SIR M. VISVESVARAYA INSTITUTE OF TECHNOLOGY

BENGALURU-562157

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

LAB MANUAL

2023-2024

ANALYSIS & DESIGN OF ALGORITHMS LABORATORY

[AS PER OUTCOME BASED EDUCATION (OBE) AND CHOICE BASED CREDIT SYSTEM (CBCS) 2022 SCHEME]

BCSL404

IV SEMESTER CSE

PREPARED AND COMPILED BY: Dr. SUMA SWAMY

UNDER THE GUIDANCE OF

HOD

Dr. ANITHA T N

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

Suggested Learning Resources:

- Virtual Labs (CSE): http://cse01-iiith.vlabs.ac.in/

Programs

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

The greedy method is most straightforward design technique and can be applied to a wide variety of problems. These problems have n inputs and require us to obtain subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the given objective function. This feasible solution is called an optimal solution.

```
1   Algorithm Greedy(a, n)
2   // a[1 : n] contains the n inputs.
3   {
4       solution := ∅; // Initialize the solution.
5       for i := 1 to n do
6       {
7           x := Select(a);
8           if Feasible(solution, x) then
9               solution := Union(solution, x);
10      }
11      return solution;
12  }
```

Kruskal's Algorithm using Greedy method

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

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

1. Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

3. Repeat step2 until there are (V-1) edges in the spanning tree.

The step2 uses Union-Find algorithm to detect cycle.

A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset.

In this post, we will discuss an application of Disjoint Set Data Structure. The application is to check whether a given graph contains a cycle or not.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. This method is based on Union-Find. This method assumes that graph doesn't contain any self-loops.

Pseudocode: 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)<=…<=w(e_i|E|)
 E_T←∅  ; ecounter ← 0 //initialise the set of tree edges and its size
 k ← 0                 //initialise the number of processed edges
 while ecounter < |V| - 1
 k←k+1
 if E_T ∪ {e_ik}is acyclic
 E_T←E_T ∪ {e_ik}; ecounter ← ecounter+1
 Return E_T
```

Using Union Find Algorithm:

1. KRUSKAL(G):
2. A = ∅
3. For each vertex v ∈ G.V:
4. MAKE-SET(v)
5. For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
6. if FIND-SET(u) ≠ FIND-SET(v):
7. A = A ∪ {(u, v)}
8. UNION(u, v)

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

    void union1(int i,int j,int s[ ])
    {
      s[i]=j;
    }

int find(int v,int s[])

        {

                while(s[v]!=v)

                v=s[v];

        return v;

        }


    void kruskal(int n,int c[10][10])
    {
     int count,i,s[10],min,j,u,v,k,t[10][2],sum;
      for(i=0;i<n;i++)
      s[i]=i;
      count=0;
      sum=0;
      k=0;
      while(count<n-1)
    {
           min=999;
    for(i=0;i<n;i++)
    {
       for(j=0;j<n;j++)
    {
          if(c[i][j]!=0  &&  c[i][j]<min)
    {
    min=c[i][j];
      u=i,v=j;
    }
    }
    }
      if(min==999)
      break;
      i=find(u,s);
      j=find(v,s);
      if(i!=j)
    {
```

```
t[k][0]=u;
    t[k][1]=v;
    k++;
    count++;
        sum=sum+min;
        union1(i,j,s);
    }
     c[u][v]=c[v][u]=999;
    }
if(count==n-1)
{
    printf("cost of spanning tree =%d\n",sum);
    printf("spanning tree is shown below\n");
    for(k=0;k<n-1;k++)
    {
    printf("%d -> %d = %d\n",t[k][0],t[k][1], c[u][v]);
    }
    return();
}
    printf("spanning tree doesn't exist\n");
}

void main()
{
    clock_t  st,end;
    int n,c[10][10],i,j;
    printf("enter no. 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",&c[i][j]);
    }
    }
    st=clock();
    kruskal(n,c);
    end=clock();
     printf("\ntime taken in sec is %d\n",(end-st));
}
```

OUTPUT

Enter the number of nodes
4
Enter the cost adjacency matrix
999 3 5 4
3 999 999 7
5 999 999 6
4 7 6 999

Cost of spanning tree = 12
Spanning tree is shown below
0→1
0→3
0→2
time taken in sec is

Analysis of Kruskal's Algorithm

If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in $O(|E| \log |V|)$. This is because the algorithm performs $|V| - 1$ deletions of the smallest element and makes $|E|$ verifications and, possibly, changes of an element's priority in a min-heap of size not exceeding $|V|$. Each of these operations, as noted earlier, is a $O(\log |V|)$ operation.

With an efficient union-find algorithm, the running time of Kruskal's algorithm will be dominated by the time needed for sorting the edge weights of a given graph. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in $O(|E| \log |E|)$.

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 using greedy Method

Prim's algorithm constructs a minimum spanning tree through a sequenceof expanding subtrees. 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, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.) The algorithm stops after all the graph's vertices have been included in the tree being constructed. Since the algorithm expands a treeby exactly one vertex on each of its iterations, the total number of such iterations is $n - 1$, where n is the number of vertices in the graph. The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.

Pseudocode:*Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph $G = \{V, E\}$

//Output: Et, the set of edges composing a minimum spanning tree of G

$Vt \leftarrow \{v0\}$ //the set of tree vertices can be initialized with any vertex

$Et \leftarrow \emptyset$

for $i \leftarrow 1$ **to** $|V| - 1$ **do**

find a minimum-weight edge $e* = (v*, u*)$ among all the edges (v, u)

such that v is in $Vt$and u is in $V - Vt$

$Vt \leftarrow Vt \cup \{u*\}$

$Et \leftarrow Et \cup \{e*\}$

return Et

```
#include<stdio.h>
#include<conio.h>
#include<time.h>
int n,v,s[20],d[20],p[20],w[20][20],t[20][20];
int findmin()
{
  int i,min;
  for(i=1;i<=n;i++)
  {
   if(s[i]==0)
   {
        min=1;
        break;
   }
}
for(i=1;i<=n;i++)
{
  if(d[i]<d[min]&&s[i]==0)
  min=i;
}
return min;
}

void prims()
{
 int k,u;
 for(int i=1;i<=n;i++)
{
s[i]=0;
d[i]=999;
 p[i]=0;
}
 s[v]=1;
 int x=0,sum=0;
for(k=1;k<=n;k++)
 {
   u=findmin();
   t[x][0]=u;
   t[x][1]=p[u];
   x++;
sum+=w[u][p[u]];
   s[u]=1;
   for(int w1=1;w1<=n;w1++)
   {
     if(w[u][w1]!=999   &&   s[w1]==0)
     {
          if(d[w1]>w[u][w1])
          {
```

```
                d[w1]=w[u][w1];
                p[w1]=u;
              }
            }
      }
    }
    if(sum>999)
printf("spanning tree doesn,t exist\n");
else
{
          printf("mincost spanning tree is\n");
          for(i=1;i<n;i++)
          {
            printf("%d--->%d=%d\n",t[i][0],t[i][1], w[u][v]);
          }
          printf("spanning tree cost is %d\n",sum);
}
}
void main()
{
            int i,j;
            clock_t st,end;
            clrscr();
            printf("enter the number of nodes\n");
            scanf("%d",&n);
            printf("Enter the cost of adjacency matrix\n");
            for(i=1;i<=n;i++)
            for(j=1;j<=n;j++)
            scanf("%d",&w[i][j]);
            st=clock();
            prims();
            end=clock();
            printf("\ntime taken in sec is %d\n",(end-st)/CLK_TCK);
            getch();
}
```

OUTPUT

```
Enter the no. of nodes:
4
Enter the cost of adjacency matrix
999     3       5       4
3       999     999     7
5       999     996     6
4       7       6       999
```

mincost spanning tree is
1→2=3
1→4=4
1→3=5
spanning tree cost is = 12

time taken in sec is

Analysis of Prims Algorithm

If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in $O(|E| \log |V|)$. This is because the algorithm performs $|V| - 1$ deletions of the smallest element and makes $|E|$ verifications and, possibly, changes of an element's priority in a min-heap of size not exceeding $|V|$. Each of these operations, as noted earlier, is a $O(\log |V|)$ operation. Hence, the running time of this implementation of Prim's algorithm is in

$$(|V| - 1 + |E|)O(\log |V|) = O(|E| \log |V|)$$

because, in a connected graph, $|V| - 1 \leq |E|$.

