JAIN COLLEGE OF ENGINEERING & RESEARCH, BELAGAVI

DEPT OF COMPUTER SCIENCE & ENGINEERING (AIML)
(ACADEMIC YEAR 2024-25)

LABORATORY MANUAL

SUBJECT: Analysis & Design of Algorithms Lab

SUB CODE: BCSL404

SEMESTER: IV-2022 CBCS Scheme

Prepared By
Prof. Inchara

Approved By
Dr. Pritam D.
HOD, CSE

INSTITUTIONAL MISSION AND VISION

Vision of the Institute

- To become a leading institute that offers quality technical education and research, nurturing professionally competent graduates with strong ethical values.

Mission of the Institute

- Achieve academic excellence through innovative and effective teaching-learning practices.
- Establish industry and academia collaborations to cultivate a culture of research.
- Ensure professional and ethical values to address societal needs.

Department of Computer Science & Engineering(AIML)

Vision of the Department

- To produce competent computer science and engineering graduates by imparting technical knowledge and research to meet industrial needs with moral values.

Mission of the Department

- To facilitate learning opportunities through teaching and mentoring.
- Impart skills in the areas of Computer Science and Engineering through research and industry collaborations.
- To inculcate strong ethical values, understand the societal needs in the field of Computer Science and Engineering.

Program Outcomes (POs)

Engineering Graduates will be able to:

PO1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2. **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.

PO3. **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.

PO4. **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.

PO5. **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.

PO6. **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.

PO7. **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.

PO8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10. **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.

PO11. **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.

PO12. 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.

Program Specific Outcomes (PSOs)

PSO1: Apply mathematical and scientific skills in the area of computer science and engineering to design, develop, analyze software and hardware based systems.

PSO2: Provide solutions using networking and database administration skills, and address the needs of environmental and societal issues through entrepreneurial practices.

Program Educational Objectives (PEOs):

1. To produce graduates who have a strong foundation of knowledge and Engineering skills that will enable them to have successful career in the field of computer science and engineering.
2. To expose graduates to professional and team building skills along with ideas of innovation and invention.
3. To prepare graduates with an ethics, social responsibilities, and professional concerns.

Analysis & Design of Algorithms Lab		Semester	4
Course Code	**BCSL404**	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	0:0:2:0	SEE Marks	50
Credits	01	Exam Hours	2
Examination type (SEE)	Practical		

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.

Sl.No	Experiments
1	Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.
2	Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.
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.
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.
5	Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.
6	Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method.
7	Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.
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.
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.
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.
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.
12	Design and implement C/C++ Program for N Queen's problem using Backtracking.

Course outcomes (Course Skill Set):
At the end of the course the student will be able to:

1. Develop programs to solve computational problems using suitable algorithm design strategy.
2. Compare algorithm design strategies by developing equivalent programs and observing running times for analysis (Empirical).
3. Make use of suitable integrated development tools to develop programs
4. Choose appropriate algorithm design techniques to develop solution to the computational and complex problems.
5. Demonstrate and present the development of program, its execution and running time(s) and record the results/inferences.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation (CIE):
CIE marks for the practical course are **50 Marks**.
The split-up of CIE marks for record/ journal and test are in the ratio **60:40**.

- Each experiment is to be evaluated for conduction with an observation sheet and record write-up. Rubrics for the evaluation of the journal/write-up for hardware/software experiments are designed by the faculty who is handling the laboratory session and are made known to students at the beginning of the practical session.
- Record should contain all the specified experiments in the syllabus and each experiment write-up will be evaluated for 10 marks.
- Total marks scored by the students are scaled down to **30 marks** (60% of maximum marks).
- Weightage to be given for neatness and submission of record/write-up on time.
- Department shall conduct a test of 100 marks after the completion of all the experiments listed in the syllabus.
- In a test, test write-up, conduction of experiment, acceptable result, and procedural knowledge will carry a weightage of 60% and the rest 40% for viva-voce.
- The suitable rubrics can be designed to evaluate each student's performance and learning ability.
- The marks scored shall be scaled down to **20 marks** (40% of the maximum marks).

The Sum of scaled-down marks scored in the report write-up/journal and marks of a test is the total CIE marks scored by the student.

Semester End Evaluation (SEE):

- SEE marks for the practical course are 50 Marks.
- SEE shall be conducted jointly by the two examiners of the same institute, examiners are appointed by the Head of the Institute.
- The examination schedule and names of examiners are informed to the university before the conduction of the examination. These practical examinations are to be conducted between the schedule mentioned in the academic calendar of the University.
- All laboratory experiments are to be included for practical examination.

