GR20 Regulations

III B.Tech I Semester

Algorithm Design And Analysis Lab

(GR20A3077)

Department of AIML Engineering

(Computer Science and Business System)

GOKARAJU RANGARAJU

INSTITUTE OF ENGINEERING AND TECHNOLOGY

(Autonomous)

GOKARAJU RANGARAJU INSTITUTE OF ENGINEERING AND TECHNOLOGY

ALGORITHM DESIGN AND ANALYSIS LAB

Course Code:GR20A3077   **L/T/P/C: 0/ 0/ 2/2**

III Year I Semester

Course Objectives:

1. Learn how to analyze a problem and design the solution for the problem.

2. Design and implement efficient algorithms for a specified application.

3. Strengthen the ability to identify and apply the suitable algorithm for the given real world problem.

4. Design and implement various algorithms in C.

5. Employ various design strategies for problem solving.

Course Outcomes:

1. Ability to write programs in C to solve problems using algorithm design techniques.

2. Compare and Measure the performance of different algorithms.

3. Write programs in C to solve problems using divide and conquer strategy.

4. Implement programs in C to solve problems using backtracking strategy.

5. To write programs in C to solve minimum spanning tree for undirected graphs using Krushkal's and prim's algorithms.

List of s:

1. Implement and analyze time complexity in best & worst case for Binary Search and Quick Sort

2. Implement and analyze time complexity in best & worst case for Merge Sort and Strassen Matrix Multiplication

3. Implement and analyze time complexity of Greedy Application Problems.

4. Implement and analyze time complexity of Dynamic Programming Application Problems.

5. Implement and analyze time complexity of Greedy Application Problems, Prims & Kruskal's Algorithms

6. Implement and analyze time complexity of Backtracking Application Problems.

7. Implement and analyze time complexity of Branch & Bound Application Problems.

8. Implement and analyze time complexity of BFS and DFS and their applications.

9. Implement and analyze time complexity of Dijkstra and Floyd Warshall Algorithms.

10. Implement and analyze time complexity of Topological sorting, Network Flow Problems.

11. Implement sample problem on P, NP, NP complete and NP hard

12. Implement and analyze time complexity of Randomized Quick Sort.

TEXT BOOKS:

1. Fundamental of Computer Algorithms, E. Horowitz and S. Sahni

2. The Design and Analysis of Computer Algorithms, A. Aho, J. Hopcroft and J. Ullman

REFERENCE BOOKS:

1. Introduction to Algorithms, T. H. Cormen, C. E. Leiserson and R. L. Rivest

2. Computer Algorithms: Introduction to Design and Analysis, S. Baase

Index

TASK 1 (Binary search and Quick sort)

Aim : To implement and analyze time complexity in best and worst cases for Binary Search and Quick Sort

Binary search

Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach, the element is always searched in the middle of a portion of an array. Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

Binary Search Working

Binary Search Algorithm can be implemented in two ways which are discussed below.

Iterative Method

Recursive Method

The recursive method follows the divide and conquer approach.

Algorithm:iterative method

```
do until the pointers low and high meet each other.
   mid = (low + high)/2
  if (x == arr[mid])
     return mid
  else if (x > arr[mid]) // x is on the right side
     low = mid + 1
  else                // x is on the left side
     high = mid - 1
```

Program:

```
#include <stdio.h>
int main()
{
int i, low, high, mid, n, key, array[100];
printf("Enter number of elements");
scanf("%d",&n);
printf("Enter %d integersn", n);
for(i = 0; i < n; i++)
```

```c
scanf("%d",&array[i]);
printf("Enter value to find");
scanf("%d", &key);
low = 0;
high = n - 1;
mid = (low+high)/2;
while (low <= high) {
if(array[mid] < key)
low = mid + 1;
else if (array[mid] == key) {
printf("%d found at location %d", key, mid+1);
break;
}
else
high = mid - 1;
mid = (low + high)/2;
}
if(low > high)
printf("Not found! %d isn't present in the list.n", key);
return 0;}
```

Output:

If key is found

If key is not found

Binary Search Using Recursive Function:

Algorithm

```
binarySearch(arr, x, low, high)
  if low > high
    return False
  else
    mid = (low + high) / 2
    if x == arr[mid]
      return mid
    else if x > arr[mid]        // x is on the right side
      return binarySearch(arr, x, mid + 1, high)
    else                        // x is on the right side
      return binarySearch(arr, x, low, mid - 1)
```

Program:

```c
#include <stdio.h>
int binaryScr(int a[], int low, int high, int m)
{
if (high >= low) {
int mid = low + (high - low) / 2;
if (a[mid] == m)
return mid;
if(a[mid] > m)
return binaryScr(a, low, mid - 1, m);
return binaryScr(a, mid + 1, high, m);
}
return -1;
}
int main(void)
{
int a[] = { 12, 13, 21, 36, 40 };
int i,m;
for(i=0;i<5;i++)
{
printf(" %d",a[i]);
}
printf(" n");
int n = sizeof(a) / sizeof(a[0]);
printf("Enter the number to be searchedn");
scanf("%d", &m);
int result = binaryScr(a, 0, n - 1, m);
(result == -1) ? printf("The element is not present in array"):
printf("The element is present at index %d",result);
return 0;
}
```

Output:

QUICK SORT:

Quicksort is a sorting algorithm based on the divide and conquer approach where

1.An array is divided into sub arrays by selecting a pivot element (element selected from the array).

