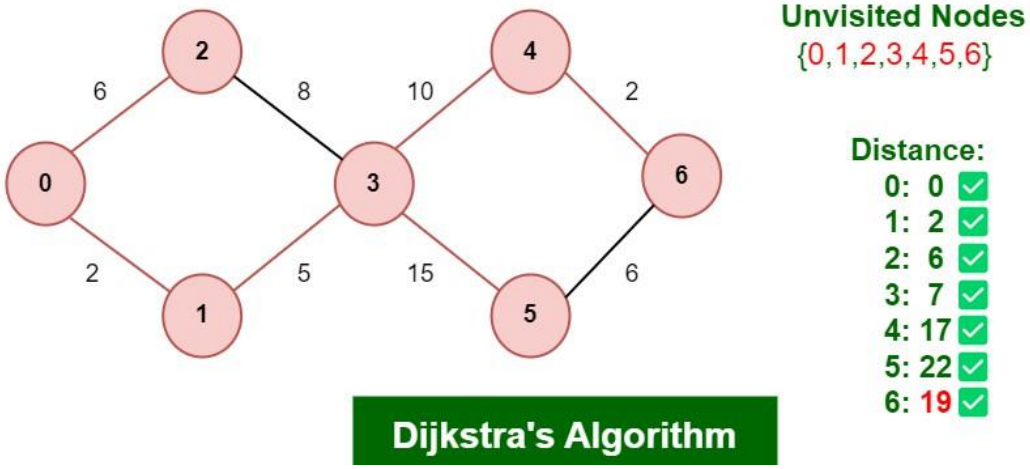
1. Mark the source node with a current distance of **0 and the rest with infinity**.
2. Set the non-visited node with the **smallest current distance as the current node**.
3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

Example:

Input Graph:



Output Graph:



Program:

```
/* Implementation of Dijkstra's Algorithm in C */
#include <stdio.h>
#define INF 9999
#define MAX 10

void DijkstraAlgorithm(int Graph[MAX][MAX], int size, int start)
{
  int cost[MAX][MAX], distance[MAX], previous[MAX];
  int visited_nodes[MAX], counter, minimum_distance, next_node, i, j;
  for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
      if (Graph[i][j] == 0)
        cost[i][j] = INF;
      else
        cost[i][j] = Graph[i][j];

  for (i = 0; i < size; i++)
  {
    distance[i] = cost[start][i];
    previous[i] = start;
    visited_nodes[i] = 0;
  }

  distance[start] = 0;
  visited_nodes[start] = 1;
  counter = 1;

  while (counter < size - 1)
  {
    minimum_distance = INF;
    for (i = 0; i < size; i++)
      if (distance[i] < minimum_distance && !visited_nodes[i])
      {
        minimum_distance = distance[i];
        next_node = i;
      }

    visited_nodes[next_node] = 1;
    for (i = 0; i < size; i++)
      if (!visited_nodes[i])
        if (minimum_distance + cost[next_node][i] < distance[i])
        {
          distance[i] = minimum_distance + cost[next_node][i];
          previous[i] = next_node;
        }
    counter++;
  }
  for (i = 0; i < size; i++)
    if (i != start)
    {
      printf("\nDistance from the Source Node to %d: %d", i, distance[i]);
```

```
    }
}

void main()
{
  int Graph[MAX][MAX], i, j, n, source;
  printf("Enter the number of nodes:\n");
  scanf("%d",&n);
  printf("Enter the cost adjacency Matrix:\n");
  for(i=0;i<n;i++)
  {
    for(j=0;j<n;j++)
    {
        scanf("%d",&Graph[i][j]);
    }
  }
  source = 0;
  DijkstraAlgorithm(Graph, n, source);
 }
```

Output:

```
Enter the number of nodes:
5
Enter the cost adjacency Matrix:
0 3 0 7 0
3 0 4 2 0
0 4 0 5 6
7 2 5 0 4
0 0 6 4 0

Distance from the Source Node to 1: 3
Distance from the Source Node to 2: 7
Distance from the Source Node to 3: 5
Distance from the Source Node to 4: 9
```


Performance Analysis:

The time complexity of Dijkstra's Algorithm is typically $O(V^2)$ when using a simple array implementaion or $O((V + E) \log V)$ with a priority queue, where V represents the number of vertices and E represents the number of edges in the graph.

Program 5: Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.

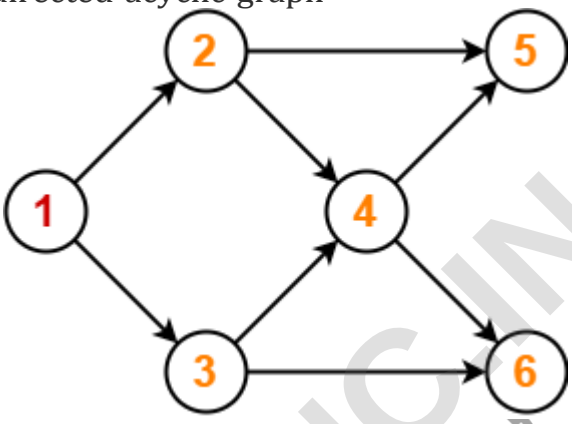
Description:

Topological sorting for **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge u-v, vertex u comes before v in the ordering.