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

Dynamic Programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Floyd's Algorithm for All-Pairs Shortest-Paths Problem using Dynamic Programming

Given a weighted connected graph (undirected or directed), the ***all-pairs shortest paths problem*** asks to find the distances—i.e., the lengths of the shortest paths—from each vertex to all other vertices. This is one of several variations of the problem involving shortest paths in graphs. Because of its important applications to communications, transportation networks, and operations research, it has been thoroughly studied over the years. Among recent applications of the all-pairs shortest-path problem is precomputing distances for motion planning in computer games.

It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***: the element dij in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex. We can generate the distance matrix with an algorithm that is very similar toWarshall's algorithm. It is called ***Floyd's algorithm*** after its co-inventor Robert W. Floyd. It is applicable to both undirected and directed weighted graphs provided that they do not contain a cycle of a negative length. (The distance between any two vertices in such a cycle can be made arbitrarily small by repeating the cycle enough times.) The algorithm can be enhanced to find not only the lengths of the shortest paths for all vertex pairs but also the shortest paths themselves.

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:

$D(0), \ldots, D(k-1), D(k), \ldots, D(n)$.

Pseudocode: *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

$D \leftarrow W$ //is not necessary if W can be overwritten

for $k \leftarrow 1$ **to** n **do**

for $i \leftarrow 1$ **to** n **do**

for $j \leftarrow 1$ **to** n **do**

$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$

return D

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
#define infinity 999
int min(int a,int b)
{
  return a<b?a:b;
}
void floyd1(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]=min(p[i][j].p[i][k]+p[k][j]);
 }
}
}
}
void main()
{
  int a[4][4]={{0,999,3,999},{2,0,999,999},{999,7,0,1},{6,999,999,0}}
  int i,j,n;
  clock_t start,finish;
  int counter;
  float sec;
  printf("\nEnter the no. of nodes:");
  scanf("%d",&n);
  printf("\nEnter the cost matrix 0 – for self loop and 999 – for no edges\n");

for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
scanf("%d",&a[i][j]);
  start=clock();
 counter=0;
  while(clock()-start<100)
{
counter++;
floyd1(a,n);
}
finish=clock();
sec=((finish – start)/18.26)/counter;
printf("Time taken: %f",sec);
printf("\nThe resultant path matrix is\n");
```

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

OUTPUT

```
Enter the number of vertices
4
Enter the cost matrix 0 – for self loop and 999 – for no edges
0      999  3  999
2 0  999  999
999  7  0  1
6  999  999  0
Time Taken: 68.455643
The resultant path matrix
0  10  3   4
2  0    5   6
7  7    0   1
6  16  9   0
```

Analysis of Floyd's Algorithm

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n}$$

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} n - 1 + 1$$

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} n$$

$$C(n) = \sum_{k=1}^{n} n \sum_{i=1}^{n} 1$$

$$C(n) = \sum_{k=1}^{n} n\ (n - 1 + 1)$$

$$C(n) = \sum_{k=1}^{n} n^2$$

$$C(n) = n^2 \sum_{k=1}^{n} 1$$

$$C(n) = n^2 \ (n - 1 + 1)$$

$$C(n) = n^3$$

the time efficiency of Floyd's algorithm is cubic

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

Warshal's algorithm using dynamic programming:

The ***transitive closure*** of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row and the j th column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the j th vertex; otherwise, t_{ij} is 0.

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 $k \leftarrow 1$ **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 $R^{(n)}$

```
#include<stdio.h>
#include<conio.h>
#include<time.h>
int n,i,j,k, a[50][50],p[50][50];
void warshall(int n,int a[50][50],int p[50][50])
{
for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
 p[i][j]=a[i][j];
for(k=1;k<=n;k++)
for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
 if(p[i][k]==1  &&  p[k][j]==1)
  p[i][j]=1;
}
void input(int n,int a[50][50])
{
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
scanf("%d",&a[i][j]);
}
void output(int n,int p[50][50])
 {
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
printf("%d\t",p[i][j]);
printf("\n");
}
}
void main()
{
int i,j;
clock_t st,end;
clrscr();
printf("enter the number of nodes:\n");
scanf("%d",&n);
printf("enter the adjacency matrix:\n");
input(n,a);
st=clock();
warshall(n,a,p);
end=clock();
printf("the transitive closure is:\n");
output(n,p);
printf("the time taken by warshall's algorithm is %d sec\n",
  (end - st)/CLK_TCK);
getch();
}
```

OUTPUT

Enter the number of nodes:
4

Enter the adjacency matrix:
0 1 0 0
0 0 0 1
0 0 0 0
1 0 1 0

The transitive closure is:
1 1 1 1
1 1 1 1
0 0 0 0
1 1 1 1

The time taken by Warshall's Algorithm is 0 sec

Analysis of Warshall's Algorithm

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n}$$

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} n - 1 + 1$$

$$C(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} n$$

$$C(n) = \sum_{k=1}^{n} n \sum_{i=1}^{n} 1$$

$$C(n) = \sum_{k=1}^{n} n \ (n - 1 + 1)$$

$$C(n) = \sum_{k=1}^{n} n^2$$

$$C(n) = n^2 \sum_{k=1}^{n} 1$$

$$C(n) = n^2 \ (n - 1 + 1)$$

$$C(n) = n^3$$

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 using Greedy Method :(Dijkstra's algorithm)

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.

Pseudocode:

```
findmin( )
{
        for i ← to n   do
                if (s[i] = 0) do
                {
                        min ← i
                        break
                }
                for i ← 1 to n do
                {
                        if (d[i]<d[min] & s[i]=0)
                        min ← i
                }
        return min
}
dijkstra( )
{
        for i ← to n   do
        {
                s[i] ← 0
                d[i] ← 999
                p[i] ← 0
        }
        d[v] ←  0
        for k ← 1 to n do
        {
                u ← findmin( )
                s[u] ← 1
                for  w1 ←1 to n do
                {
                        if (w[u][w1]!=999 & s[w1] = 0)
                        {
                                if (d[w1]>d[u]+w[u][w1] )
```

```
                                {
                                        d[w1] ← d[u]+w[u][w1]
                                        p[w1] ←u
                                }
                        }
                }
        }
        display "shortest path costs"
        for i ←1 to n do
        {
                if (d[i]=999)
                        display "sorry!no path for source v to vertex i"
                else
                        display “path cost from v to i is d[i]”
        }
        display "shortest group of paths are"
        for i ←1 to n do
        {
                if i!=v & d[i]!=999
                {
                        display i
                        j ←p[i]
                        while p[j]!=0 do
                        {
                                print "→" j
                                j ← p[j];
                        }
                        print "→"v
                }
        }
}
main( )
{
        accept number of vertices n
        accept weight matrix 999 for ∞
        accept source vertex
        call dijkstra( )
}
```