While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

2.The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

3.At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Algorithm:

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
```

```
    pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)


partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
  for i <- leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Program:

```c
#include<stdio.h>
void quicksort(int number[25],int first,int last){
int i, j, pivot, temp;
if(first<last){
i=first;
j=last;
while(i<j){
while(number[i]<=number[pivot]&&i<last)
i++;
while(number[j]>number[pivot])
j--;
if(i<j){
temp=number[i];
number[i]=number[j];
number[j]=temp;
}
```

```c
}
temp=number[pivot];
number[pivot]=number[j];
number[j]=temp;
quicksort(number,first,j-1);
quicksort(number,j+1,last);
}
}
int main(){
int i, count, number[25];
printf("Enter some elements (Max. - 25): ");
scanf("%d",&count);
printf("Enter %d elements: ", count);
for(i=0;i<count;i++)
scanf("%d",&number[i]);
quicksort(number,0,count-1);
printf("The Sorted Order is: ");
for(i=0;i<count;i++)
printf(" %d",number[i]);
return 0;
}
```

Output:

```
Enter some elements (Max. - 25): 5
Enter 5 elements: 34
9
76
56
12
The Sorted Order is:  9 12 34 56 76
------------------------------
Process exited after 44.29 seconds with return value 0
Press any key to continue . . .
```

Time Complexity

Best	$O(n*\log n)$
Worst	$O(n^2)$
Average	$O(n*\log n)$

TASK 2(Merge sort and Strassen Matrix Multiplication):

Aim:Implement and analyze time complexity in best & worst case for Merge Sort and Strassen Matrix Multiplication

MERGE SORT:

Merge sort runs in O (n log n) running time. It is very efficient sorting algorithm with near optimal number of comparison. Recursive algorithm used for merge sort comes under the category of divide and conquer technique. An array of n elements is split around its center producing two smaller arrays. After these two arrays are sorted independently, they can be merged to produce the final sorted array. The process of splitting and merging can be carried recursively till there is only one element in the array. An array with 1 element is always sorted.

Algorithm:

MergeSort(arr[], l, r)

If r > l

   1. Find the middle point to divide the array into two halves:

     middle m = l+ (r-l)/2

   2. Call mergeSort for first half:

     Call mergeSort(arr, l, m)

   3. Call mergeSort for second half:

     Call mergeSort(arr, m+1, r)

   4. Merge the two halves sorted in step 2 and 3:

     Call merge(arr, l, m, r)

Program:

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
```

```c
int main()
{
int a[30],n,i;
printf("Enter no of elements:");
scanf("%d",&n);
printf("Enter array elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
mergesort(a,0,n-1);
printf("\nSorted array is :");
for(i=0;i<n;i++)
printf("%d ",a[i]);
return 0;
}
void mergesort(int a[],int i,int j)
{
int mid;
if(i<j)
{
mid=(i+j)/2;
mergesort(a,i,mid); //left recursion
mergesort(a,mid+1,j); //right recursion
merge(a,i,mid,mid+1,j); //merging of two sorted sub-arrays
}
}
void merge(int a[],int i1,int j1,int i2,int j2)
{
int temp[50]; //array used for merging
int i,j,k;
i=i1; //beginning of the first list
j=i2; //beginning of the second list
k=0;
while(i<=j1 && j<=j2) //while elements in both lists
{
```

```
if(a[i]<a[j])
temp[k++]=a[i++];
else
temp[k++]=a[j++];
}
while(i<=j1) //copy remaining elements of the first list
temp[k++]=a[i++];
while(j<=j2) //copy remaining elements of the second list
temp[k++]=a[j++];
//Transfer elements from temp[] back to a[]
for(i=i1,j=0;i<=j2;i++,j++)
a[i]=temp[j];
}
```

Output:

```
Enter no of elements:5
Enter array elements:56
9
87
6
54

Sorted array is :6 9 54 56 87
-----------------------------
Process exited after 10.39 seconds with return value 0
Press any key to continue . . .
```

Time complexity:

Best Case Time Complexity: **O(n*log n)**

Worst Case Time Complexity: **O(n*log n)**

Average Time Complexity: **O(n*log n)**

11

STRASSEN MATRIX MULTIPLICATION:

Algorithm:

The major work in matrix multiplication is multiplication only. So, the idea is:- If we reduced the number of multiplications then that will make the matrix multiplication faster.

Strassen had given another algorithm for finding the matrix multiplication. Unlike a simple divide and conquer method which uses 8 multiplications and 4 additions, Strassen's algorithm uses 7 multiplications which reduces the time complexity of the matrix multiplication algorithm a little bit.

The addition and Subtraction operation takes less time compared to the multiplication process. In Strassen's matrix multiplication algorithm, the number of multiplication was reduced but the number of addition and subtraction increased.

Algorithm STRESSEN_MAT_MUL (int *A, int *B, int *C, int n)

// A and B are input matrices

// C is the output matrix

// All matrices are of size n   n

if n == 1 then

  *C = *C + (*A) * (*B)

else

  STRESSEN_MAT_MUL (A, B, C, n/4)

  STRESSEN_MAT_MUL (A, B + (n/4), C + (n/4), n/4)

  STRESSEN_MAT_MUL (A + 2 * (n/4), B, C + 2 * (n/4), n/4)

  STRESSEN_MAT_MUL (A + 2 * (n/4), B + (n/4), C + 3 * (n/4), n/4)

STRESSEN_MAT_MUL (A + (n/4), B + 2 * (n/4), C, n/4)

STRESSEN_MAT_MUL (A + (n/4), B + 3 * (n/4), C + (n/4), n/4)

STRESSEN_MAT_MUL (A + 3 * (n/4), B + 2 * (n/4), C + 2 * (n/4), n/4)

STRESSEN_MAT_MUL (A + 3 * (n/4), B + 3 * (n/4), C + 3 * (n/4), n/4)

end

program:

```c
#include<stdio.h>
int main(){
 int a[2][2], b[2][2], c[2][2], i, j;
 int m1, m2, m3, m4 , m5, m6, m7;
 printf("Enter the 4 elements of first matrix: ");
 for(i = 0;i < 2; i++)
    for(j = 0;j < 2; j++)
        scanf("%d", &a[i][j]);
 printf("Enter the 4 elements of second matrix: ");
 for(i = 0; i < 2; i++)
    for(j = 0;j < 2; j++)
        scanf("%d", &b[i][j]);
printf("\nThe first matrix is\n");
 for(i = 0; i < 2; i++){
    printf("\n");
    for(j = 0; j < 2; j++)
        printf("%d\t", a[i][j]);
}
 printf("\nThe second matrix is\n");
 for(i = 0;i < 2; i++){
    printf("\n");
```

```c
        for(j = 0;j < 2; j++)

            printf("%d\t", b[i][j]);

    }

     m1= (a[0][0] + a[1][1]) * (b[0][0] + b[1][1]);

    m2= (a[1][0] + a[1][1]) * b[0][0];

    m3= a[0][0] * (b[0][1] - b[1][1]);

    m4= a[1][1] * (b[1][0] - b[0][0]);

    m5= (a[0][0] + a[0][1]) * b[1][1];

    m6= (a[1][0] - a[0][0]) * (b[0][0]+b[0][1]);

    m7= (a[0][1] - a[1][1]) * (b[1][0]+b[1][1]);

    c[0][0] = m1 + m4- m5 + m7;

    c[0][1] = m3 + m5;

    c[1][0] = m2 + m4;

    c[1][1] = m1 - m2 + m3 + m6;

     printf("\nAfter multiplication using Strassen's algorithm \n");

     for(i = 0; i < 2 ; i++){

       printf("\n");

       for(j = 0;j < 2; j++)

            printf("%d\t", c[i][j]);

    }

     return 0;

}
```

Output:

```
Enter the 4 elements of first matrix: 4 5 6 7
Enter the 4 elements of second matrix: 7 8 9 5

The first matrix is

4       5
6       7
The second matrix is

7       8
9       5
After multiplication using Strassen's algorithm

73      57
105     83
--------------------------------
Process exited after 10.68 seconds with return value 0
Press any key to continue . . .
```

Time Complexity

- Worst case time complexity: **Θ(n^2.8074)**

- Best case time complexity: **Θ(1)**

- Space complexity: **Θ(logn)**

TASK 3   (Greedy Application Problems):

Aim: Implement and analyze time complexity of Greedy Application Problems

A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

Greedy algorithms have some advantages and disadvantages:

1.It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem.

2.Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.

3.The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

Application of greedy method:

● Fractional Knapsack problem

● Job sequencing with dead lines

● Minimum cost spanning tree

● Single source shortest path

Job sequencing with deadlines:

The problem of Job sequencing with deadlines can be easily solved using the greedy algorithm.

The job which gives you maximum profit: In this case, the job may have a long

deadline and by doing this job you may miss some jobs with a short deadline. And it can be unprofitable to you if the sum of profits from missed jobs is greater than the profit of the job which you are doing.

The job which has a short deadline: In this case, It may happen that you have completed multiple jobs within the given deadlines but there could be the possibility that you could have earned more if you would have chosen the job with max profit.

To solve the ambiguity of which job to choose, we use greedy approach.

Algorithm:

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

D(0) := J(0) := 0

k := 1

J(1) := 1   // means first job is selected

for i = 2 … n do

  r := k

  while D(J(r)) > D(i) and D(J(r)) ≠ r do

    r := r − 1

  if D(J(r)) ≤ D(i) and D(i) > r then

    for l = k … r + 1 by -1 do

      J(l + 1) := J(l)

      J(r + 1) := i

      k := k + 1

Program

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#define MIN(a, b) (a < b ? a : b)


typedef struct job_ {
```

```c
    int id, deadline, profit;
}job;


int compare(const void *a, const void *b) { // descending
    return (((job *) a)->profit < ((job *) b)->profit);
}


void schedule(job data[], int n) {
    int i, j, check[n], ans = 0;
    memset(check, 0, sizeof(check));
    for (i = 0; i < n; i++) {
            for (j = MIN(data[i].deadline, n)-1; j >= 0; j--) {
                    if (!check[j]) {
                            check[j] = data[i].id;
                            ans += data[i].profit;
                            break;
                    }
            }
    }
    printf("The sequence of job is: \n");
    for (i = 0; i < n; i++)
            if (check[i])
                    printf("%d ", check[i]);
    printf("\nThe max profit is: %d\n", ans);
}
int main() {
    job data[10];
    int n, i, j;
```

```
    printf("Enter number of jobs:\n");

    scanf("%d", &n);

    printf("Enter jobs in the order (id deadline profit):\n");

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

            scanf("%d%d%d", &data[i].id, &data[i].deadline, &data[i].profit);

    qsort(data, n, sizeof(job), compare);

    schedule(data, n);

    return 0;

}
```