DAGs are a special type of graphs in which each edge is directed such that no cycle exists in the graph, before understanding why Topological sort only exists for DAGs.

Example:

Consider the following directed acyclic graph-



For this graph, following 4 different topological orderings are possible-

- **1 2 3 4 5 6**
- **1 2 3 4 6 5**
- **1 3 2 4 5 6**
- **1 3 2 4 6 5**

Program:

```
/* Program to find the topological ordering of vertices */
#include <stdio.h>
const int MAX = 10;
void fnTopological(int a[MAX][MAX], int n);
void main()
{
    int a[MAX][MAX],n;
    int i,j;

    printf("Topological Sorting Algorithm -\n");
    printf("\nEnter the number of vertices : ");
    scanf("%d",&n);
    printf("Enter the adjacency matrix:\n");
    for (i=0; i<n; i++)
        for    (j=0;    j<n;    j++)
            scanf("%d",&a[i][j]);
    fnTopological(a,n);
    printf("\n");
}

void fnTopological(int a[MAX][MAX], int n)
{
    int in[MAX], out[MAX], stack[MAX], top=-1;
    int i,j,k=0;
    for (i=0;i<n;i++)
```

```
    {
        in[i] = 0;
        for (j=0; j<n; j++)
            if (a[j][i] == 1)
                in[i]++;
    }
    while(1)
    {
        for (i=0;i<n;i++)
        {
            if (in[i] == 0)
            {
                stack[++top] = i;
                in[i] = -1;
            }
        }

        if (top == -1)
            break;

        out[k] = stack[top--];

        for (i=0;i<n;i++)
        {
            if (a[out[k]][i] == 1)
                in[i]--;
        }
        k++;
    }

    printf("Topological Sorting as follows:- \n");
    for (i=0;i<k;i++)
        printf("%d ",out[i] + 1);
}
```

Output:

```
Topological Sorting Algorithm -

Enter the number of vertices: 5
Enter the adjacency matrix:
0 0 1 0 0
0 0 1 0 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0
Topological Sorting as follows:-
2 1 3 4 5
```

Program 6: Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method.

Description:

Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

What is the 0/1 knapsack problem?

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Program:

```
/* Program to solve 0/1 Knapsack problem using Dynamic Programming method.
*/
#include<stdio.h>
int max(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}
int knapsack(int w[], int p[], int n, int M)
{
    if(M==0)
        return 0;
    if(n==0)
        return 0;
    if(w[n-1]>M)
        return knapsack(w,p,n-1,M);
    return   max(knapsack(w,p,n-1,M),p[n-1]+knapsack(w,p,n-1,M-w[n-1]));
}
void main()
{
    int i,n;
    int M;  //capacity of knapsack
    int w[10];  //weight of items
    int p[10];  //value of items
    printf("Enter the no. of items:\n");
    scanf("%d",&n);
    printf("Enter the weight and price of all items:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d%d",&w[i],&p[i]);
    }
    printf("Enter the capacity of knapsack:\n");
    scanf("%d",&M);
```

```
    printf("The maximum value of items that can be put into knapsack is =
%d\n",knapsack(w,p,n,M));
 }
```

Output:

```
Enter the no. of items:
4
Enter the weight and price of all items:
2 12
1 10
3 2
2 15
Enter the capacity of knapsack:
5
The maximum value of items that can be put into knapsack is = 37
```

Performance analysis:

Time Complexity: $O(2^N)$

Program 7: Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.

Description:

Given the weights and profits of N items, in the form of **{profit, weight}** put these items in a **knapsack of capacity W** to get the **maximum total profit** in the knapsack. In Fractional Knapsack, we can **break items for maximizing** the total value of the knapsack.

Example:

Input: arr[] = {{60, 10}, {100, 20}, {120, 30}}, W = 50
Output: 240
Explanation: By taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg.
Hence total price will be 60 +100 + (2/3)(120) = 240

Input: arr[] = {{500, 30}}, W = 10
Output: 166.667

/* Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method. */

```
#include <stdio.h>
void main()
{
   int cur_w,n;
   float tot_v;
   int p[10],w[10],W;
   int i, maxi;
   int used[10];
   printf("Enter the no. of items:\n");
   scanf("%d",&n);
   printf("Enter the weight and price of all items:\n");
   for(i=0;i<n;i++)
   {
      scanf("%d%d",&w[i],&p[i]);
   }
   printf("Enter the capacity of knapsack:\n");
   scanf("%d",&W);
   for (i = 0; i < n; ++i)
     used[i] = 0;
  cur_w = W;
  while (cur_w > 0)
  {
     maxi = -1;
     for (i = 0; i < n; ++i)
        if ((used[i] == 0) &&
             ((maxi == -1) || ((float)w[i]/p[i] > (float)w[maxi]/p[maxi])))
           maxi = i;
     used[maxi] = 1;
     cur_w -= p[maxi];
     tot_v += w[maxi];
     if (cur_w >= 0)
        printf("Added object %d (%d, %d) completely in the bag. Space left:
%d.\n", maxi + 1, w[maxi], p[maxi], cur_w);
     else
```

```
            {
                printf("Added %d%% (%d, %d) of object %d in the bag.\n", (int)((1 +
        (float)cur_w/p[maxi]) * 100), w[maxi], p[maxi], maxi + 1);
                tot_v -= w[maxi];
                tot_v += (1 + (float)cur_w/p[maxi]) * w[maxi];
            }
        }
        printf("Filled the bag with objects worth %.2f.\n", tot_v);
    }
```