```
#include<stdio.h>
#include<conio.h>
#include<time.h>
int d[20],s[20],w[20][20],p[20],n,v;
int findmin()
{
    int i,min;
    for(i=1;i<=n;i++)
        if(s[i]==0)
        {
                min=i;
               break;
        }
for(i=1;i<=n;i++)
{
          if(d[i]<d[min] && s[i]==0)
          min=i;
}
     return min;
}
void dijikstra()
{
    int i,w1,u,k,j;
    for(i=1;i<=n;i++)
{
        s[i]=0;
        d[i]=999;
        p[i]=0;
 }
    d[v]=0;
    for(k=1;k<=n;k++)
{
u=findmin();
         s[u]=1;
           for(w1=1;w1<=n;w1++)
        {
                if(w[u][w1]!=999 && s[w1]==0)
         {
                 if(d[w1]>d[u]+w[u][w1])
                 {
                    d[w1]=d[u]+w[u][w1];
                    p[w1]=u;
                  }
        }
          }
        }
        printf("shortest path costs\n");
        for(i=1;i<=n;i++)
```

```
        {
                if(d[i]==999)
                    printf("sorry!  no path for source %d to %d vertex",v,i);
                 else
                    printf("path cost from %d to %d is %d\n",v,i,d[i]);
         }
         printf("shortest group of paths are\n");
         for(i=1;i<=n;i++)
         {
            if(i!=v && d[i]!=999)
            {
                    printf("%d",i);
                    j=p[i];
                    while(p[j]!=0)
                    {
                    printf("-->%d",j);
                    j=p[j];
                    }
                    printf("-->%d\n",v);
          }
         }
}
  void main()
  {
    int i,j;
    clock_t end,st;
    clrscr();
    printf("enter the number of vertices\n");
    scanf("%d",&n);
    printf("enter the cost of vertices\n");
          for(i=1;i<=n;i++)
                  for(j=1;j<=n;j++)
                          scanf("%d",&w[i][j]);
    printf("enter the source vertex\n");
    scanf("%d",&v);
    st=clock();
    dijkstra( );
   end=clock();
    printf("time taken is %d\n",(end-st)/CLK_TCK);
    getch();
  }
```

OUTPUT

```
Enter the number of vertices
 5
```

```
Enter the cost of vertices
0       3       999     7       999
999     0       4       2       999
999     4       0       5       6
7       2       5       0       4
999     999     6       4       0

Enter the source vertex
 1
shortest path costs
     path cost from  1  to  1  is 0
     path cost from  1  to  2  is 3
     path cost from  1  to  3  is 7
     path cost from  1  to  4  is 5
     path cost from  1  to  5  is 9


     shortest group of paths are
     2→1
   3→2→1
    4→2→1
   5→4→2→1

Time Taken is 0
```

Analysis of Dijkstra's Algorithm

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself. For the reasons explained in the analysis of Prim's algorithm in Section 9.1, it is in $(|V|^2)$ for graphs represented by their weight matrix and the priority queue implemented as an unordered array. For graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is in $O(|E| \log |V|)$.

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

The ***decrease-and-conquer*** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance. Once such a relationship is established, it can be exploited either top down or bottom up. The former leads naturally to a recursive implementation, although, as one can see from several examples in this chapter, an ultimate implementation may well be nonrecursive. The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the ***incremental approach***.

There are three major variations of decrease-and-conquer:

1. decrease by a constant
2. decrease by a constant factor
3. variable size decrease

In the ***decrease-by-a-constant*** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 5.1), although other constant size reductions do happen occasionally.

source-removal algorithm for the topological sorting: It is based on a direct implementation of the decrease-(by one)-and-conquer technique: repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved) The order in which the vertices are deleted yields a solution to the topological sorting problem.

```
#include<stdio.h>
#include<conio.h>
int indeg[10],flag[10],n,a[10][10];
void topology();
void main()
{
  int i,j;
  clrscr();
  printf("enter the number of vertices:");
  scanf("%d",&n);
  printf("enter the adjacency matrix:");
  for(i=1;i<=n;i++)
        for(j=1;j<n;j++)
        scanf("%d",&a[i][j]);
  printf("\nthe topolgical ordering of the vertices is:");
  topology();
  getch();
}
void topology()
{
 int i,j,cnt=0,m=0;
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
indeg[i]+=a[j][i];
while(cnt<n)
{
  m++;
 if(m==n)
{
        printf("\ntopological ordering is not possible");
        printf("\nthe given graphic is cyclic");
        getch();
        exit(0);
        }
        for(i=1;i<n;i++)
        {
         if(indeg[i]==0&&flag[i]==0)
        {
            printf("\n%d",i);
            cnt++;
            flag[i]=1;
            for(j=1;j<=n;j++)
              if(a[i][j]==1)
                    indeg[j]--;
        }
 }
}
 }
```

OUPUT

Enter the number of vertices: 4

Enter the adjacency matrix:

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

The topological ordering of the vertices is:
1
2
3
4

Analysis of Topological Sorting

Using DFS method it is O(V+E)

Using Source Removal Method:

For input graph G = (V,E), Run Time = ? Break down into total time required to:

- Initialize In-Degree array: O(|E|)
- Find vertex with in-degree 0: |V| vertices, each takes O(|V|) to search In-Degree array. Total time = O(|V| 2)
- Reduce In-Degree of all vertices adjacent to a vertex: O(|E|)
- Output and mark vertex: O(|V|) Total time = O(|V| ^{2} + |E|) Quadratic time!

As this in nonrecursive algorithm,

$$C(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

$$C(n) = \sum_{n} (n - 1 + 1)$$

$$\sum_{i=1}^{n}$$

$$C(n) = n\sum_{i=1}^{n} 1$$

$$C(n) = n(n - 1 + 1)$$

$$C(n) = n^2$$

$$C(n) \in \Theta(n^2)$$

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

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

Optimal Substructure:

To consider all subsets of items, there can be two cases for every item:

(1) the item is included in the optimal subset, (2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of following two values.

1) Maximum value obtained by n-1 items and W weight (excluding nth item).

2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

0/1(Discrete) Knapsack Pseudocode:

```
max(a,b)
{
return(a>b)?a:b;
}

 knap(i, j)
 {
        if(i=0 or j=0) then
                        v[i][j]=0
                elseif(j<w[i]) then
                        v[i][j]=knap(i-1,j)
```

```
            else
                v[i][j]=max(knap(i-1,j), value[i]+knap(i-1,j-w[i]))
            return v[i][j]
 }
optimal( i,j)
 {
            if(i>=1 or j>=1) then
                if(v[i][j]!=v[i-1][j]) then
                {
                     Print Item i
                    b[i]=1
                    j=j-w[i]
                    optimal(i-1,j)
                }
                else
                optimal(i-1,j);
 }
```

```
#include<iostream.h>
#include<conio.h>
int w[10], v[10][10],value[10],b[10];
int max(int a,int b)
{
 return(a>b)?a:b;
}
 int knap(int i,int j)
 {
  if(i==0 || j==0)
     v[i][j]=0;
  else if(j<w[i])
      v[i][j]=knap(i-1,j);
  else
      v[i][j]=max(knap(i-1,j), value[i]+knap(i-1,j-w[i]));
   return v[i][j];
  }
   void optimal(int i,int j)
   {
 if(i>=1 || j>=1)
       if(v[i][j]!=v[i-1][j])
       {
           cout<<"item: "<<i<<endl;
           b[i]=1;
           j=j-w[i];
           optimal(i-1,j);
           }
else
            optimal(i-1,j);
  }
 void main()
{
          int profit, w1,n,i,j;
          clrscr();
          cout<<"enter the number of items:";
          cin>>n;
          cout<<"enter the capacity of the knapsack:";
          cin>>w1;
          cout<<"enter the values:";
          for(i=1;i<=n;i++)
          cin>>value[i];
          cout<<"enter the weights:";
          for(i=1;i<=n;i++)
          cin>>w[i];
          profit=knap(n,w1);
          cout<<"profit: "<<profit<<"\noptimal subset is:\n";
          optimal(n,w1);
          cout<<"the solution vector is:";
```