Output:

```
Enter number of jobs:
3
Enter jobs in the order (id deadline profit):
5
6
3
4
2
7
56
7
34
The sequence of job is:
5 4 56
The max profit is: 44

--------------------------------
Process exited after 10.18 seconds with return value 0
Press any key to continue . . .
```

Time complexity of job sequencing with deadline using greedy algorithm **O(*n²*)**

TASK 4(Dynamic program ming applications):

Aim: Implement and analyze dynamic programming applications

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

Applications of dynamic programming:

- All pair shortest path problem

- Matrix chain multiplication

- Travelling sales man problem

- 0/1 knapsack problem

- Optimal binary search tree(OBST)

Here we are discussing 0/1 kanpsack problem

We are given n items with some weights and corresponding values and a knapsack of capacity W. The items should be placed in the knapsack in such a way that the total value is maximum and total weight should be less than knapsack capacity.

In this problem 0-1 means that we can't put the items in fraction. Either put the complete item or ignore it. Below is the solution for this problem in C using dynamic programming.

Algorithm:

Dynamic-0-1-knapsack (v, w, n, W)

for w = 0 to W do

  $c[0, w] = 0$

for i = 1 to n do

  $c[i, 0] = 0$

  for w = 1 to W do

    if $w_i \leq w$ then

      if $v_i + c[i-1, w-w_i]$ then

        $c[i, w] = v_i + c[i-1, w-w_i]$

```
    else c[i, w] = c[i-1, w]
  else
    c[i, w] = c[i-1, w]
```

Program:

```
#include<stdio.h>
int w[10],p[10],v[10][10],n,i,j,cap,x[10]={0};
int max(int i,int j)
{
return ((i>j)?i:j);
}
int knap(int i,int j)
{
int value;
if(v[i][j]<0)
{
if(j<w[i])
value=knap(i-1,j);
else
value=max(knap(i-1,j),p[i]+knap(i-1,j-w[i]));
v[i][j]=value;
}
return(v[i][j]);
}
void main()
{
int profit,count=0;
printf("\nEnter the number of elements\n");
scanf("%d",&n);
printf("Enter the profit and weights of the elements\n");
for(i=1;i<=n;i++)
{
printf("For item no %d\n",i);
scanf("%d%d",&p[i],&w[i]);
}
printf("\nEnter the capacity \n");
scanf("%d",&cap);
for(i=0;i<=n;i++)
for(j=0;j<=cap;j++)
```

```c
if((i==0)||(j==0))

v[i][j]=0;
else
v[i][j]=-1;
profit=knap(n,cap);
i=n;
j=cap;
while(j!=0&&i!=0)
{
if(v[i][j]!=v[i-1][j])
{
x[i]=1;
j=j-w[i];
i--;
}
else
i--;

}
printf("Items included are\n");
printf("Sl.no\tweight\tprofit\n");
for(i=1;i<=n;i++)
if(x[i])
printf("%d\t%d\t%d\n",++count,w[i],p[i]);

printf("Total profit = %d\n",profit);
getch();
}
```

Output:

```
Enter the number of elements
3
Enter the profit and weights of the elements
For item no 1
67 78
For item no 2
78
89
For item no 3
87
89

Enter the capacity
100
Items included are
Sl.no   weight  profit
Total profit = 100
_
```

Time complexity: O(N*W)

N is the number of items available w is capacity of knapsack

TASK 5   (Prim's and Kruskal's Algorithms):

Aim: To implement and analyze time complexity of Greedy application Problems,Prim's and Kruskal's algorithm.

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

Algorithm:

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has mini mum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T
5. [END OF LOOP]
6. Step 5: EXIT

Program:

```
#include<stdio.h>

int a,b,u,v,n,i,j,ne=1;

int visited[10]={0},min,mincost=0,cost[10][10];

void main()

{

printf("\n Enter the number of nodes:");
```

```c
scanf("%d",&n);

printf("\n Enter the adjacency matrix:\n");

for(i=1;i<=n;i++)

for(j=1;j<=n;j++)

{

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

if(cost[i][j]==0)

cost[i][j]=999;

}

visited[1]=1;

printf("\n");

while(ne<n)

{

for(i=1,min=999;i<=n;i++)

for(j=1;j<=n;j++)

if(cost[i][j]<min)

if(visited[i]!=0)

{

min=cost[i][j];

a=u=i;

b=v=j;

}

if(visited[u]==0 || visited[v]==0)

{

printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);

mincost+=min;

visited[b]=1;

}
```

```
        cost[a][b]=cost[b][a]=999;

    }

    printf("\n Minimun cost=%d",mincost);

    getch();

    }
```

Output:

```
Enter the number of nodes:4

Enter the adjacency matrix:
0 34 23 12
6 0 45 24
67 47 0 16
17 19 99 0


Edge 1:(1 4) cost:12
Edge 2:(4 2) cost:19
Edge 3:(1 3) cost:23
Minimun cost=54
```

Time complexity of Prims Algorithm is O(E log V)

o **Time Complexity**

Data structure used for the minimum edge weight	Time Complexity
Adjacency matrix, linear searching	$O(|V|^2)$
Adjacency list and binary heap	$O(|E| \log |V|)$
Adjacency list and Fibonacci heap	$O(|E|+ |V| \log |V|)$

Kruskal's Algorithm:

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

Algorithm

1. Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
2. Step 2: Create a set E that contains all the edges of the graph.
3. Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning
4. Step 4: Remove an edge from E with minimum weight
5. Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F
6. (**for** combining two trees into one tree).
7. ELSE
8. Discard the edge
9. Step 6: END

Program:

```
#include<stdio.h>

#include<stdlib.h>

int i,j,k,a,b,u,v,n,ne=1;

int min,mincost=0,cost[9][9],parent[9];

int find(int);

int uni(int,int);

void main()

{

printf("\n\n\tImplementation of Kruskal's algorithm\n\n");

printf("\nEnter the no. of vertices\n");
```

```c
scanf("%d",&n);
printf("\nEnter the cost adjacency matrix\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=999;
}
}
printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
while(ne<n)
{
for(i=1,min=999;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if(cost[i][j]<min)
{
min=cost[i][j];
a=u=i;
b=v=j;
}
}
}
u=find(u);
v=find(v);
```

```c
if(uni(u,v))

{

printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);

mincost +=min;

}

cost[a][b]=cost[b][a]=999;

}

printf("\n\tMinimum cost = %d\n",mincost);

getch();

}

int find(int i)

{

while(parent[i])

 i=parent[i];

return i;

}

int uni(int i,int j)

{

if(i!=j)

{

 parent[j]=i;

 return 1;

}

return 0;

}
```

Output:

```
        Implementation of Kruskal's algorithm


Enter the no. of vertices
4

Enter the cost adjacency matrix
0 30 40 450
10 0 20 50
60 70 0 80
90 100 99 0

The edges of Minimum Cost Spanning Tree are


1 edge (2,1) =10

2 edge (2,3) =20

3 edge (2,4) =50

        Minimum cost = 80
```

Time complexity:

The time complexity of Kruskal's algorithm is O(E logE) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

TASK 6(Backtracking Application problems):

Aim: Implement and analyze time complexity of Backtracking Application Problems

Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy. It uses the Brute force search to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions. This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems. In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.

Applications of Backtracking

o N-queen problem

o Sum of subset problem

o Graph coloring

o Hamiliton cycle

Here we are discussing the N-queen problem

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

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

Rules:

No two queens on the same row

No two queens on the same column

No two queens on the same diagonal

Algorithm:

```
1) Start in the leftmost column
2) If all queens are placed
   return true
```

3) Try all rows in the current column.

  Do following for every tried row.

  a) If the queen can be placed safely in this row

    then mark this [row, column] as part of the

    solution and recursively check if placing

    queen here leads to a solution.

  b) If placing the queen in [row, column] leads to

    a solution then return true.

  c) If placing queen doesn't lead to a solution then

    unmark this [row, column] (Backtrack) and go to

    step (a) to try other rows.

4) If all rows have been tried and nothing worked,

  return false to trigger backtracking.

Program:

```c
#include<math.h>
#include<stdio.h>
int a[30],count=0;
int place(int pos)
{
int i;
for(i=1;i<pos;i++)
{
if((a[i]==a[pos])||((abs(a[i]-a[pos])==abs(i-pos))))
return 0;
}
return 1;
}
void print_sol(int n)
{
int i,j;
```

```
count++;

printf("\n\nSolution #%d:\n",count);

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{

if(a[i]==j)

printf("Q\t");

else

printf("*\t");

}

printf("\n");

}

}

void queen(int n)

{

int k=1;

a[k]=0;

while(k!=0)

{

a[k]=a[k]+1;

while((a[k]<=n)&&!place(k))

a[k]++;

if(a[k]<=n)

{

if(k==n)

print_sol(n);

else
```

```c
{
k++;
a[k]=0;
}
}
else
k--;
}
}
void main()
{
int i,n;
printf("Enter the number of Queens\n");
scanf("%d",&n);
queen(n);
printf("\nTotal solutions=%d",count);
getch();
}
```

Output:

```
Enter the number of Queens
4


Solution #1:
*       Q       *       *
*       *       *       Q
Q       *       *       *
*       *       Q       *


Solution #2:
*       *       Q       *
Q       *       *       *
*       *       *       Q
*       Q       *       *

Total solutions=2
```

Time complexity:O(n^2)

TASK 7 (Branch & Bound Application Problems):

.

Aim:Implement and analyze time complextity of Branch and Bound Application Problems

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.

The Branch and Bound Algorithm technique solves these problems relatively quickly

Program:

```
#include<stdio.h>
int a[10][10],visited[10],n,cost=0;

void get()
{
int i,j;
printf("Enter No. of Cities: ");
scanf("%d",&n);
printf("\nEnter Cost Matrix: \n");
for( i=0;i<n;i++)
{
printf("\n Enter Elements of Row # : %d\n",i+1);
for( j=0;j<n;j++)
scanf("%d",&a[i][j]);
visited[i]=0;
}
printf("\n\nThe cost list is:\n\n");
for( i=0;i<n;i++)
```

```c
{
printf("\n\n");
for( j=0;j<n;j++)
printf("\t%d",a[i][j]);
}
}
void mincost(int city)
{
int i,ncity;
visited[city]=1;
printf("%d ->",city+1);
ncity=least(city);
if(ncity==999)
{
ncity=0;
printf("%d",ncity+1);
cost+=a[city][ncity];
return;
}
mincost(ncity);
}
int least(int c)
{
int i,nc=999;
int min=999,kmin;
for(i=0;i<n;i++)
{
if((a[c][i]!=0)&&(visited[i]==0))
if(a[c][i]<min)
{
min=a[i][0]+a[c][i];
kmin=a[c][i];
nc=i;
}
}
```

```
}
if(min!=999)
cost+=kmin;
return nc;
}

void put()
{
printf("\n\nMinimum cost:");
printf("%d",cost);
}
void main()
{
get();
printf("\n\nThe Path is:\n\n");
mincost(0);
put();
getch();
}
```

Output:

Timecomplexity:O(n^2)

TASK 8 (BFS and DFS):

Aim: Implement and analyze the time complexity of BFS and DFS and their applications

A Graph G = (V, E) is a collection of sets V and E where V is a collection of vertices and E is a collection of edges.