- (Rubrics) Breakup of marks and the instructions printed on the cover page of the answer script to be strictly adhered to by the examiners. OR based on the course requirement evaluation rubrics shall be decided jointly by examiners.
- Students can pick one question (experiment) from the questions lot prepared by the examiners jointly.
- Evaluation of test write-up/ conduction procedure and result/viva will be conducted jointly by examiners.
- General rubrics suggested for SEE are mentioned here, writeup-20%, Conduction procedure and result in - 60%, Viva-voce 20% of maximum marks. SEE for practical shall be evaluated for 100 marks and scored marks shall be scaled down to 50 marks (however, based on course type, rubrics shall be decided by the examiners)
- Change of experiment is allowed only once and 15% of Marks allotted to the procedure part are to be made zero.
- The minimum duration of SEE is 02 hours

CONTENTS

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

Kruskal's algorithm:

Kruskal’s algorithm finds the minimum spanning tree for a weighted connected graph G=(V,E) to get an acyclic subgraph with |V|-1 edges for which the sum of edge weights is the smallest. Consequently the algorithm constructs the minimum spanning tree as an expanding sequence of subgraphs, which are always acyclic but are not necessarily connected on the intermediate stages of algorithm. The algorithm begins by sorting the graph’s edges in non decreasing order of their weights. Then starting with the empty subgraph, it scans the sorted list adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

```
Algorithm : Kruskal(G)
// Kruskal’s algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G=(V,E)
//Output: $E_T$, the set of edges composing a minimum spanning tree of G
{
        Sort E in non decreasing  order of the edge weights $w(e_{i1}) <= \ldots\ldots >= w(e_{i\,|E|})$
        $E_T \leftarrow \varnothing$ ; ecounter ← 0 //Initialize the set of tree edges and
        its size k ← 0 //initialize the number of processed edges
        while ecounter <|V|-1 do
        {
                k ←     k+1
                if $E_T U \{e_{ik}\}$ is acyclic
                        $E_T \leftarrow E_T U \{e_{ik}\}$; ecounter ← ecounter+1
        }
        return $E_T$
}
```

Complexity: With an efficient sorting algorithm, the time efficiency of kruskal’s algorithm will be in O(|E| log |E|).

Program:

```
#include<stdio.h>
int ne=1,min_cost=0;

void main()
{
        int n,i,j,min,a,u,b,v,cost[20][20],parent[20];

        printf("Enter the no. of vertices:");
        scanf("%d",&n);

        printf("\nEnter the cost matrix:\n");
        for(i=1;i<=n;i++)
                for(j=1;j<=n;j++)
                        scanf("%d",&cost[i][j]);

        for(i=1;i<=n;i++)
                parent[i]=0;

        printf("\nThe edges of spanning tree are\n");

        while(ne<n)
        {
                min=999;
                for(i=1;i<=n;i++)
                {
                        for(j=1;j<=n;j++)
                        {
                                if(cost[i][j]<min)
                                {
                                        min=cost[i
                                        ][j]; a=u=i;
                                        b=v=j;
                                }
                        }
                }
                while(parent[u])
                       u=parent[u];

                while(parent[v])
                        v=parent[v];

                if(u!=v)
                {
                        printf("Edge %d\t(%d->%d)=%d\n",ne++,a,b,min);
                        min_cost=min_cost+min;
                        parent[v]=u;
                }
                cost[a][b]=cost[a][b]=999;
        }
        printf("\nMinimum  cost=%d\n",min_cost);
```

OUTPUT:

```
Enter the no. of vertices:6

Enter the cost matrix:
999 3 999 999 6 5
3 999 1 999 999 4
999 1 999 6 999 4
999 999 6 999 8 5
6 999 999 8 999 2
5 4 4 5 2 999

The edges of spanning tree are
Edge 1 (2->3)=1
Edge 2 (5->6)=2
Edge 3 (1->2)=3
Edge 4 (2->6)=4
Edge 5 (4->6)=5

Minimum cost=15
```

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

Prim's Algorithm:

Prim's algorithm finds the minimum spanning tree for a weighted connected graph G=(V,E) to get an acyclic subgraph with |V|-1 edges for which the sum of edge weights is the smallest. Consequently the algorithm constructs the minimum spanning tree as expanding sub-trees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed.

```
Algorithm : Prim(G)
// Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G=(V,E)
//Output: $E_T$,the set of edges composing a minimum spanning tree of G
{
      $V_T \leftarrow \{v_0\}$  //the set of tree vertices can be initialized with any
      vertex $E_T \leftarrow \varnothing$

      for i ← 0 to |V| - 1 do
          find a minimum-weight edge e* = (v*, u*) among all the edges (v,
          u) such that v is in $V_T$ and u is in $V-V_T$
          $V_T \leftarrow V_T \cup \{u^*\}$
          $E_T \leftarrow E_T \cup \{e^*\}$

      return $E_T$
}
```

Complexity: The time efficiency of prim's algorithm will be in O(|E| log |V|).

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:");
       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();
}
```

OUTPUT:

```
Enter the No of nodes or vertices:4


 Enter the Cost Adjacency matrix of the given graph:

0 1 5 2

1 0 0 0

5 0 0 3

2 0 3 0


 The Minimum Cost Spanning tree is...