```
        for(i=1;i<=n;i++)
        cout<<b[i];
         getch();
  }
```

OUTPUT

```
Enter the number of items: 4
Enter the capacity of knapsack: 5
Enter the values:
12
10
20
15
Enter the weights:
2
1
3
2
profit: 37
Optimal Subset is:
item: 4
item:  2
item:  1
The solution Vector is:
              0    1 0 1
```

Analysis of 0/1 Knapsack Problem

Time efficiency and space efficiency of this algorithm are both in *(nW).* The time needed to find the composition of an optimal solution is in *O(n).*

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

Fractional (Continuous) Knapsack using Greedy Method

Algorithm:

- Assume knapsack holds weight W and items have value v_i and weight w_i
- Rank items by value/weight ratio: v_i / w_i
- Thus: $v_i / w_i \geq v_{i+1} / w_{i+1}$
- Consider items in order of decreasing ratio
- Take as much of each item as possible

Pseudocode: GreedyKnapsack(m,n)

```
//p[1:n] and w[1:n] – contains the profits and weights of n objects ordered such that
//p[i]/w[i] >= p[I+1]/w[i+1]
//m is size of knapsack and x[1:n] is the solution vector
{
        for i<- 1 to n do
            x[i]=0.0
            u <- m
           for i <- 1 to n do
            {
                    if (w[i] > u) then break
                    x[i] <- 1.0
                    u <- u-w[i]
            }
       if (i<=n) then
       x[i]=u / w[i]
}
```

```
#include<iostream.h>
#include<conio.h>

 void knapsack(int n, int item[],float weight[], float profit[], float capacity)
{
                float tp = 0,u;
                int i;
                u =  capacity;
                float x[20];
                for (i = 0; i < n; i++)
                x[i] = (float) 0.0;
                for (i = 0; i < n; i++)
                 {
                    if (weight[i] > u)
                      break;
                    else {
                                x[i] = (float) 1.0;
                                tp = tp + profit[i];
                                u = (int) (u - weight[i]);
                              }
                  }

                  if (i < n)
                     x[i] = u / weight[i];
                  tp = tp + (x[i] * profit[i]);
                cout<<"\nThe result vector is:- ";
                  for (i = 0; i < n; i++)
                cout<<"\tItem "+item[i]+":" +x[i];
                  cout<<"\nMaximum profit is:- " +tp;
}

void main( )
{
```

```
  float weight[20];
  float profit[20];
  float capacity;
  int num, i, j;
  float ratio[20], temp;
  int item[10];

  cout<<"\nEnter the no. of objects:- ";
  cin>>num;
  cout<<"\nEnter the  items, weights and profits of each object:- ";
  for (i = 0; i < num; i++)
 {
      cin>>item[i];
     cin>>weight[i];
     cin>>profit[i];
  }
cout<<"\nEnter the capacity of knapsack:- ";
  cin>>capacity;
  for (i = 0; i < num; i++)
  {
    ratio[i] = profit[i] / weight[i];
  }
  for (i = 0; i < num; i++)
 {
     for (j = i + 1; j < num; 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;

            temp=item[j];
            item[j]=item[i];
            item[i]=(int)temp;
          }
        }
    }
    knapsack(num, item,weight, profit, capacity);
}
}
```

Output:

```
Enter the no. of objects:-
3

Enter the items, wts and profits of each object:-
2 15 24
3 10 15
1 18 25

Enter the capacity of knapsack:-
20

The result vector is:-
        Item 2:1.0
        Item 3:0.5
        Item 1:0.0

Maximum profit is:- 31.5
```

Analysis of Knapsack problem

As this algorithm is nonrecursive,

$$C(n) = \sum_{i=1}^{n} 1$$

$$C(n) = n - 1 + 1$$

$$C(n) = n$$

$$C(n) \in O(n)$$

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.

Backtracking: The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for *any* remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

We consider the ***subset-sum problem***: find a subset of a given set $A = \{a_1, \ldots, a_n\}$ of n positive integers whose sum is equal to a given positive integer d. For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: {1,2,6} and {1,8}.Of course, some instances of this problem may have no solutions.

The state-space tree can be constructed as a binary tree like that in Figure 11.9 for the instance $A = \{3, 5, 6, 7\}$ and $d = 15$. The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left and right children represent, respectively, inclusion and exclusion of a_1 in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of a_2 while going to the right corresponds to its exclusion, and so on. Thus, a path from the root to a node on the ith level of the tree indicates which of the first i numbers have been included in the subsets represented by that node.

We record the value of s, the sum of these numbers, in the node. If s is equal to d, we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent. If s is not equal to d, we can terminate the node as nonpromising.

Pseudocode:

```
sumofsub(0,1,sum)
{
        x[k]=1
        if(s +w[k] <- u)
        {
                print solution v++
                for i<- 1 to n do
```

```
                    if x[i]=1
                        print w[i]
        }
        else if (s+ w[k] +w[k+] < = m) do
            call sumofsub(s + w[k],k+1 , r-w[k])
        if( s +r-w[k]>=m and s+w[k+] <=m) do
        {
            x[k]<-0
            call sumofsub(s,k+1,r-w[k])
        }
}
```

```
#include<stdio.h>
#include<conio.h>
#include<time.h>
int w[20],x[10],n,m;
void sumofsub(int s,int k,int r)
{
int i;
x[k]=1;
if(s+w[k]==m)
{
printf("soln vector is");
for(i=1;i<=n;i++)
{
        printf("%d\t",x[i]);
}
 }
else
{
        if(s+w[k]+w[k+1]<=m)
sumofsub(s+w[k],k+1,r-w[k]);
        if(s+r-w[k]>=m&&s+w[k+1]<=m)
        {
                x[k]=0;
                sumofsub(s,k+1,r-w[k]);

        }
        }
 }
void main()
{
         int i,sum=0;
         clock_t st,end;
         clrscr();
         printf("Enter the number of elements\n");
         scanf("%d",&n);
         printf("enter the set\n");
        for(i=1;i<=n;i++)
        {
                scanf("%d",&w[i]);
                sum=sum+w[i];
                x[i]=0;
         }
         printf("enter the max subset value\n");
         scanf("%d",&m);
         printf("soln vector is\t");
         st=clock();
         sumofsub(0,1,sum);
         end=clock();
```

```
        printf("time taken is %d\n",(end-st)/CLK_TCK);
        getch();
}
```

OUTPUT

```
Enter the number of element
5
Enter the set
1       2       5       6       8
Enter the max subset value
9
Soln vector is
0 1  0  1  0
0 0  0  0  1