Output:

```
Enter the no. of items:
5
Enter the weight and price of all items:
10 3
15 3
10 2
12 5
8 1
Enter the capacity of knapsack:
10
Added object 5 (8, 1) completely in the bag. Space left: 9.
Added object 2 (15, 3) completely in the bag. Space left: 6.
Added object 3 (10, 2) completely in the bag. Space left: 4.
Added object 1 (10, 3) completely in the bag. Space left: 1.
Added 19% (12, 5) of object 4 in the bag.
Filled the bag with objects worth 45.40.
```

Performance Analysis:

Time Complexity: O(N * W) where N is items and W is capacities.

Program 8: Design and implement C/C++ Program to find a subset of a given set S = {sl , s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d.

Description:

Given a set of non-negative integers and a value sum, the task is to check if there is a subset of the given set whose sum is equal to the given sum.

For the recursive approach, there will be two cases.

1. Consider the 'last' element to be a part of the subset. Now the new required sum = required sum – value of 'last' element.
2. Don't include the 'last' element in the subset. Then the new required sum = old required sum.

In both cases, the number of available elements decreases by 1.

Examples:

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 9
Output: True
Explanation: There is a subset (4, 5) with sum 9.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30
Output: False
Explanation: There is no subset that add up to 30.

Program:

```
/* Program to find a subset of a given set S = {sl , s2,..... ,sn} of n positive integers */
#include<stdio.h>
int s[10],d,n,set[10],count=0;
void display(int);
int flag=0;
int subset(int,int);
   void main()
   {

      int i;
      printf("Enter the number of elements in set\n");
      scanf("%d",&n);
      printf("Enter the set values\n");
      for(i=0;i<n;++i)
      scanf("%d",&s[i]);
      printf("Enter the sum\n");
      scanf("%d",&d);
      printf("The program output is\n");
      subset(0,0);
      if(flag==0)
      printf("There is no solution");
   }
   int subset(int sum,int i)
   {
      if(sum==d)
      {
         flag=1;
         display(count);
         return (0);
      }
      if(sum>d||i>=n)
```

```
            return 0;
        else
        {
            set[count]=s[i];
            count++;
            subset(sum+s[i],i+1);
            count--;
            subset(sum,i+1);
        }
    }
    void display(int count)
    {
        int i;
        printf("{");
        for(i=0;i<count;i++)
        printf("%d ",set[i]);
        printf("}");
    }
```

Output 1:

Enter the number of elements in set
5
Enter the set values
1 2 5 6 8
Enter the sum
9
The program output is
{1 2 6}{1 8}

Output 2:

Enter the number of elements in set
5
Enter the set values
1 2 5 6 8
Enter the sum
4
The program output is
There is no solution

Output 3:

Enter the number of elements in set
5
Enter the set values
1 2 5 6 8
Enter the sum
7
The program output is
{1 6}{2 5}

Performance Analysis:

Time Complexity: $O(2^n)$ The above solution may try all subsets of the given set in worst case. Therefore time complexity of the above solution is exponential. The problem is in-fact NP-Complete (There is no known polynomial time solution for this problem).

Program 9: Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Description:

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include<time.h>
void swap(long int*a,long int*b)
{
    int tmp=*a;
    *a=*b;
    *b=tmp;
}
void selectionsort (long int arr[],long int n)
{
    long int i,j,midx;
    for(i=0;i<n-1;i++)
    {
        midx=i;
        for(j=i+1;j<n;j++)
            if(arr[j]<arr[midx])
                midx=j;
        swap(&arr[midx],&arr[i]);
    }
}
void main()
{
    long int n=1000;
    int it=0;
    double tim1[10];
    printf("Input Size, Selection Sorting time \n");
    while(it++<5)
    {
        long int a[n];
        for(int i=0;i<n;i++)
        {
            long int no=rand()%n+1;
            a[i]=no;
        }
        //using clock t to store time
        clock_t start,end;
        start=clock();
        selectionsort(a,n);
        end=clock();
        tim1[it]=(double)(end-start)/1000;
        printf("   %ld  =  %ld  ms\n",n,(long int)tim1[it]);
```

```
            n+=1000;
        }
    }
```

Output:

```
Input Size, Selection Sorting time
   1000      = 1 ms
   2000      = 5 ms
   3000      = 13 ms
   4000      = 23 ms
   5000      = 35 ms
```

Performance Analysis:

Best-case: $O(n^2)$, best case occurs when the array is already sorted. (where n is the number of integers in an array)

Average-case: $O(n^2)$, the average case arises when the elements of the array are in a disordered or random order, without a clear ascending or descending pattern.

Worst-case: $O(n^2)$, The worst-case scenario arises when we need to sort an array in ascending order, but the array is initially in descending order.

Program 10: Design and implement C/C++ Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

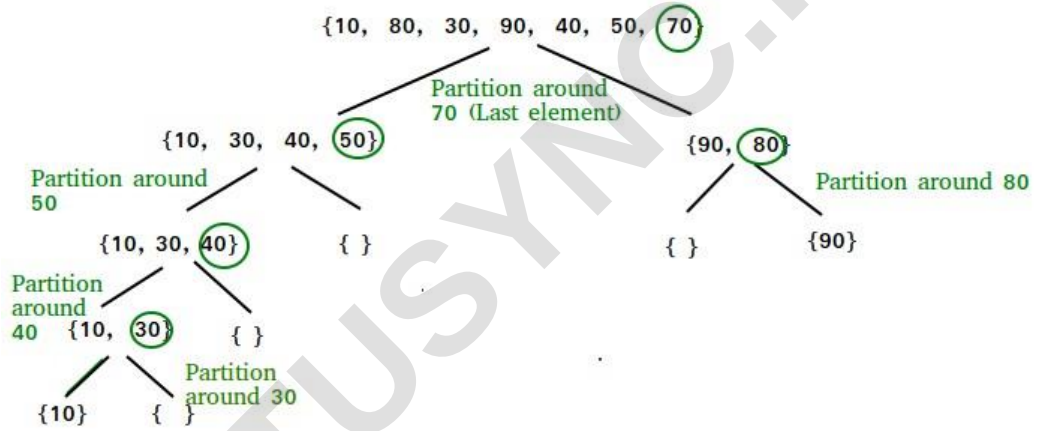
Description:

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort work?

The key process in **quickSort is a partition().** The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.



Program:

```
/* Program to arrange the elements in increasing order */
#include <stdio.h>
#include <stdlib.h>
#include<time.h>

static int max= 5000;
   static int partition(long int arr[],int low,int high)
   {
      int pivot = arr[low];
      int i = low;
      int j= high+1;
      while(i<=j)
      {
      do
      {
      i++;
      }while(pivot>=arr[i] && i<=high);
      do
      {
      j--;
      } while(pivot<arr[j]);
```

```
        if(i<j)
        {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        }
        }
        int temp = arr[low];
        arr[low] = arr[j];
        arr[j] = temp;
        return j;
    }
    static void qs(long int arr[],int low,int high)
    {
        int mid;
        if(low<high)
        {
            mid = partition(arr, low, high);
            qs(arr,low,mid-1);
            qs(arr,mid+1,high);
        }
    }
    void main()
    {
    int n,i;
    long int a[5000], no;
    double tm;
    //using clock t to store time
    clock_t start,end;
    printf("\n Enter the number of elements:\n");
    scanf("%d",&n);

    for(i=0;i<n;i++)
    {
            no=rand()%n+1;
            a[i]=no;
    }
    start=clock();
    qs(a,0,n-1);
    end=clock();
    tm = (end - start);
    printf("    %d = %lf\n Nano Seconds",n,tm);
}
```

Output:

```
Enter the number of elements:
1000
   1000 = 128.000000
 Nano Seconds
```

Time Complexity:

Best Case: Ω (N log (N))
The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.
In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
Average Case: θ (N log (N))
Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
Worst Case: O(N^2)
The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element.

Program 11: Design and implement C/C++ Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Description:

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then it merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Algorithm:

Step 1: Start
Step 2: Declare an array and left, right, mid variable
Step 3: Perform merge function.
mergesort(array,left,right)
mergesort (array, left, right)
if left > right
return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
Step 4: Stop

Program:

```
/* Program to implement Merge Sort */
#include<stdio.h>
#include<time.h>
#include <stdlib.h>
#define max 5000
int array[max];
void merge(int low, int mid, int high)
{
    int temp[max];
    int i = low;
    int j = mid +1;
    int k = low ;
    while((i <= mid) && (j <=high))
    {
        if(array[i] <= array[j])
            temp[k++] = array[i++] ;
        else
            temp[k++] = array[j++] ;
    }
    while( i <= mid )
        temp[k++]=array[i++];
    while( j <= high )
        temp[k++]=array[j++];
    for(i= low; i <= high ; i++)
        array[i]=temp[i];
```

```
}
void merge_sort(int low, int high)
{
        int mid;
        if( low != high )
        {
                mid = (low+high)/2;
                merge_sort(low , mid);
                merge_sort(mid+1, high);
                merge(low, mid, high);
        }
}
void main()
{
        int i,n, no;
        double tm;
        clock_t start,end;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                no=rand()%n+1;
               array[i]=no;
        }
        printf("Unsorted list is :\n");
        for( i = 0 ; i<n ; i++)
                printf("%d ", array[i]);
       start=clock();
        merge_sort(0, n-1);
        printf("\nSorted list is :\n");
        for( i = 0 ; i<n ; i++)
                printf("%d ", array[i]);
        printf("\n");
        end=clock();
    tm = (end - start);
    printf("    %d = %lf  Nano  Seconds  \n",n,tm);
        printf("\n");
}/*End of main()*/
```

Output:

```
Enter the number of elements : 20
Unsorted list is :
4 7 18 16 14 16 7 13 10 2 3 8 11 20 4 7 1 7 13 17
Sorted list is :
1 2 3 4 4 7 7 7 7 8 10 11 13 13 14 16 16 17 18 20
   20 = 26.000000 Nano Seconds
```

Performance Analysis:

Time Complexity: O(nlog(n))

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

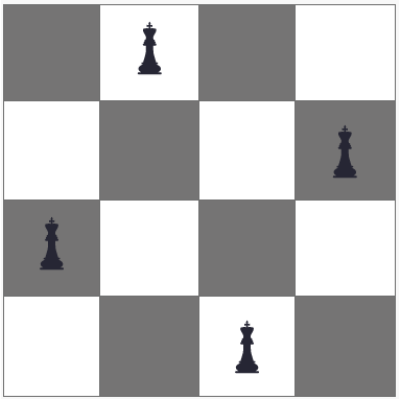
$$T(n) = 2T(n/2) + \theta(n)$$

Program 12: Design and implement C/C++ Program for N Queen's problem using Backtracking.

Description:

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 following is a solution for the 4 Queen problem.



N Queen Problem using Backtracking:

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes **with already placed queens**. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a **row due to clashes**, then we backtrack and return false.

Program:

```
/* Program for N Queen's problem using Backtracking */
#include<stdio.h>
#include<math.h>
int board[20],count;
void main()
{
        int n,i,j;
        void queen(int row,int n);
        printf(" - N Queens Problem Using Backtracking -");
        printf("\n Enter number of Queens:");
        scanf("%d",&n);
        queen(1,n);
}

//function for printing the solution
void print(int n)
{
   int i,j;
   printf("\n\nSolution  %d:\n\n",++count);

   for(i=1;i<=n;++i)
     printf("\t%d",i);

   for(i=1;i<=n;++i)
   {
     printf("\n\n%d",i);
     for(j=1;j<=n;++j)
     {
```