(1,2)-->1

(4,1)-->2

(3,4)-->3

 The Required Mincost of the Spanning Tree is:6
```

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

Floyd's Algorithm:

Floyd's algorithm is applicable to both directed and undirected graphs provided that they do not contain a cycle. It is convenient to record the lengths of shortest path in an n- by- n matrix D called the distance matrix. The element d_{ij} in the ith row and jth column of matrix indicates the shortest path from the ith vertex to jth vertex (1<=i, j<=n). The element in the ith row and jth column of the current matrix $D^{(k-1)}$ is replaced by the sum of elements in the same row i and kth column and in the same column j and the kth column if and only if the latter sum is smaller than its current value.

```
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
//Output: The distance matrix of shortest paths length
{
        D ← W
        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
}
```

Complexity: The time efficiency of Floyd's algorithm is cubic i.e. Θ (n^3)

Program:

```
#include<stdio.h>

void floyd(int[10][10],int);
int min(int,int);

void main()
{
        int n,a[10][10],i,j;
        printf("Enter the no.of nodes : ");
        scanf("%d",&n);

        printf("\nEnter the cost adjacency matrix\n");
        for(i=1;i<=n;i++)
                for(j=1;j<=n;j++)
                        scanf("%d",&a[i][j]);
        floyd(a,n);
}

void floyd(int a[10][10],int n)
{
        int d[10][10],i,j,k;

for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                        d[i][j]=a[i][j];
        }

for(k=1;k<=n;k++)
        {
                for(i=1;i<=n;i++)
                {
                        for(j=1;j<=n;j++)
                        {
                                d[i][j]=min(d[i][j],d[i][k]+d[k][j]);
                        }
                }
        }

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

```
int min (int a,int b)
{
        if(a<b)
                return a;
        else
                return b;
}
```

OUTPUT:

```
Enter the no.of nodes : 4

Enter the cost adjacency matrix
0 999 3 999
2 0 999 999
999 7 0 1
6 999 999 0

The distance matrix is
0       10      3       4
2        0      5       6
7        7      0       1
6       16      9       0
```

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

Warshall's algorithm:

The transitive closure of a directed graph with n vertices can be defined as the n-by-n boolean matrix T={t_{ij}}, in which the element in the ith row(1<=i<=n) and jth column(1<=j<=n) is 1 if there exists a non trivial directed path from ith vertex to jth vertex, otherwise, t_{ij} is 0.

Warshall's algorithm constructs the transitive closure of a given digraph with n vertices through a series of n-by-n boolean matrices: $R^{(0)}$,….,$R^{(k-1)}$, $R^{(k)}$,….,$R^{(n)}$ where, $R^{(0)}$ is the adjacency matrix of digraph and $R^{(1)}$ contains the information about paths that use the first vertex as intermediate. In general, each subsequent matrix in series has one more vertex to use as intermediate for its path than its predecessor. The last matrix in the series $R^{(n)}$ reflects paths that can use all n vertices of the digraph as intermediate and finally transitive closure is obtained. The central point of the algorithm is that we compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor R $^{(k-1)}$ in series.

```
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 digraph
{
        $R^{(0)} \leftarrow$ A
        for  k $\leftarrow$ 1 to n do
        {
                for  i $\leftarrow$ 1 to n do
                {
                     for j $\leftarrow$ 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 $R^{(n)}$
}
```

Complexity: The time efficiency of Warshall's algorithm is in Θ (n^3).

Program:

```
#include<stdio.h>

void warshall(int[10][10],int);

void main()
{
        int a[10][10],i,j,n;

        printf("Enter the number of nodes:");
        scanf("%d",&n);

        printf("\nEnter the adjacency matrix:\n");
        for(i=1;i<=n;i++)
                for(j=1;j<=n;j++)
                        scanf("%d",&a[i][j]);

        printf("The adjacency matirx is:\n");
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                {
                        printf("%d\t",a[i][j]);
                }
                printf("\n");
}
        warshall(a,n);
}

void warshall(int p[10][10],int n)
{
        int i,j,k;

        for(k=1;k<=n;k++)
        {
                for(j=1;j<=n;j++)
                {
                        for(i=1;i<=n;i++)
                        {
                                if((p[i][j]==0) && (p[i][k]==1) && (p[k][j]==1))
                                {
                                        p[i][j]=1;
                                }
                        }
                }
        }
        printf("\nThe path matrix is:\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 nodes:5

Enter the adjacency matrix:
0 1 1 0 1
0 0 1 0 0
0 0 0 1 0
0 0 0 0 0
0 0 0 1 0
The adjacency matirx is:
0       1       1       0       1
0       0       1       0       0
0       0       0       1       0
0       0       0       0       0
0       0       0       1       0

The path matrix is:
0       1       1       1       1
0       0       1       1       0
0       0       0       1       0
0       0       0       0       0
0       0       0       1       0
```

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

Single Source Shortest Paths Problem:

For a given vertex called the source in a weighted connected graph, find the shortest paths to all its other vertices. Dijkstra's algorithm is the best known algorithm for the single source shortest paths problem. This algorithm is applicable to graphs with nonnegative weights only and finds the shortest paths to a graph's vertices in order of their distance from a given source. It finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. It is applicable to both undirected and directed graphs

```
Algorithm : Dijkstra(G,s)
//Dijkstra's algorithm for single-source shortest paths
//Input :A weighted connected graph G=(V,E) with nonnegative weights and its
vertex s
//Output : The length dv of a shortest path from s to v and its penultimate vertex
pv for
//every v in V.
{
        Initialise(Q)     // Initialise vertex priority queue to
        empty for every vertex v in V do
        {
                dv←œ; pv←null
                Insert(Q,v,dv)   //Initialise vertex priority queue in the priority
                queue
        }

        ds←0;   Decrease(Q,s ds)        //Update priority of s with ds
        Vt←Ø
        for i←0 to |v|-1 do
        {
                u* ←
                DeleteMin(Q)            //delete the minimum priority element
                Vt ←Vt U {u*}

                for every vertex u in V-Vt that is adjacent to u* do
                {
                        if du* + w(u*,u)<du
                        {
                                du←du* + w(u*, u):  pu←u*
                                Decrease(Q,u,du)
                        }
                }



        }
}
```

Complexity: The Time efficiency for graphs represented by their weight matrix and the priority queue implemented as an unordered array and for graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is O(|E| log |V|).

Program:

```
#include<stdio.h>
void readf();
void SP();
int cost[20][20],dist[20],s[20];
int n,u,min,v,w;

void readf()
{
   int i,j;
   printf("\n Enter the no of vertices:");
   scanf("%d",&n);
   printf("\n Enter the Cost of
   vertices:");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;
     }
   }
}
void SP()
{
   int i,j;
   printf("\n Enter the source vertex:");
   scanf("%d",&v);
   for(i=1;i<=n;i++)
   {
      s[i]=0;
      dist[i]=cost[v][i];
   }
   s[v]=1;
   dist[v]=0;
   for(i=2;i<=n;i++)
   {
      min=dist[i];
      for(j=2;j<=n;j++)
      {
             if(s[j]==0)


             {
                if(min>dist[j])
                {
                   min=dist[j
                   ];u=j;
                }
              }
      }
      s[u]=1;
      for(w=1;w<=n;w++)
      {
              if(cost[u][w]!=0 && s[w]==0)
              {
                 if(dist[w]>(dist[u]+cost[u][w]))
                 {
                    dist[w]=dist[u]+cost[u][w];
                 }
              }
      }
   }
   printf("\n From the Source vertex %d",v);
   for(i=1;i<=n;i++)
       printf("\n%d->%d = %d",v,i,dist[i]);
}
void main()
{
   readf();
   SP();
}
```

OUTPUT:

```
Enter the no of vertices:6

 Enter the Cost of vertices:999 3 999  999 6 5
3 999 1 999 999 4
999 1 999 6 999 4
999 999 6 999 8 5
6 999 999 8 999 2
5 4 4 5 2 999

 Enter the source vertex:1

 From the Source vertex 1
1->1=0
1->2=3
1->3=4
1->4=10
1->5=6
1->6=5
```

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

Topological Ordering:

This method is based on decrease and conquer technique a vertex with no incoming edges is selected and deleted along with no incoming edges is selected and deleted along with the outgoing edges. If there are several vertices with no incoming edges arbitrarily a vertex is selected. The order in which the vertices are visited and deleted one by one results in topological sorting.

```
Algorithm  topological_sort(a,n,T)
        //purpose :To obtain the sequence of jobs to be executed
          resut In topolocical order
        // Input:a-adjacency matrix of the given graph
        //n-the number of vertices in the graph
        //output:
        // T-indicates the jobs that are to be executed in the order

              For j<-0 to n-1 do
                     Sum-0
                   For  i<- 0to n-1 do
                     Sum<-sum+a[i][j]
                End for
                   Top ← -1
                    For i<- 0 to n-1 do
                          If(indegree [i]=0)
                                Top <-top+1
                                S[top]<- i
                   End if
                    End for
          While top!= 1
               u<-s[top]
              top<-top-1
      Add u to solution
                vector T
               For each vertex v adjacent to u
                      Decrement indegree [v] by one
                          If(indegree  [v]=0)
                          Top<-top+1
                          S[top]<-v
        End if
        End for
        End
        while
        Write T
        return
```

1) Find the vertices whose indegree is zero and place them on the stack
2) Pop a vertex u and it is the task to be done
3) Add the vertex u to the solution vector
4) Find the vertices v adjacent to vertex u. The vertices v represents the jobs which depend on job u
5) Decrement indegree[v] gives the number of depended jobs of v are reduced by one.

Complexity: Complexity of topological sort is given by o[V*V].