Time taken is 0
```

Analysis of Sum of subset problem:

Efficiency of the sum of subset problem is O(n * m) where n is number of elements in the set and m is the sum.

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.

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

The "force" implied by the strategy's definition is that of a computer and not that of one's intellect. "Just do it!" would be another way to describe the prescription of the brute-force approach. And often, the brute-force strategy is indeed the one that is easiest to apply.

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last n − 1 elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the ith pass through the list, which we number from 0 to $n-2$, the algorithm searches for the smallest item among the last $n-i$ elements and swaps it with A_i:

$$\underbrace{A_0 \le A_1 \le \cdots \le A_{i-1}}_{\text{in their final positions}} \;\big|\; \underbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

After n−1 passes, the list is sorted.

Working of Selection Sort in C

In each pass, the minimum element is identified from the unsorted part of the array and placed at the beginning of the sorted part of the array. The sorted portion of the array gradually grows with each pass until the entire array becomes sorted.

ALGORITHM *SelectionSort*$(A[0..n-1])$
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
$min \leftarrow i$
for $j \leftarrow i+1$ **to** $n-1$ **do**
if $A[j] < A[min]$ $min \leftarrow j$
swap $A[i]$ and $A[min]$

Selection Sort Algorithm in C

```
selectionSort(array, size)
  loop i from 0 to size - 2
    set minIndex as i
    loop j from first unsorted to size - 1
      check if array[j] < array[minIndex]
        set minIndex as j
    swap array[i] with array[minIndex]
  end for
end selectionSort
```

Selection Sort Program in C

```
// C program for implementation of selection sort

#include <stdio.h>



void swap(int* xp, int* yp)

{
```

```
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n - 1; i++) {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
```

```
        // Swap the found minimum element with the first
        // element
        swap(&arr[min_idx], &arr[i]);
    }
}

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

```
// Driver program to test above functions

int main()

{

    int arr[] = { 64, 25, 12, 22, 11 };

    int n = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, n);

    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}
```

Output

```
Sorted array:
11 12 22 25 64
```

a) **The elements generated using the random number generator**

```
#include <stdio.h>
#include<time.h>              //  for time()
#include<stdlib.h>            // for rand()

int  main()
{
        static int max=50000;
        clrscr();
        clock_t start,end;
        int a[MAX],n,i=0;
        printf("\n Enter the number of elements \n");
        scanf("%d", &n);
        printf("\n Enter the array :\n");
        for(i=0;i<n;i++)
        a[i]=rand()%1000; // storing a random number between 0 and 1000
        printf("\n The array is :\n");
        for(i=0;i<n;i++)
       printf("%d\n", a[i]);
        start=clock();
        selectionSort(arr, n);
        end=clock();
        printf("\n The sorted array is :\n");
        for(i=0;i<n;i++)
         printf("%d\n", a[i]);
        printf("\n Time taken is :", (end-start)/(double)CLOCKS_PER_SEC);
        return 0;
}


selectionSort(int[], nums)
{
for (int i = 0; i < nums-1; i++)
{
// min is the index of the smallest element with an index greater or equal to i
int swap, min = i;
for (int j = i + 1; j < n; j++)
{
if (nums[j] < nums[min])
{
min = j;
}
    if ( min != i )
```

```
    {
      swap = nums[i];
      nums[i] = nums[min];
      nums[min]= swap;
    }

}
```

Output:

Selectionsort Test
Enter the number of elements
10
the random elements are
17
31
81
44
91
24
87
42
57
17
The Elements after sorting
17
17
24
31
42
44
57
81
87
91
Time taken for execution is: _____ nanoseconds

Note: For n=5000 to 50000 in step of 5000 note down the time in nanoseconds. Convert time in seconds and plot the graph with n as x axis and time as y axis.

b) The elements read from a file

Create a text file in notepad and enter random numbers one in each line. Save the file as t.txt

```
selectionSort(int[], nums)
{
for (int i = 0; i < nums; i++)
```

```
{
// min is the index of the smallest element with an index greater or equal to i
int min = i;
for (int j = i + 1; j < nums; j++)
{
if (nums[j] < nums[min])
{
min = j;
}
}

int  main()
{
        clrscr();
         FILE *f1;
         char line[50];
         int limit;
         clock_t start,end;
         int a[MAX],n,i=0;
         cout<<"\n Enter the number of elements \n";
         cin>>n;
         f1=fopen("t.txt","r");
         while((i<=n-1)&&(fgets(line,50,f1)!=NULL))
         {
         sscanf(line,"%d",&limit);
         a[i]=limit;
         i++;
         }
         fclose(f1);
         cout<<"\n The array is : \n";
         for(i=0;i<n;i++)
         cout<<"  " <<a[i];
         start=clock();
         selectionSort(a,n);
         end=clock();
         cout<<"\n The sorted array is :\n";
         for(i=0;i<n;i++)
         cout<<"  "<<a[i];
         cout<<"\n Time taken is :"<<(end-start)/(double)CLOCKS_PER_SEC;
         getch();
}
```

Analysis of Selection Sort:

Time Complexity: $O(N^2)$, as there are two nested loops.

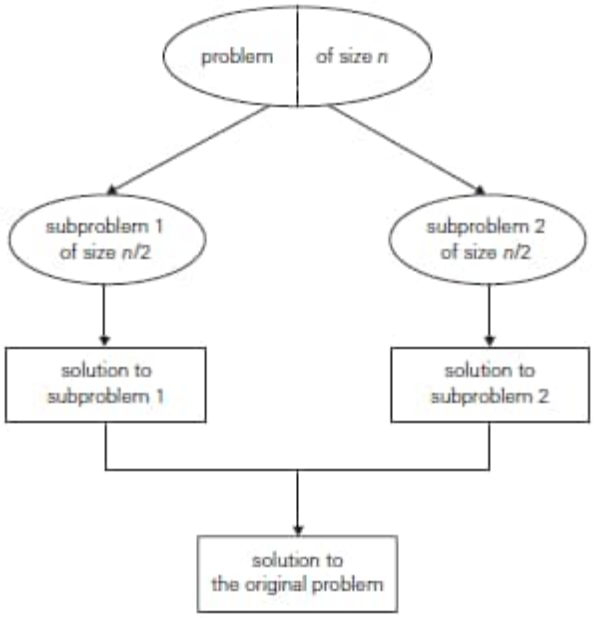
Auxiliary Space: O(1), as the only extra memory used is for temporary variable while swapping two values in Array.

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.

Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

The divide-and-conquer technique is diagrammed in Figure 5.1, which depicts the case of dividing a problem into two smaller subproblems, by far the most widely occurring case (at least for divide-and-conquer algorithms designed to be executed on a single-processor computer).



Quicksort is the important sorting algorithm that is based on the divide-and-conquer approach divides the input according to their value.. A partition is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition is achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of $A[s]$ independently (e.g., by the same method). Note the difference with mergesort: there, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

ALGORITHM *qs(A[l..r])*

//Sorts a subarray by quicksort

//Input: Subarray of array *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

qs(A[*l..s*−1]*)*

qs(A[*s*+1..*r*]*)*

ALGORITHM partit*ion(A*[*l..r*]*)*

//Partitions a subarray by Hoare's algorithm, using the first //element as a pivot

//Input: Subarray of array *A*[0..*n*− 1], defined by its left and right //indices *l* and *r (l* < *r)*

//Output: 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
```

[USING RANDOM FUNCTION]

```
#include <iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<time.h>
const int MAX=1000;

int partition(int a[],int low,int high)
{
        int i,j,temp,key;
        key=a[low];
        i=low;
        j=high+1;
        while(i<=j)
        {
                do
                        i++;
                while (key>=a[i]);
                do
                        j--;
                while(key<a[j]);
                if(i<j)
                {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
                }
        }
        temp=a[low];
        a[low]=a[j];
        a[j]=temp;
        return j;
}
void qs (int a[],int low, int high)
{
        int mid;
        if(low<high)
        {
                mid=partition(a,low,high);
                qs(a,low,mid-1);
                qs(a,mid+1,high);
        }
}
void main()
```