A Graph can be of two types:

1. Directed Graph

2. Undirected Graph

In data structures, there is a popular term known as 'Traversal'. It is the process of systematically visiting or examining (may be to update the Graph nodes) each node in a tree data structure, exactly once.

There are two most common methods to traverse a Graph:

1. Breadth First Search

2. Depth First Search

A standard BFS implementation puts each vertex of the graph into one of two categories:

Visited

Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1.Start by putting any one of the graph's vertices at the back of a queue.

2.Take the front item of the queue and add it to the visited list.

3.Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

4.Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

Program:

```c
#include<stdio.h>

int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;

void bfs(int v)

{

for(i=1;i<=n;i++)

 if(a[v][i] && !visited[i])

 q[++r]=i;

if(f<=r)

{

 visited[q[f]]=1;

 bfs(q[f++]);

}

}

void main()

{

Int v;

printf("\n Enter the number of vertices:");

scanf("%d",&n);

for(i=1;i<=n;i++)

{
```

```c
  q[i]=0;

 visited[i]=0;

}

printf("\n Enter graph data in matrix form:\n");

for(i=1;i<=n;i++)

 for(j=1;j<=n;j++)

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

printf("\n Enter the starting vertex:");

scanf("%d",&v);

bfs(v);

printf("\n The node which are reachable are:\n");

for(i=1;i<=n;i++)

 if(visited[i])

 printf("%d\t",i);

getch();

}
```

Output:


```
Enter the number of vertices:4

Enter graph data in matrix form:
0 1 0 0
0 0 0 1
0 0 0 0
0 0 1 0

Enter the starting vertex:1

The node which are reachable are:
2       3       4       ■
```

Time complexity:

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

DFS:

Depth First Search is an algorithm used to search the Tree or Graph. DFS search starts from root node then traversal into left child node and continues, if item found it stops other wise it continues.

#Algorithm

1.Start by putting any one of the graph's vertices on top of a stack.

2.Take the top item of the stack and add it to the visited list.

3.Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

4.Keep repeating steps 2 and 3 until the stack is empty.

Program:

```
#include<stdio.h>

int a[20][20],reach[20],n;
void dfs(int v)
{
int i;
reach[v]=1;
for(i=1;i<=n;i++)
if(a[v][i] && !reach[i])
{
printf("\n %d->%d",v,i);
dfs(i);
}
}
void main()
{
int i,j,count=0;

printf("\n Enter number of vertices:");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
reach[i]=0;
for(j=1;j<=n;j++)
a[i][j]=0;
}
printf("\n Enter the adjacency matrix:\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
scanf("%d",&a[i][j]);
dfs(1);
printf("\n");
for(i=1;i<=n;i++)
```

```
{
if(reach[i])
count++;
}
if(count==n)
printf("\n Graph is connected");
else
printf("\n Graph is not connected");
getch();
}
```

Output:

```
Enter number of vertices:4

Enter the adjacency matrix:
1 3 4 6
3 8 5 7
45 8 90 3
10 20 40 50

1->2
2->3
3->4

Graph is connected
```

Time complexity:

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

TASK 9 (Dijkstra and Floyd Warshall Algorithm):

Aim: Implement and analyze the time complexity of Dijkstra and Floyd Warshall Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

Program:

```
#include<stdio.h>

#define infinity 999

void dij(int n,int v,int cost[10][10],int dist[100])

{

int i,u,count,w,flag[10],min;

for(i=1;i<=n;i++)

flag[i]=0,dist[i]=cost[v][i];

count=2;

while(count<=n)

{

min=99;

for(w=1;w<=n;w++)

if(dist[w]<min && !flag[w])

min=dist[w],u=w;

flag[u]=1;

count++;

for(w=1;w<=n;w++)

if((dist[u]+cost[u][w]<dist[w]) && !flag[w])

dist[w]=dist[u]+cost[u][w];

}
```

```c
}
void main()
{
int n,v,i,j,cost[10][10],dist[10];
printf("\n Enter the number of nodes:");
scanf("%d",&n);
printf("\n Enter the cost matrix:\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=infinity;
}
printf("\n Enter the source matrix:");
scanf("%d",&v);
dij(n,v,cost,dist);
printf("\n Shortest path:\n");
for(i=1;i<=n;i++)
if(i!=v)
printf("%d->%d,cost=%d\n",v,i,dist[i]);
getch();
}
```

Output :

```
Enter the number of nodes:4

Enter the cost matrix:
0 5 12 17
999 0 8 16
999 9 0 90
99 10 20 0

Enter the source matrix:1

Shortest path:
1->2,cost=5
1->3,cost=12
1->4,cost=17
```