Program:

```
/*To obtain the topological order in of vertices in a digraph.*/

#include<stdio.h>

void findindegree(int [10][10],int[10],int);
void topological(int,int [10][10]);

void main()
{
        int a[10][10],i,j,n;

        printf("Enter the number of nodes:");
        scanf("%d",&n);

        printf("\nEnter the adjacency matrix\n");
        for(i=1;i<=n;i++)
                for(j=1;j<=n;j++)
                        scanf("%d",&a[i][j]);

        printf("\nThe adjacency matirx is:\n");
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                {
                        printf("%d\t",a[i][j]);
                }
                printf("\n");
        }
        topological(n,a);
}

void findindegree(int a[10][10],int indegree[10],int n)
{
        int i,j,sum;

        for(j=1;j<=n;j++)
        {
                sum=0;

                for(i=1;i<=n;i++)
                {
                        sum=sum+a[i][j];
                }
                indegree[j]=sum;
        }
}

void topological(int n,int a[10][10])
{
        int k,top,t[100],i,stack[20],u,v,indegree[20];
```

```
	k=1;
	top=-1;

	findindegree(a,indegree,n);

	for(i=1;i<=n;i++)
	{
		if(indegree[i]==0)
		{
			stack[++top]=i;
		}
	}

	while(top!=-1)
	{
		u=stack[top--];
		t[k++]=u;

		for(v=1;v<=n;v++)
		{
			if(a[u][v]==1)
			{
				indegree[v]--;
				if(indegree[v]==0)
				{
					stack[++top]=v;
				}
			}
		}
	}

	printf("\nTopological sequence is\n");
	for(i=1;i<=n;i++)
		printf("%d\t",t[i]);
}
```

OUTPUT:

```
Enter the number of nodes: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

The adjacency matirx is:
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 sequence is
2       1      3      4      5
```

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

<u>0/1 Knapsack problem:</u>

Given: A set S of n items, with each item i having

- b_i - a positive benefit
- w_i - a positive weight

Goal: Choose items with maximum total benefit but with weight at most W. i.e.

- Objective: maximize $\sum_{i \in T} b_i$
- Constraint: $\sum_{i \in T} w_i \leq W$

```
Algorithm: 0/1Knapsack(S, W)
//Input: set S of items with benefit b_i and weight w_i; max. weight W
//Output: benefit of best subset with weight at most W
// Sk: Set of items numbered 1 to k.
//Define B[k,w] = best selection from Sk with weight exactly equal to w
{
    for w< -0 to n-1 do
            B[w] <- 0
    for k <-1 to n do
    {
        for w <-W downto w_k do
        {
            if B[w-w_k]+b_k > B[w] then
                B[w] <- B[w-w_k]+b_k
        }
    }
}
```

Complexity: The Time efficiency and Space efficiency of 0/1 Knapsack algorithm is Ɵ(nW).

Program:

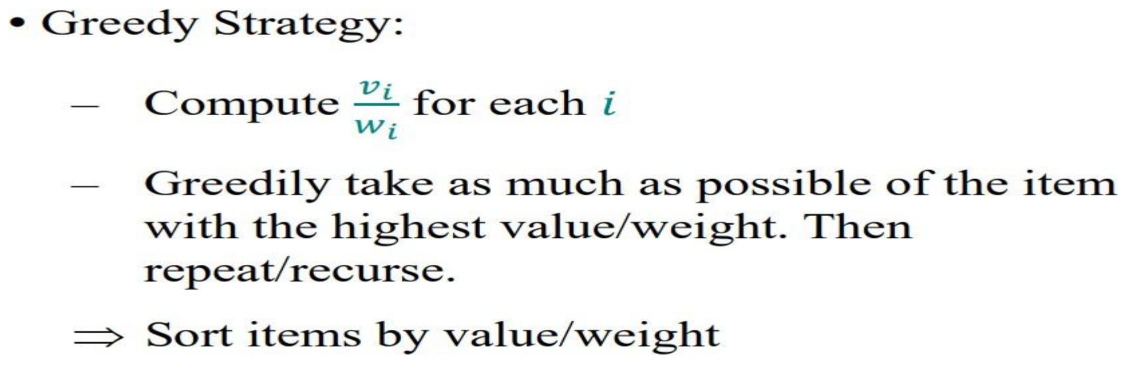
```
#include<stdio.h>
#define MAX 50

int p[MAX],w[MAX],n;
int knapsack(int,int);
int max(int,int);

void main()
{
        int m,i,optsoln;

        printf("Enter no. of objects: ");
        scanf("%d",&n);

        printf("\nEnter the weights:\n");
        for(i=1;i<=n;i++)
                scanf("%d",&w[i]);

        printf("\nEnter the profits:\n");
        for(i=1;i<=n;i++)
                scanf("%d",&p[i]);

        printf("\nEnter the knapsack capacity:");
        scanf("%d",&m);

        optsoln=knapsack(1,m);

        printf("\nThe optimal soluntion is:%d",optsoln);
}

int knapsack(int i,int m)
{
        if(i==n)
                return (w[n]>m) ? 0 : p[n];

        if(w[i]>m)
                return knapsack(i+1,m);

        return max(knapsack(i+1,m),knapsack(i+1,m-w[i])+p[i]);
}

int max(int a,int b)
{
        if(a>b)
                return a;
        else
                return b;
}
```

OUTPUT:

```
Enter no. of objects: 3

Enter the weights:
100 14 10

Enter the profits:
20 18 15

Enter the knapsack capacity:116

The optimal soluntion is:38
```

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

Algorithm:



- Greedy Strategy:
  - Compute $\frac{v_i}{w_i}$ for each i
  - Greedily take as much as possible of the item with the highest value/weight. Then repeat/recurse.