```
{
        clrscr();
        clock_t start,end;
        int a[MAX],n,i=0;
        cout<<"\n Enter the number of elements \n";
        cin>>n;
        cout<<"\n Enter the array :\n";
        for(i=0;i<n;i++)
        a[i]=rand();
        cout<<"\n The array is :\n";
        for(i=0;i<n;i++)
        cout<<"  "<<a[i];
        start=clock();
        qs(a,0,n-1);
        end=clock();
        cout<<"\n The sorted array is :\n";
        for(i=0;i<n;i++)
        cout<<"  " <<a[i];
        cout<<"\n Time taken is :"<< (end-start)/(double)CLOCKS_PER_SEC;
        getch();
}
```

Output:

```
Enter the number of elements
10
Enter the array :
The array is :
346  130  10982  1090  11656  7117  17595  6415  22948  31126
The sorted array is :
130  346  1090  6415  7117  10982  11656  17595  22948  31126
Time taken is :0.0
```

ANALYSIS

Best Case Efficiency: We start our discussion of quicksort's efficiency by noting that the number of key comparisons made before a partition is achieved is $n + 1$ if the scanning indices cross over and n if they coincide (why?). If all the splits happen in the middle of corresponding subarrays,

we will have the best case. The number of key comparisons in the best case satisfies the recurrence

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \qquad C_{best}(1) = 0.$$

According to the Master's Theorem, a = 2, b = 2, f(n) = n^d

Hence d =1 as f(n) = n and a = b^d as $2 = 2^1$

$C_{best}(n) \in \Theta(n^d \log_2 n)$

$C_{best}(n) \in \Theta(n \log_2 n)$

According to the Master Theorem, $C_{best}(n) \in (n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \text{ initial condition} \quad C_{best}(1) = 0.$$

Substituting $n = 2^k$

$$C_{best}(2^k) = 2C_{best}(2^{k-1}) + 2^k$$
$$= 2(2C_{best}(2^{k-2}) + 2^{k-1}) + 2^k$$
$$= 2^2 C_{best}(2^{k-2}) + 2.2^{k-1} + 2^k$$
$$= 2^2 C_{best}(2^{k-2}) + 2.2^k$$
$$= 2^2(2C_{best}(2^{k-3}) + 2^{k-2}) + 2.2^k$$
$$= 2^3 C_{best}(2^{k-3}) + 2^{2.}2^{k-2} + 2.2^k$$
$$= 2^3 C_{best}(2^{k-3}) + 2^k + 2.2^k$$
$$= 2^3 C_{best}(2^{k-3}) + 3.2^k$$

Generalizing the above statement by replacing 3 by i we get,

$$C_{best}(2^k) = 2^i C_{best}(2^{k-i}) + i.2^k$$

Substitute i = k to apply initial condition $C_{best}(1) = 0$

$$C_{best}(2^k) = 2^k C_{best}(2^{k-k}) + k.2^k$$
$$C_{best}(2^k) = 2^k C_{best}(2^0) + k.2^k$$
$$= 0 + k.2^k$$

Replacing $k = \log_2 n$ as $n = 2^k$

$$C_{best}(n) = n \log_2 n$$
$$C_{best}(n) \in \Omega(n \log_2 n)$$

Worst Case Efficiency: In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. It occurs when all the elements are arranged in ascending or

descending order. This unfortunate situation will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is already solved!.

After partitioning there are 2 cases:

1. When all the elements are arranged in ascending order no elements in the left**.** Hence after partitioning in worst case, n – 1 elements are towards the right. Indeed, if $A[0..n-1]$ is a strictly increasing array and we use $A[0]$ as the pivot, the left-to-right scan will stop on $A[1]$ while the right-to-left scan will go all the way to reach $A[0]$, indicating the split at position 0.

2. When all the elements are arranged in descending order no elements in the right**.** Hence after partitioning in worst case, n – 1 elements are towards the left.

For the first case, $C_{worst}(n) = 0$, if $n = 1$

$= C_{worst}(0) + C_{worst}(n-1) + n, n > 1$

So, after making $n+1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will be left with the strictly increasing array $A[1..n-1]$ to sort. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one $A[n-2..n-1]$ has been processed. The total number of key comparisons made will be equal to

$C_{worst}(n) = C_{worst}(0) + C_{worst}(n-1) + n$

$= C_{worst}(n-1) + n$

$= C_{worst}(n-2) + n -1+ n$

$= C_{worst}(n-2) + 2n – 1$

$= C_{worst}(n-3) + n-2+2n – 1$

$= C_{worst}(n-3) + 3n – 3$

Generalizing the above statement by replacing 3 by i we get,

$= C_{worst}(n-i) + n +(n – 1) + …..+(n – i + 1)$

Replacing i by n

$= C_{worst}(n-n) + n +(n – 1) + …..+(n – n + 1)$

$= C_{worst}(0) + n +(n – 1) + …..+1$

$= 0 + 1 + 2 + 3 + …….+n$

$= n(n + 1)/2$

$= n^2/2 + n/2$

$= n^2/2$

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

Average Case Efficiency: Thus, the question about the utility of quicksort comes down to its average-case behavior. Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s $(0 \le s \le n-1)$ after $n + 1$ comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and $n-1-s$ elements, respectively.

$C_{avg}(n)$ = time for partition + time to sort left sub array + time to sort right sub array.

Therefore, $C_{avg}(n) = (n + 1) + C_{avg}(s) + C_{avg}(n - 1 - s)$

Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$
$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) = 1/n \sum_{s=0}^{n-1} ((n + 1) + C_{avg}(s) + C_{avg}(n - 1 - s))$$

$$C_{avg}(n) = (1/n)\ (n + 1)\ \sum_{s=0}^{n-1} 1 + \sum_{s=0}^{n-1} C_{avg}(s) + \sum_{s=0}^{n-1} C_{avg}(n - 1 - s)$$

$$C_{avg}(n) = (1/n)\ (n + 1)\ (n) + \sum_{s=0}^{n-1} C_{avg}(s) + \sum_{s=0}^{n-1} C_{avg}(n - 1 - s)$$

$C_{avg}(n) = (1/n)\ (n + 1)\ (n) + C_{avg}(0) + C_{avg}(1) + \ldots C_{avg}(n - 1) + C_{avg}(n - 1)$

$+ C_{avg}(n-2) + \ldots . C_{avg}(0)$

$nC_{avg}(n) = n(n + 1) + 2(C_{avg}(0) + C_{avg}(1) + \ldots C_{avg}(n - 1))$ ----------eqn 1

Replace n by $n - 1$

$(n - 1)C_{avg}(n - 1) = (n- 1)(n) + 2(C_{avg}(0) + C_{avg}(1) + \ldots C_{avg}(n - 2))$----eqn 2

eqn 1 – eqn 2

$n\ C_{avg}(n) - (n-1)\ C_{avg}(n-1) = (\ n(n+1) + 2(C_{avg}(0) + C_{avg}(1) + \ldots C_{avg}(n-1))) -$

$(n-1)(n) + 2(C_{avg}(0) + C_{avg}(1) + \ldots C_{avg}(n-2))$

$nC_{avg}(n) - (n-1)C_{avg}(n-1) = n^2 + n - n2 + n + 2\ C_{avg}(n-1)$

$nC_{avg}(n) - n\ C_{avg}(n-1) + C_{avg}(n-1) = 2\ n + 2\ C_{avg}(n-1)$

$nC_{avg}(n) = 2\ n + 2\ C_{avg}(n-1) + n\ C_{avg}(n-1) - C_{avg}(n-1)$

$nC_{avg}(n) = 2\ n + C_{avg}(n-1)(2+n-1)$

$nC_{avg}(n) = 2\ n + C_{avg}(n-1)(\ n+1)$

divide by $n(n+1)$

$nC_{avg}(n) /n(n+1) = (2\ n + C_{avg}(n-1)(\ n+1)\ ) / n(n+1)$

$C_{avg}(n) / (n+1) = (2 /(n+1) + C_{avg}(n-1) / n$----------eqn 3

Replace n by n – 1 in eqn 3

$C_{avg}(n-1) / (n) = (2 / n + C_{avg}(n-2) / n - 1$ ----------eqn 4

Replace eqn 4 in eqn 3

$C_{avg}(n) / (n+1) = (2 /(n+1) + 2 / n + C_{avg}(n-2) / n - 1$-------- eqn 5

Replace n by n – 1 in eqn 4

$C_{avg}(n-2) / (n-1) = 2 / n-1+ C_{avg}(n - 3) / n – 2$-------- eqn 6

Replace eqn 6 in eqn 5

$C_{avg}(n) / (n + 1) = 2 /(n + 1)+ 2 / n+ 2 / n-1 + C_{avg}(n - 3) / n – 2$

Replace n by 4 in last term

$C_{avg}(n) / (n + 1) = 2 /(n + 1)+ 2 / n + 2 / n-1 + C_{avg}(1) / 2$

Cavg(1) = 0

$$C_{avg}(n) / (n + 1) = 2 \sum_{i=3}^{n+1} 1/ i$$

$$C_{avg}(n) = 2 (n + 1) \sum_{i=3}^{n+1} 1/ i$$

$$\int_{3}^{n+1} 1 / x \, dx = [\log_e x]_{3}^{n+1} = \log_e (n + 1) - \log_e 3$$

$$C_{avg}(n) = 2(n +1)[\log_e (n + 1) - \log_e 3]$$

$$= n \log n$$

$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$

[USING FILE]