```
            if(board[i]==j)
               printf("\tQ");
            else
               printf("\t*");
       }
    }
}
int place(int row,int column)
{
    int i;
    for(i=1;i<=row-1;++i)
    {
      if(board[i]==column)
       return 0;
      else
       if(abs(board[i]-column)==abs(i-row))
        return 0;
    }
    return 1;
}
void queen(int row,int n)
{
    int column;
    for(column=1;column<=n;++column)
    {
          if(place(row,column))
          {
               board[row]=column;
               if(row==n)
                   print(n);
               else
                   queen(row+1,n);
          }
    }
}
```

Output:

Enter number of Queens: 5
Solution 1:

	1	2	3	4	5
1	Q	*	*	*	*
2	*	*	Q	*	*
3	*	*	*	*	Q
4	*	Q	*	*	*
5	*	*	*	Q	*

Solution 2:

	1	2	3	4	5
1	Q	*	*	*	*
2	*	*	*	Q	*
3	*	Q	*	*	*
4	*	*	*	*	Q
5	*	*	Q	*	*

Solution 3:

	1	2	3	4	5
1	*	Q	*	*	*
2	*	*	*	Q	*
3	Q	*	*	*	*
4	*	*	Q	*	*
5	*	*	*	*	Q

Solution 4:

	1	2	3	4	5
1	*	Q	*	*	*
2	*	*	*	*	Q
3	*	*	Q	*	*
4	Q	*	*	*	*
5	*	*	*	Q	*

4 more solutions are possible.

Performance Analysis:

Time Complexity: O(N!)

MAHARAJA INSTITUTE OF TECHNOLOGY MYSORE
BELAWADI, SRIRANGAPATNA Tq, MANDYA-571477
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
2023-24 (Even) – 4th Semester

Subject: **Analysis & Design of Algorithms Lab** Subject Code: **BCSL404**

Viva Questions

1. What is an algorithm?
Answer: An algorithm is a step-by-step procedure for solving a problem or accomplishing a task.

2. What are the characteristics of a good algorithm?
Answer: A good algorithm should be correct, efficient, and easy to understand.

3. What is the difference between an algorithm and a program?
Answer: An algorithm is a step-by-step procedure for solving a problem, while a program is an implementation of an algorithm in a particular programming language.

4. What is the time complexity of an algorithm?
Answer: The time complexity of an algorithm is a measure of the amount of time it takes to run as a function of the size of the input data.

5. What is the space complexity of an algorithm?
Answer: The space complexity of an algorithm is a measure of the amount of memory it requires as a function of the size of the input data.

6. What is the Big O notation?
Answer: The Big O notation is used to describe the upper bound on the time complexity of an algorithm.

7. What is the worst-case time complexity of an algorithm?
Answer: The worst-case time complexity of an algorithm is the maximum amount of time it takes to run over all possible inputs of a given size.

8. What is the best-case time complexity of an algorithm?
Answer: The best-case time complexity of an algorithm is the minimum amount of time it takes to run over all possible inputs of a given size.

9. What is the average-case time complexity of an algorithm?
Answer: The average-case time complexity of an algorithm is the expected amount of time it takes to run over all possible inputs of a given size.

10. What is the difference between the best-case and worst-case time complexity of an algorithm?
Answer: The best-case time complexity is the minimum amount of time an algorithm can take to run, while the worst-case time complexity is the maximum amount of time an algorithm can take to run.

11. What is the difference between the average-case and worst-case time complexity of an algorithm?
Answer: The worst-case time complexity is the maximum amount of time an algorithm can take to run, while the average-case time complexity is the expected amount of time an algorithm will take to run.

12. What is the difference between time complexity and space complexity?
Answer: Time complexity measures the amount of time an algorithm takes to run, while space complexity measures the amount of memory an algorithm requires.

13. What is a sorting algorithm?
Answer: A sorting algorithm is an algorithm that puts a collection of data items into a specific order, such as alphabetical or numerical order.

14. What is memoization in dynamic programming?
Answer: Memoization is a technique of storing the results of solved subproblems in a table to avoid their repeated calculation in future recursive calls.

15. What is the difference between dynamic programming and divide-and-conquer algorithms?
Answer: Dynamic programming involves solving subproblems and reusing their solutions to solve the main problem, while divide-and-conquer algorithms divide the problem into independent subproblems and solve them separately.

16. What is the time complexity of the brute-force approach?
Answer: The time complexity of the brute-force approach is typically O(n^n) or O(2^n), where n is the size of the input.

17. What is the difference between stable and unstable sorting algorithms?
Answer: Stable sorting algorithms preserve the relative order of equal elements in the input, while unstable sorting algorithms may not.

18. What is the time complexity of the quicksort algorithm?
Answer: The average-case time complexity of the quicksort algorithm is O(n*log*n), where n is the size of the input.

19. What is the difference between in-place and out-of-place sorting algorithms?
Answer: In-place sorting algorithms sort the input array in place without using additional memory, while out-of-place sorting algorithms require additional memory to store the sorted output.

20. What is the time complexity of the mergesort algorithm?
Answer: The time complexity of the mergesort algorithm is O(n*log*n), where n is the size of the input.

21. What is the difference between breadth-first search and depth-first search algorithms?
Answer: Breadth-first search explores the nodes in the graph in a breadth-first order, while depth-first search explores the nodes in a depth-first order.

22. What is the difference between a graph and a tree data structure?
Answer: A tree is a special case of a graph, where there are no cycles, and every pair of nodes is connected by a unique path.

23. What is the time complexity of the Dijkstra's algorithm?
Answer: The time complexity of Dijkstra's algorithm is O(E*log*V), where E is the number of edges and V is the number of vertices in the graph.

24. What is the difference between a directed graph and an undirected graph?
Answer: A directed graph has directed edges, where each edge points from one vertex to another, while an undirected graph has undirected edges, where each edge connects two vertices without any direction.

25. What is the difference between a complete graph and a sparse graph?
Answer: A complete graph has all possible edges between every pair of vertices, while a sparse graph has relatively fewer edges.

26. What is the difference between a greedy algorithm and a dynamic programming algorithm?
Answer: A greedy algorithm makes locally optimal choices at each step, while a dynamic programming algorithm solves subproblems and reuses their solutions to solve the main problem.

27. What is a divide-and-conquer algorithm?
Answer: A divide-and-conquer algorithm is an algorithm that recursively divides a problem into subproblems of smaller size, solves the subproblems, and combines the solutions to solve the original problem.