  $\Rightarrow$ Sort items by value/weight

Complexity: The runtime complexity of greedy Knapsack algorithm is O(nlogn).

```
#include<stdio.h>
int main()
{
     float weight[50],profit[50],ratio[50],Totalvalue,temp,capacity,amount;
     int n,i,j;
     printf("Enter the number of items :");
     scanf("%d",&n);
    for (i = 0; i < n; i++)
    {
       printf("Enter Weight and Profit for item[%d] :\n",i);
       scanf("%f %f", &weight[i], &profit[i]);
    }
    printf("Enter the capacity of knapsack :\n");
    scanf("%f",&capacity);

     for(i=0;i<n;i++)
         ratio[i]=profit[i]/weight[i];

    for (i = 0; i < n; i++)
      for (j = i + 1; j < n; j++)
         if (ratio[i] < ratio[j])
        {
            temp = ratio[j];
            ratio[j] = ratio[i];
            ratio[i] = temp;

            temp = weight[j];
            weight[j] = weight[i];
            weight[i] = temp;

            temp = profit[j];
            profit[j] = profit[i];
            profit[i] = temp;
         }

    printf("Knapsack problems using Greedy Algorithm:\n");
    for (i = 0; i < n; i++)
    {
     if (weight[i] > capacity)
         break;
      else
     {
         Totalvalue = Totalvalue + profit[i];
         capacity = capacity - weight[i];
      }
    }
      if (i < n)
      Totalvalue = Totalvalue + (ratio[i]*capacity);
    printf("\nThe maximum value is :%f\n",Totalvalue);
    return 0;
}
```

OUTPUT:-

```
Enter the number of items :4
Enter Weight and Profit for item[0] :
2
12
Enter Weight and Profit for item[1] :
1
10
Enter Weight and Profit for item[2] :
3
20
Enter Weight and Profit for item[3] :
2
15
Enter the capacity of knapsack :
5
Knapsack problems using Greedy Algorithm:

The maximum value is :38.333332
```

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.

<u>Sum of Subsets</u>

Subset-Sum Problem is to find a subset of a given set S= {s1, s2… sn} of n positive integers whose sum is equal to a given positive integer d. It is assumed that the set's elements are sorted in increasing order. The state-space tree can then be constructed as a binary tree and applying backtracking algorithm, the solutions could be obtained. Some instances of the problem may have no solutions

```
Algorithm SumOfSub(s, k, r)
//Find all subsets of w[1…n] that sum to m. The values of x[j], 1<= j < k, have already
//been determined. s=$\sum_{j=1}^{k-1}$ w[j]*x[j]  and r =$\sum_{j=k}^{n}$ w[j]. The w[j]'s are in ascending order.
{
    x[k] ← 1  //generate left child
    if (s+w[k] = m)
            write (x[1...n]) //subset found
    else if ( s + w[k]+w[k+1] <= m)
            SumOfSub( s + w[k], k+1, r-w[k])
    //Generate right child
    if( (s + r - w[k] >= m) and (s + w[k+1] <= m) )
    {
            x[k] ← 0
            SumOfSub( s, k+1, r-w[k] )
    }
}
```

Complexity:

Subset sum problem solved using backtracking generates at each step maximal two new subtrees, and the running time of the bounding functions is linear, so the running time is O(2^n).

Program:

```
#include<stdio.h>
void subset(int,int,int);
int x[10],w[10],d,count=0;

void main()
{
int i,n,sum=0;
printf("Enter the no. of elements: ");
scanf("%d",&n);
printf("\nEnter the elements in ascending order:\n");
for(i=0;i<n;i++)
scanf("%d",&w[i]);
printf("\nEnter the sum: ");
scanf("%d",&d);
for(i=0;i<n;i++)
sum=sum+w[i];
if(sum<d)
{
printf("No solution\n");
return;
}
subset(0,0,sum);
if(count==0)
{
printf("No solution\n");
return;
}

}

void subset(int cs,int k,int r)
{
int i; x[k]=1;
if(cs+w[k]==d)
{
printf("\n\nSubset %d\n",++count); for(i=0;i<=k;i++)
if(x[i]==1)
printf("%d\t",w[i]);
}
else
 if(cs+w[k]+w[k+1]<=d)
   subset(cs+w[k],k+1,r-w[k]);
  if(cs+r-w[k]>=d && cs+w[k]<=d)
   {
      x[k]=0;
       subset(cs,k+1,r-w[k]);
   }
}
```

OUTPUT:

```
Enter the no. of elements: 5

Enter the elements in ascending order:
1 2 5 6 8

Enter the sum: 9


Subset 1
1       2       6

Subset 2
1       8
```

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.