```
#include <iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
#include<time.h>
const int MAX=1000;
int partition(int a[],int low ,int high)
{
        int i,j,temp,key;
        key=a[low];
        i=low;
        j=high+1;
        while(i<=j)
        {
                do
                        i++;
                while (key>=a[i]);
                do
                        j--;
                while(key<a[j]);
                if(i<j)
                {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
                }
        }
        temp=a[low];
        a[low]=a[j];
        a[j]=temp;
        return j;
}
void qs (int a[],int low, int high)
{
        int mid;
        if(low<high)
        {
                mid=partition(a,low,high);
                qs(a,low,mid-1);
                qs(a,mid+1,high);
        }
}
void main()
```

```
{
        clrscr();
        FILE *f1;
        char line[50];
        int limit;
        clock_t start,end;
        int a[MAX],n,i=0;
        cout<<"\n Enter the number of elements \n";
        cin>>n;
        f1=fopen("t.txt","r");
        while((i<=n-1)&&(fgets(line,50,f1)!=NULL))
        {
        sscanf(line,"%d",&limit);
        a[i]=limit;
        i++;
        }
        fclose(f1);
        cout<<"\n The array is : \n";
        for(i=0;i<n;i++)
        cout<<"  " <<a[i];
        start=clock();
        qs(a,0,n-1);
        end=clock();
        cout<<"\n The sorted array is :\n";
        for(i=0;i<n;i++)
        cout<<"  "<<a[i];
        cout<<"\n Time taken is :"<<(end-start)/(double)CLOCKS_PER_SEC;
        getch();
}
```

Output:

[In the file t.txt elements are]
10
9
8
44
3
57
100
98
54
32

Enter the number of elements
10

The array is :
10 9 8 44 3 57 100 98 54 32
The sorted array is :
3 8 9 10 32 44 54 57 98 100
Time taken is :0

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.

Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\ n/2-1]$ and $A[\ n/2\ ..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM *Mergesort(A[0..n−1])*
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\ n/2-1]$ to $B[0..\ n/2\ \ -1]$
copy $A[\ n/2\ ..n-1]$ to $C[0..\ n/2\ \ -1]$
Mergesort(B[0.. n/2 − 1])
Mergesort(C[0.. n/2 − 1])
Merge(B, C, A)

The ***merging*** of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

ALGORITHM *Merge(B[0..p−1], C[0..q−1], A[0..p+q−1])*
//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
if $B[i] \le C[j]$
$A[k] \leftarrow B[i]$; $i \leftarrow i+1$
else $A[k] \leftarrow C[j]$; $j \leftarrow j+1$
$k \leftarrow k+1$
if $i = p$
copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

[Using random generation]

```
#include<time.h>
#include<stdio.h>
#include<iostream.h>
#include<conio.h>
#define MAX 100
void mergesort(int[100],int,int);
void merge(int[100],int,int,int);
int a[max];
void main( )
{
 int i,n;
clock_t s,e;
printf("Enter the no. of elements\n");
scanf("%d",&n);
printf("Elements of the array before sorting\n");
for(i=0;i<n;i++)
{
a[i]=rand ( );
printf("%d\t",a[i]);
}
s=clock();
mergesort(a,0, n-1);
e=clock();
printf("\nElements of the array after sorting\n");
for(i=0;i<n;i++)
printf("%d\t".a[i]);
printf("the time taken=%f\n",(e-s)/CLOCKS_PER_SEC);
getch();
}
void mergesort(int a[100],int low,int high)
{
int mid;
       if(high>low)
{
mid=(low+high)/2;
mergesort(a,low,mid);
mergesort(a,mid+1,high);
 merge(a,low,mid,high);
}
}
void merge(int a[100],int low,int mid,int high)
{
```

```
int k=low,j=mid+1,i=low,c[max];
  while((i<=mid)&&(j<=high))
{
if(a[i]<=a[j])
{
c[k]=a[i];
i=i+1;
}
              else
{
c[k]=a[j];
 j=j+1;
}
           k=k+1;
}
if(k>mid)
while(i<=mid)
{
c[k]=a[i];
k=k+1;
i=i+1;
}
while(j<=high)
{
c[k]=a[j];
k=k+1;
j=j+1;
}
for(i=low,i<=high;i++)
a[i]=c[i];
}
```

OUTPUT

```
Enter the number of elements to be sorted
10
The elements of array before sorting:
83
86
77
15
93
35
86
92
```

```
49
21
Elements of array after sorting:
15
21
35
49
77
83
86
86
92
93

Time taken = 8.173808
```

[Using File]

```
void main()
{
        clrscr();
        FILE *f1;
        char line[50];
        int limit;
        clock_t start,end;
        int a[MAX],n,i=0;
        cout<<"\n Enter the number of elements \n";
        cin>>n;
        f1=fopen("t.txt","r");
        while((i<=n-1)&&(fgets(line,50,f1)!=NULL))
        {
        sscanf(line,"%d",&limit);
        a[i]=limit;
        i++;
        }
        fclose(f1);
        cout<<"\n The array is : \n";
        for(i=0;i<n;i++)
        cout<<"  " <<a[i];
        start=clock();
        mergesort(a,0,n-1);
        end=clock();
        cout<<"\n The sorted array is :\n";
        for(i=0;i<n;i++)
```