28. What is a greedy algorithm?
Answer: A greedy algorithm is an algorithm that makes locally optimal choices at each step, hoping to find a globally optimal solution.

29. What is dynamic programming?
Answer: Dynamic programming is an algorithmic technique that solves problems by breaking them down into smaller subproblems and storing the solutions to these subproblems to avoid redundant calculations.

30. What is backtracking?
Answer: Backtracking is an algorithmic technique that involves exploring all possible solutions to a problem by systematically trying different choices, and undoing choices that lead to dead ends.

31. What is recursion?
Answer: Recursion is a programming technique in which a function calls itself to solve a problem.

32. What is the difference between recursion and iteration?
Answer: Recursion involves calling a function from within itself to solve a problem, while iteration involves using loops to repeat a block of code until a condition is met.

33. What is the difference between top-down and bottom-up dynamic programming?
Answer: Top-down dynamic programming involves solving a problem by breaking it down into subproblems, while bottom-up dynamic programming involves solving the subproblems first and combining them to solve the original problem.

34. What is the Knapsack problem?
Answer: The Knapsack problem is a classic optimization problem in which a set of items with different weights and values must be packed into a knapsack of a given capacity, while maximizing the total value of the items.

35. What is the traveling salesman problem?
Answer: The traveling salesman problem is a classic optimization problem in which a salesman must visit a set of cities, each only once, and return to his starting point, while minimizing the total distance traveled.

36. What is the complexity of the brute-force solution to the traveling salesman problem?
Answer: The brute-force solution to the traveling salesman problem has a time complexity of O(n!), where n is the number of cities.

37. What is a graph?
Answer: A graph is a data structure that consists of a set of vertices (nodes) and a set of edges that connect pairs of vertices.

38. What is a directed graph?
Answer: A directed graph is a graph in which each edge has a direction, indicating a one-way connection between two vertices.

39. What is an undirected graph?
Answer: An undirected graph is a graph in which each edge has no direction, indicating a bidirectional connection between two vertices.

40. What is a weighted graph?
Answer: A weighted graph is a graph in which each edge has a weight or cost assigned to it, indicating the cost or distance between the two vertices it connects.

41. What is a cycle in a graph?
Answer: A cycle in a graph is a path that starts and ends at the same vertex, and includes at least one edge.

42. What is a connected graph?
Answer: A connected graph is a graph in which there is a path between every pair of vertices.

43. What is a spanning tree?
Answer: A spanning tree of a graph is a subgraph that includes all the vertices of the graph and forms a tree (a connected acyclic graph).

44. What is the minimum spanning tree of a graph?
Answer: The minimum spanning tree of a graph is the spanning tree with the minimum sum of the weights of its edges.

45. What is Kruskal's algorithm?
Answer: Kruskal's algorithm is a greedy algorithm that finds the minimum spanning tree of a graph by repeatedly adding the next lightest edge that does not form a cycle.

46. What is Prim's algorithm?
Answer: Prim's algorithm is a greedy algorithm that finds the minimum spanning tree of a graph by starting at a random vertex and repeatedly adding the next lightest edge that connects a vertex in the tree to a vertex outside the tree.

47. What is the time complexity of Kruskal's algorithm and Prim's algorithm?
Answer: The time complexity of Kruskal's algorithm and Prim's algorithm is O(E log E), where E is the number of edges in the graph.

48. What is a topological sort?
Answer: A topological sort of a directed acyclic graph is a linear ordering of its vertices such that for every directed edge from vertex u to vertex v, u comes before v in the ordering.

49. What is the time complexity of a topological sort?
Answer: The time complexity of a topological sort is O(V + E), where V is the number of vertices and E is the number of edges in the graph.

50. What is the difference between breadth-first search and topological sort?
Answer: Breadth-first search is a graph traversal algorithm that visits all the vertices in the graph at a given distance (level) from a starting vertex before moving on to vertices at a greater distance. Topological sort, on the other hand, is a way of ordering the vertices of a directed acyclic graph such that for every directed edge u->v, u comes before v in the ordering.

51. What is Dijkstra's algorithm?
Answer: Dijkstra's algorithm is a greedy algorithm that finds the shortest path between a starting vertex and all other vertices in a graph with non-negative edge weights. It works by maintaining a set of vertices for which the shortest path is known, and repeatedly selecting the

vertex with the shortest path and updating the shortest paths to its neighbors.

52. What is the time complexity of Dijkstra's algorithm?
Answer: The time complexity of Dijkstra's algorithm is O((V+E)log V) using a binary heap, where V is the number of vertices and E is the number of edges in the graph.

53. What is the Floyd-Warshall algorithm?
Answer: The Floyd-Warshall algorithm is a dynamic programming algorithm that finds the shortest path between all pairs of vertices in a graph with possibly negative edge weights. It works by maintaining a table of shortest path estimates for all pairs of vertices, and repeatedly updating these estimates based on intermediate vertices.

54. What is the time complexity of the Floyd-Warshall algorithm?
Answer: The time complexity of the Floyd-Warshall algorithm is O(V^3), where V is the number of vertices in the graph.

55. What is the difference between dynamic programming and greedy algorithms?
Answer: Both dynamic programming and greedy algorithms are techniques for solving optimization problems, but they differ in their approach. Dynamic programming involves breaking a problem down into smaller subproblems and solving each subproblem only once, storing the solution in a table to avoid re-computation. Greedy algorithms, on the other hand, make the locally optimal choice at each step, without considering the global optimal solution. Dynamic programming is generally more computationally expensive than greedy algorithms, but can handle more complex optimization problems.

56. What is the principle of optimality?
Answer: The principle of optimality is a key concept in dynamic programming that states that an optimal solution to a problem can be constructed from optimal solutions to its subproblems.

57. What is quicksort algorithm?