Selection sort:
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundaries by one element to the right.

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

Complexity: its average and worst case complexities are of **O(n^2)**, where **n** is the number of items.

Program:

```
#include <stdio.h>
#include<time.h>

// function to swap the the position of two elements
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
  int min_idx = step;
    for (int i = step + 1; i < size; i++) {

      // To sort in descending order, change > to < in this line.
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])
        min_idx = i;
    }

    // put min at the correct position
    swap(&array[min_idx], &array[step]);
  }
}

// function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
  printf("%d  ", array[i]);
  }
  printf("\n");
}

// driver code
int main() {
  int data[] = {20, 12, 10, 15, 2};
  clock_t s,e;
  int size = sizeof(data) / sizeof(data[0]);
  s=clock();
  selectionSort(data, size);
  e=clock();
  printf("Sorted array in Acsending Order:\n");
  printArray(data, size);
  printf("\nTime taken:%f",(e-s)/CLOCKS_PER_SEC);

}
```

OUTPUT:

```
Sorted array in Acsending Order:
2  10  12  15  20

Time taken:0.000000
```

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.

Quick Sort Method:

Quick Sort divides the array according to the value of elements. It rearranges elements of a given array A[0..n-1] to achieve its partition, where the elements before position s are smaller than or equal to A[s] and all the elements after position s are greater than or equal to A[s].

A[0]…A[s-1] A[s] A[s+1]…A[n-1]

$\longleftrightarrow$ $\longleftrightarrow$

All are <=A[s] all are >=A[s]

```
Algorithm : QUICKSORT(a[l..r])
//Sorts a subarray by quicksort
//Input: A subarray A[l..r] of A[0..n-1],defined by its left and right indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
{
        if l<r
        {
                s← Partition(A[l..r])  //s is a split
                position QUICKSORT(A[l..s-1])
                QUICKSORT(A[s+1..r])
        }
}

Algorithm : Partition(A[l..r])
//Partition a subarray by using its  first element as its pivot
//Input:A subarray A[l..r] of A[0..n-1],defined by its left and right indices l and r (l<r)
//Output:A partition of A[l..r],with the split position returned as this function's value
{
        p ←  A[l]
        i ← l; j ←r+1
        repeat
        {       repeat i ← i+1 until A[i] >=p
                        repeat j ← j-1 until A[j] <=p
                swap(A[i],A[j])
        } until i>=j
        swap(A[i],A[j]) // undo last swap when i>=j
        swap(A[l],A[j])
        return j
}
```

Complexity: $C_{best}(n) = 2\ C_{best}(n/2) + n$

for n>1 $C_{best}(1) = 0$

$C_{worst}(n) \in \theta(n^2)$

$$C_{avg}(n) \approx 1.38n\log_2 n$$

Program:

```
/*To sort a set of elements using Quick sort algorithm.*/

#include<stdio.h>
#include<time.h>
#define max 500

void qsort(int [],int,int);
int partition(int [],int,int);

void main()
{
        int a[max],i,n;
        clock_t s,e;

        printf("Enter the value of n:");

        scanf("%d",&n);

         printf("Enter the array elements");

        for(i=0;i<n;i++)

        scanf("%d",&a[i]);

        printf("\nThe array elements before\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);

        s=clock();

        qsort(a,0,n-1);
        e=clock();

        printf("\nElements of the array after sorting are:\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
        printf("\nTime  taken:%f",(e-s)/CLOCKS_PER_SEC);
}

void qsort(int a[],int low,int high)
{
        int j;
        if(low<high)
        {
                j=partition(a,low,high);
                qsort(a,low,j-1);
                qsort(a,j+1,high);
        }
}

int partition(int a[], int low,int high)
{
        int pivot,i,j,temp;

        pivot=a[low];
        i=low+1;
        j=high;
        while(1)
        {
                while(pivot>a[i] && i<=high)
                        i++;
                while(pivot<a[j])
                        j--;
                if(i<j)
                {
                        temp=a[i];
                        a[i]=a[j];
                        a[j]=temp;
                }
                else
                {
                        temp=a[j];
                        a[j]=a[low];
                        a[low]=temp;
                        return j;
                }
        }
}
```

OUTPUT:

```
Enter the value of n:5
Enter the array elements1 6 7 2 4

The array elements before
1       6       7       2       4
Elements of the array after sorting are:
1       2       4       6       7
Time taken: 0.000000
```

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.

<u>Merge Sort Algorithm:</u>

Merge sort is a perfect example of a successful application of the divide-and-conquer technique.