Time Complexity: O(E Log V)

where, E is the number of edges and V is the number of vertices**.**

Floyd Warshall Algorithm:

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

Program:

```
#include<stdio.h>

int min(int,int);

void floyds(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++)

if(i==j)
```

```c
p[i][j]=0;

else

p[i][j]=min(p[i][j],p[i][k]+p[k][j]);

}

int min(int a,int b)

{

if(a<b)

return(a);

else

return(b);

}

void main()

{

int p[10][10],w,n,e,u,v,i,j;;

printf("\n Enter the number of vertices:");

scanf("%d",&n);

printf("\n Enter the number of edges:\n");

scanf("%d",&e);

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

p[i][j]=999;

}

for(i=1;i<=e;i++)

{

printf("\n Enter the end vertices of edge%d with its weight \n",i);

scanf("%d%d%d",&u,&v,&w);

p[u][v]=w;
```

```c
}
printf("\n Matrix of input data:\n");

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

printf("%d \t",p[i][j]);

printf("\n");

}

floyds(p,n);

printf("\n Transitive closure:\n");

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

printf("%d \t",p[i][j]);

printf("\n");

}

printf("\n The shortest paths are:\n");

for(i=1;i<=n;i++)

for(j=1;j<=n;j++)

{

if(i!=j)

printf("\n <%d,%d>=%d",i,j,p[i][j]);

}

getch();

}
```

Output:

```
 Enter the end vertices of edge5 with its weight
4 1 6

 Matrix of input data:
999     999     4       999
999     999     999     8
999     7       999     1
6       999     999     999

 Transitive closure:
0       11      4       5
14      0       18      8
7       7       0       1
6       17      10      0

 The shortest paths are:

 <1,2>=11
 <1,3>=4
 <1,4>=5
 <2,1>=14
 <2,3>=18
 <2,4>=8
 <3,1>=7
 <3,2>=7
 <3,4>=1
 <4,1>=6
 <4,2>=17
 <4,3>=10
```

Time Complexity:

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is O(n3).

TASK 10 (Topological Sorting,Network Flow Problems):

Aim: Implement and analyze the time complexity of Topological Sorting,Network Flow Problems.

Topological sort : of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; topological sorting is just a valid sequence for the tasks. In topological sorting, we need to print a vertex before its adjacent vertices. Topological Sorting for a graph is not possible if the graph is not a DAG.

Applications :

1. DAG

2. Kahn's Algorithm

3. DFS

4. Parallel Algorithms

Algorithm

---

1. Store each vertex's In-Degree in an array D

2. Initialize queue with all "in-degree=0" vertices

3. While there are vertices remaining in the queue:

      (a) Dequeue and output a vertex

      (b) Reduce In-Degree of all vertices adjacent to it by 1

      (c) Enqueue any of these vertices whose In-Degree became zero

 4. If all vertices are output then success, otherwise there is a cycle.

Program:

```c
#include<stdio.h>

int a[10][10],n,indegre[10];

void find_indegre()

{ int j,i,sum;

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

{

sum=0;

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

sum+=a[i][j];

indegre[j]=sum;

}

}

void topology()

{

int i,u,v,t[10],s[10],top=-1,k=0;

find_indegre();

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

{

if(indegre[i]==0) s[++top]=i;

}
```

```c
while(top!=-1)

{

u=s[top--];

t[k++]=u;

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

{

if(a[u][v]==1)

{

indegre[v]--;

if(indegre[v]==0) s[++top]=v;

}

}

}

printf("The topological Sequence is:\n");

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

printf("%d ",t[i]);

}

void main()

{

int i,j;

printf("Enter number of jobs:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");
```

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

{

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

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

}

topology();

getch();

}
```

Output:

```
Enter number of jobs:6

Enter the adjacency matrix:
0 0 1 1 0 0
0 0 0 1 1 0
0 0 0 1 0 1
0 0 0 0 0 1
0 0 0 0 0 1
0 0 0 0 0 0
The topological Sequence is:
1 4 0 2 3 5
```

Time complexity

The time complexity of topological sort using Kahn's algorithm is **O(V+E)**, where V = Vertices, E = Edges

Network Flow:

Flow Network is a directed graph that is used for modeling material Flow. There are two different vertices; one is a source which produces material at some steady rate, and another one is sink which consumes the content at the same constant speed. The flow of the material at any mark in the

system is the rate at which the element moves.

Some real-life problems like the flow of liquids through pipes, the current through wires and delivery of goods can be modeled using flow network**s.**

Program:

```
#include <stdio.h>

#define A 0

#define B 1

#define C 2

#define MAX_NODES 1000

#define O 1000000000

int n;

int e;

int capacity[MAX_NODES][MAX_NODES];

int flow[MAX_NODES][MAX_NODES];

int color[MAX_NODES];

int pred[MAX_NODES];

int min(int x, int y) {

  return x < y ? x : y;

}

int head, tail;

int q[MAX_NODES + 2];

void enqueue(int x) {

  q[tail] = x;
```

```
   tail++;

  color[x] = B;

}

int dequeue() {

  int x = q[head];

  head++;

  color[x] = C;

  return x;

}

// Using BFS as a searching algorithm

int bfs(int start, int target) {

  int u, v;

  for (u = 0; u < n; u++) {

    color[u] = A;

  }

  head = tail = 0;

  enqueue(start);

  pred[start] = -1;

  while (head != tail) {

   u = dequeue();

   for (v = 0; v < n; v++) {

     if (color[v] == A && capacity[u][v] - flow[u][v] > 0) {

       enqueue(v);
```

```
        pred[v] = u;