```
        cout<<"  "<<a[i];
        cout<<"\n Time taken is :"<<(end-start)/(double)CLOCKS_PER_SEC;
        getch();
}
```

ANALYSIS

How efficient is mergesort? Assuming for simplicity that *n* is a power of 2, the recurrence relation for the number of key comparisons *C(n)* is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad \text{initial condition } C(1) = 0.$$

Best Case Efficiency: When the whole array is sorted already then it is best case. The number of comparisons to merge two sorted arrays i.e left subarray and right subarray is n/2. Hence, the recurrence relation is given by,

$$C_{best}(n) = 2C_{best}(n/2) + n/2 \quad \text{for } n > 1, \text{ initial condition } C_{best}(1) = 0.$$

Substituting $n = 2^k$

$C_{best}(2^k) = 2C_{best}(2^{k-1}) + 2^{k-1}$

$= 2(2C_{best}(2^{k-2}) + 2^{k-2}) + 2^{k-1}$

$= 2^2C_{best}(2^{k-2}) + 2.2^{k-2} + 2^{k-1}$

$= 2^2C_{best}(2^{k-2}) + 2^{k-1} + 2^{k-1}$

$= 2^2C_{best}(2^{k-2}) + 2.2^{k-1}$

$= 2^2(2C_{best}(2^{k-3}) + 2^{k-3}) + 2.2^{k-1}$

$= 2^3C_{best}(2^{k-3}) + 2^{2}.2^{k-3} + 2.2^{k-1}$

$= 2^3C_{best}(2^{k-3}) + 2^{k-1} + 2.2^{k-1}$

$= 2^3C_{best}(2^{k-3}) + 3.2^{k-1}$

Generalizing the above statement by replacing 3 by i we get,

$C_{best}(2^k) = 2^iC_{best}(2^{k-i}) + i.2^{k-1}$

Substitute i = k to apply initial condition $C_{best}(1) = 0$

$C_{best}(2^k) = 2^kC_{best}(2^{k-k}) + k.2^{k-1}$

$C_{best}(2^k) = 2^kC_{best}(2^0) + k.2^{k-1}$

$= 0 + k.2^{k-1}$

Replacing $k = \log_2 n$ as $n = 2^k$

$C_{best}(n) = n/2 \log_2 n = 1/2n \log_2 n$

$C_{best}(n) \in \Omega(n \log_2 n)$

Worst Case Efficiency: Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \qquad \text{for } n > 1, \text{ initial condition } C_{worst}(1) = 0.$$

Hence, according to the Master Theorem,

a=b=2 and f(n)= n– 1.

f(n) ∈ Θ(n^d). Therefore f(n) ∈ Θ(n). Hence d=1 and a = b^d($2 = 2^1$)

Therefore $C_{worst}(n) \in \Theta(n \log n)$ (why?). In fact, it is easy to find the exact solution to the worst-case recurrence for $n = 2^k$:

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1$$

Substituting $n = 2^k$

$$C_{worst}(2^k) = 2C_{worst}(2^{k-1}) + 2^k - 1$$
$$= 2(2C_{worst}(2^{k-2}) + 2^{k-1} - 1) + 2^k - 1$$
$$= 2^2 C_{worst}(2^{k-2}) + 2.2^{k-1} - 2.1 + 2^k - 1$$
$$= 2^2 C_{worst}(2^{k-2}) + 2.2^k - 2 - 1$$
$$= 2^2(2Cworst(2^{k-3}) + 2^{k-2} - 1) + 2.2^k - 2 - 1$$
$$= 2^3 Cworst(2^{k-3}) + 2^{2.} 2^{k-2} - 2^2 + 2.2^k - 2 - 1$$
$$= 2^3 Cworst(2^{k-3}) + 2^k - 2^2 + 2.2^k - 2 - 1$$
$$= 2^3 Cworst(2^{k-3}) + 3.2^k - 2^2 - 2 - 1$$
$$= 2^3 Cworst(2^{k-3}) + 3.2^k - (2^3 - 1)$$

Generalizing the above statement by replacing 3 by i we get,

$C_{worst}(2^k) = 2^i Cworst(2^{k-i}) + i.2^k - (2^i - 1)$

Substitute i = k to apply initial condition $C_{worst}(1) = 0$

$$C_{worst}(2^k) = 2^k Cworst(2^{k-k}) + k.2^k - (2^k - 1)$$

$C_{worst}(2^k) = 2^k C_{worst}(2^0) + k.2^k - (2^k - 1)$

$= 0 + k.2^k - (2^k - 1)$

Replacing $k = \log_2 n$ as $n = 2^k$

$C_{worst}(n) = n \log_2 n - n+1$

$C_{worst}(n) \in O(n \log_2 n)$

Average Case Efficiency: In average case, all the elements are distinct and arrangements of elements are equally likely. Average case is nearly equal to worst case with the following:

1. Two largest elements are in different subarray and require (n -1) comparisons. The merge happens approximately ½ the time.
2. Two largest elements are in same subarray and third largest element is in the other subarray. It requires (n – 2) comparisons. The merge happens approximately $1/4^{th}$ time.
3. Three largest elements are in same sub array and the 4^{th} largest element is in the other sub array. It requires (n – 3) comparisons. The merge happens approximately $1/8^{th}$ time.

$C_{avg}(n) = \sum_{i=Dn} P(i) * t(i)$

Where P(i) = probability of event i

t(i) = work done on input i

Dn = set of distinct inputs

$C_{avg}(n) = (n – 1) / 2 + (n – 2)/4 + (n – 3)/8 + \ldots\ldots$

$C_{avg}(n) = n[1/2 + 1/2^2 + 1/2^3 + \ldots.] – [½ - 2/2^2 – 3/2^3 - \ldots\ldots..]$

$C_{avg}(n) = n[(1 – (1/2)^n) / (1 – ½)] - \sum_{i=1}^{n-1} i.(1/2)^i$

$= n \sum_{i=1}^{n-1} (1/2)^i - \sum_{i=1}^{n-1} i.(1/2)^i$

$= n – n(1/2)^{n-1} – 1(1/2)^{n-1}$

$= \Theta(n)$

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

The problem is to place *n* queens on an *n*×*n* chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. For *n*= 1, the problem has a trivial solution, and it is easy to see that there is no solution for *n*= 2 and *n*= 3. So let us consider the four-queens problem and solve it by the backtracking technique. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board.

We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem.

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9               or (Abs(x[j] − i) = Abs(j − k)))
10                  // or in the same diagonal
11              then return false;
12      return true;
13  }
```

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

```
#include<stdio.h>
#include<conio.h>
#include<time.h>
int i,j,k,cnt;
void ps(int n,int x[20])]
{
  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");
  }

int place(int x[20],int k)
{
for(i=1;i<=k-1;i++)
{
if(x[i]-x[k]==i-k || x[i]==x[k] || i-x[i]==k-x[i] || i+x[i]==k+x[k])
return 0;
   }
return 1;
}

void nq(int n)
{
   int x[20];
   cnt=0;
   k=1;
   x[k]=0;
   while(k!=0)
   {
  x[k]+=1;
 while(x[k]<=n && (!place(x,k)))
{
x[k]+=1;
}
     if(x[k]<=n)
     {
       if(k==n)
       {
cnt++;
printf("solution is %d\n",cnt);
```

```
ps(n,x);
      }
   else
{
      k++;
      x[k]=0;
}
  }
 else
      k--;
}
}
void main()
 {
      int n;
      clrscr();
      printf("enter the number of queens\n");
      scanf("%d",&n);
      nq(n);
      getch();
 }
```

OUTPUT

Enter the no. of Queens: 4

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

Solution 2 is X X Q X
Q X X X
X X X Q
X Q X X

Analysis of n-Queens Problem:

For Place( ), $C(n) = \sum_{}^{k-1} 1 = k\text{-}1 \text{ -}1 \text{ +}1 = k - 1$

j =1

For, N-Queens( )
$C(n) = n * C(n – 1) + O(n^2)$

Hence the best case, worst case and average case efficiency = O(n!)