Answer: Quicksort is a popular sorting algorithm that uses a divide-and-conquer strategy to sort an array of elements. It works by selecting a pivot element from the array, partitioning the array into two subarrays based on the pivot, and recursively sorting the subarrays. Quicksort has an average case time complexity of O(n log n), making it one of the fastest sorting algorithms in practice.

58. What is merge sort algorithm?
Answer: Merge sort is a popular sorting algorithm that uses a divide-and-conquer strategy to sort an array of elements. It works by dividing the array into two halves, recursively sorting each half, and then merging the two sorted halves together. Merge sort has a worst-case time complexity of O(n log n), making it a good choice for sorting large datasets.

59. What is the difference between quicksort and mergesort?
Answer: Quicksort and mergesort are both popular sorting algorithms that use the divide-and-conquer strategy, but they differ in their approach. Quicksort uses a pivot element to partition the array and sort the subarrays recursively, while mergesort divides the array into two halves and sorts them recursively before merging the sorted halves together. In general, quicksort is faster than mergesort in practice, but mergesort has a more predictable worst-case time complexity.

60. What is dynamic programming used for?
Answer: Dynamic programming is a technique used to solve optimization problems that can be broken down into smaller subproblems. It is often used in problems involving sequence alignment, shortest path finding, knapsack problems, and other optimization problems.

61. What is the time complexity of bubble sort?
Answer: The time complexity of bubble sort is O(n^2), where n is the size of the array being

sorted. Bubble sort works by repeatedly swapping adjacent elements that are out of order, so it needs to make O(n^2) comparisons and swaps in the worst case.

62. What is the time complexity of insertion sort?
Answer: The time complexity of insertion sort is O(n^2), where n is the size of the array being sorted. Insertion sort works by iteratively inserting each element in the proper position in a sorted subarray, so it needs to make O(n^2) comparisons and swaps in the worst case.

63. What is the time complexity of selection sort?
Answer: The time complexity of selection sort is O(n^2), where n is the size of the array being sorted. Selection sort works by repeatedly finding the smallest element in the unsorted portion of the array and swapping it with the first unsorted element, so it needs to make O(n^2) comparisons and swaps in the worst case.

64. What is the time complexity of a linear search?
Answer: The time complexity of a linear search is O(n), where n is the size of the array being searched. Linear search works by iterating through each element in the array until it finds the target element, so it needs to make O(n) comparisons in the worst case.

65. What is the difference between the Dynamic programming and Greedy method?
Answer:

Characteristic	Dynamic Programming	Greedy Method
Approach	Bottom-up approach (start from subproblems and build up to solve the main problem)	Top-down approach (start from the main problem and make locally optimal choices)
Solution quality	Optimal solution guaranteed	Optimal solution not always guaranteed
Subproblem reuse	Subproblems are solved only once and their solutions are stored in a table	Subproblems are not revisited, and the locally optimal choice is made at each step
Solution space	Considers all possible solutions	Considers only the locally optimal solution
Time complexity	Can have a higher time complexity than Greedy Method	Can have a lower time complexity than Dynamic Programming
Suitable problems	Suitable for problems that exhibit optimal substructure and overlapping subproblems	Suitable for problems where making locally optimal choices leads to a global optimal solution
Examples	Fibonacci sequence, Knapsack problem	Coin changing problem, Huffman coding

66. List the advantage of Huffman's encoding?
Answer: Huffman's encoding is a crucial file compression technique that provides the following benefits:
Easy to use
Flexible
Implements optimal and minimum length encoding

67. What is the Algorithm's Time Complexity?
Answer: The time complexity of an algorithm refers to the total amount of time required for the program to run until it finishes.
It is typically expressed using the big O notation.
The algorithm's time complexity indicates the length of time needed for the program to run entirely.
Algorithm's Time Complexity

68. What is a Greedy method in DAA?
Answer: Greedy algorithms solve optimization problems by constructing a solution piece by piece. At each step, they select the next component that provides an immediate benefit without taking prior decisions into account. This method is primarily employed for addressing optimization problems.

69. Can you explain Asymptotic Notation?
Answer: Asymptotic Notation is a mathematical technique used to analyze and describe the behavior of functions as their input size approaches infinity. This notation involves methods whose domains are the set of natural numbers and is useful for defining the worst-case running time function T(n). It can also be extended to the domain of the real numbers.

70. What is the difference between Time Efficiency and Space Efficiency?
Answer: Time Efficiency refers to the measure of the number of times the critical algorithm functions are executed, while Space Efficiency calculates the number of additional memory units utilized by the algorithm.

71. Can you provide an overview of how Merge sort works, and can you give an example of its implementation?
Answer: Merge sort is a sorting algorithm that involves dividing the original list into two smaller sub-lists until only one item is left in each sub-list. These sub-lists are then sorted, and the sorted sub-lists are merged to form a sorted parent list. This process is repeated recursively until the original list is completely sorted.

For example, suppose we have an unsorted list of numbers: [5, 2, 8, 4, 7, 1, 3, 6]. The Merge sort algorithm will first divide the list into two sub-lists: [5, 2, 8, 4] and [7, 1, 3, 6]. Each sub-list will then be recursively divided until only one item is left in each sub-list: [5], [2], [8], [4], [7], [1], [3], [6]. These single-item sub-lists are then sorted and merged pairwise to form new sub-lists: [2, 5], [4, 8], [1, 7], [3, 6]. The process continues recursively until the final sorted list is obtained: [1, 2, 3, 4, 5, 6, 7, 8].

72. Can you explain the concept of Huffman code?
Answer: Huffman code refers to a variable-length encoding technique that involves constructing an optimal prefix tree to assign bit strings to characters based on their frequency in a given text.

73. Can you describe dynamic Huffman coding?
Answer: Dynamic Huffman coding involves updating the coding tree every time a new character is read from the source text. It is an improved version of the simplest Huffman coding technique and is used to overcome its limitations.

74. Can you define the n-queen problem?
Answer: The n-queen problem involves placing n queens on an n-by-n chessboard in a way that none of the queens attack each other by being in the same row, column, or diagonal.