1. Split array A[1..*n*] in two and make copies of each half in arrays B[1.. *n*/2] and C[1.. *n*/2]
2. Sort arrays B and C
3. Merge sorted arrays B and C into array A as follows:
   a) Repeat the following until no elements remain in one of the arrays:
      i. compare the first elements in the remaining unprocessed portions of the arrays
      ii. copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
   b) Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Algorithm:

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
        merge(A, p, q, r)
```

Complexity:

- All cases have same efficiency: $\Theta(n \log n)$
- Number of comparisons is close to theoretical minimum for comparison-based sorting: $\log n! \approx n \lg n - 1.44n$
- Space requirement: $\Theta(n)$ (<u>NOT</u> in-place)

Program:

```
#include <stdio.h>
#include<time.h>


// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {

  // Create L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  // Maintain current index of sub-arrays and main array
  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  // Until we reach either end of either L or M, pick larger among
  // elements L and M and place them in the correct position at A[p..r]
  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k]  =  M[j];
    j++;
    k++;
  }
}
```

```
// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r) {
  if (l < r) {

    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}

// Print the array
void printArray(int arr[], int size) {
  for (int i = 0; i < size; i++)
  printf("%d ", arr[i]);
  printf("\n");
}

// Driver program
int main() {
  int arr[] = {6, 5, 12, 10, 9, 1};
  clock_t s,e;

  int size = sizeof(arr) / sizeof(arr[0]);
s=clock();

  mergeSort(arr, 0, size - 1);
e=clock();
  printf("Sorted array: \n");
  printArray(arr, size);
  printf("\nTime taken:%f",(e-s)/CLOCKS_PER_SEC);
}
```

OUTPUT

```
Sorted array:
1 5 6 9 10 12

Time taken:0.000000
```

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

N Queen's problem:

The *n*-queens problem consists of placing *n* queens on an *n* x *n* checker board in such a way that they do not threaten each other, according to the rules of the game of chess. Every queen on a checker square can reach the other squares that are located on the same horizontal, vertical, and diagonal line. So there can be at most one queen at each horizontal line, at most one queen at each vertical line, and at most one queen at each of the 4*n*-2 diagonal lines. Furthermore, since we want to place as many queens as possible, namely exactly *n* queens, there must be exactly one queen at each horizontal line and at each vertical line. The concept behind backtracking algorithm which is used to solve this problem is to successively place the queens in columns. When it is impossible to place a queen in a column (it is on the same diagonal, row, or column as another token), the algorithm backtracks and adjusts a preceding queen

```
Algorithm NQueens (k, n)
//Using backtracking, this procedure prints all possible placements of n queens
//on an n x n chessboard so that they are non-attacking
{
        for i ← 1 to n do
        {
              if(Place(k,i) )
              {
                    x[k] ← i
                    if (k=n)
                         write ( x[1...n])
                    else
                         Nqueens (k+1, n)
              }
        }
}

Algorithm Place( k, i)
//Returns true if a queen can be placed in kth row and ith column. Otherwise it
//returns false. x[] is a global array whose first (k-1) values have been set. Abs(r)
//returns the absolute value of r.
{
      for j ← 1 to k-1 do
      {
            if (  (x[j]=i or Abs(x[j]-i) = Abs(j-k) )
            {
                  return false
            }
      }
}
```

Complexity:

The power of the set of all possible solutions of the n queen’s problem is n! and the bounding function takes a linear amount of time to calculate, therefore the running time of the n queens problem is O (n!).

Program:

```
#include<stdio.h>

void nqueens(int);
int place(int[],int);
void printsolution(int,int[]);

void main()
{
        int n;
        clrscr();
        printf("Enter the no.of queens: ");
        scanf("%d",&n);
        nqueens(n);
        getch();
}

void nqueens(int n)
{
        int x[10],count=0,k=1;
        x[k]=0;

        while(k!=0)
        {
                x[k]=x[k]+1;

                while(x[k]<=n&&(!place(x,k)))
                        x[k]=x[k]+1;

                if(x[k]<=n)
                {
                        if(k==n)
                        {
                                count++;
                                printf("\nSolution %d\n",count);
                                printsolution(n,x);
                        }
                        else
                        {
                                k++;
                                x[k]=0;
                        }
                }
                else
                {

                        k--;  //backtracking
                }
        }
        return;
}
```

```
int place(int x[],int k)
{
        int i;

        for(i=1;i<k;i++)
                if(x[i]==x[k]||(abs(x[i]-x[k]))==abs(i-k))
                        return 0;
        return 1;
}

void printsolution(int n,int x[])
{
        int i,j;
        char c[10][10];

        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                        c[i][j]='X';

        }

        for(i=1;i<=n;i++)
                c[i][x[i]]='Q';

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

OUTPUT:

```
Enter the no.of queens: 4

Solution 1
X       Q       X       X
X       X       X       Q
Q       X       X       X
X       X       Q       X

Solution 2
X       X       Q       X
Q       X       X       X
X       X       X       Q
X       Q       X       X
```