      }

    }

  }

  return color[target] == C;

}

// Applying fordfulkerson algorithm

int fordFulkerson(int source, int sink) {

  int i, j, u;

  int max_flow = 0;

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

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

      flow[i][j] = 0;

    }

  }

// Updating the residual values of edges

  while (bfs(source, sink)) {

    int increment = O;

    for (u = n - 1; pred[u] >= 0; u = pred[u]) {

      increment = min(increment, capacity[pred[u]][u] - flow[pred[u]][u]);

    }

    for (u = n - 1; pred[u] >= 0; u = pred[u]) {

      flow[pred[u]][u] += increment;
```

```
      flow[u][pred[u]] -= increment;

    }

    // Adding the path flows

    max_flow += increment;

  }

  return max_flow;

}

int main() {

    int i,j;

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

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

     capacity[i][j] = 0;

    }

  }

  n = 6;

  e = 7;

  capacity[0][1] = 8;

  capacity[0][4] = 3;

  capacity[1][2] = 9;

  capacity[2][4] = 7;

  capacity[2][5] = 2;

  capacity[3][5] = 5;

  capacity[4][2] = 7;
```

```
capacity[4][3] = 4;

int s = 0, t = 5;

printf("Max Flow: %d\n", fordFulkerson(s, t));

}
```

Output:


```
Max Flow: 6

--------------------------------
Process exited after 0.07463 seconds with return value 12
Press any key to continue . . .
```

Time complexity:

O(M*f) with integer capacities, where M is the number of edges and f the value of maximal flow,

TASK 11 (N,NP,NP complete,NP hard):

Aim:Implement sample problems on N,NP,NP complete and NP hard

In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as Complexity Classes. In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to groups problems based on how much time and space they require to solve problems and verify the solutions. It is the branch of the theory of computation that deals with the resources required to solve a problem.

The common resources are time and space, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage.

The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer.

The space complexity of an algorithm describes how much memory is required for the algorithm to operate.

Complexity classes are useful in organizing similar types of problems.

Types of Complexity Classes

This article discusses the following complexity classes:

1. P Class

2. NP Class

3. CoNP Class

4. NP hard

5. NP complete

P Class

The P in the P class stands for Polynomial Time. It is the collection of decision problems(problems with a "yes" or "no" answer) that can be solved by a deterministic machine in polynomial time.

Features:

1. The solution to P problems is easy to find.

2. P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many natural problems like:

1. Calculating the greatest common divisor.

2. Finding a maximum matching.

3. Decision versions of linear programming.

NP Class

The NP in NP class stands for Non-deterministic Polynomial Time. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Features:

1. The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.

2. Problems of NP can be verified by a Turing machine in polynomial time.

Example:

Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to some personal reasons.

This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.

It indicates that if someone can provide us with the solution to the problem, we can find the correct and incorrect pair in polynomial time. Thus for the NP class problem, the answer is possible, which can be calculated in polynomial time.

This class contains many problems that one would like to be able to solve effectively:

1. Boolean Satisfiability Problem (SAT).

2. Hamiltonian Path Problem.

3. Graph coloring.

Co-NP Class

Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is No, then there is proof that can be checked in polynomial time.

Features:

1. If a problem X is in NP, then its complement X' is also is in CoNP.

2. For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer "yes" or "no" in

polynomial time for a problem to be in NP or CoNP.

Some example problems for C0-NP are:

1. To check prime number.

2. Integer Factorization.

NP-hard class

An NP-hard problem is at least as hard as the hardest problem in NP and it is the class of the problems such that every problem in NP reduces to NP-hard.

Features:

1. All NP-hard problems are not in NP.

2. It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.

3. A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

Some of the examples of problems in Np-hard are:

1. Halting problem.

2. Qualified Boolean formulas.

3. No Hamiltonian cycle.

4. subset sum problem

NP-complete class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hardest problems in NP.

Features:

1. NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.

2. If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

Some example problems include:

1. 0/1 Knapsack.

2. Hamiltonian Cycle.

3. Satisfiability.

4. Vertex cover.

{P} problems are quick to solve

{NP} problems are quick to verify but slow to solve

{NP}Complete problems are also quick to verify, slow to solve and can be reduced to

{NP}Complete problem

{NP}Hard problems are slow to verify, slow to solve and can be reduced to any other \mathcal{NP} problem

As a final note, if {P}={NP} has proof in the future, humankind has to construct a new way of security aspects of the computer era. When this happens, there has to be another complexity level to identify new hardness levels than we have currently.

P problem:

Program:

```c
#include <stdio.h>
#include <conio.h>
int main()
{
    // declare the variables
    int n1, n2, i, GCD_Num;
    printf ( " Enter any two numbers: \n ");
    scanf ( "%d %d", &n1, &n2);

    // use for loop
    for( i = 1; i <= n1 && i <= n2; ++i)
    {
        if (n1 % i ==0 && n2 % i == 0)
            GCD_Num = i; /* if n1 and n2 is completely divisible by i, the divisible number will be the GCD_Num */
    }
    // print the GCD of two numbers
    printf (" GCD of two numbers %d and %d is %d.", n1, n2, GCD_Num);
    return 0;

}
```

TASK 12 (<u>Randamized Quick Sort</u>):

Aim:Implement and analyze the time complexity of Randamized Quick Sort

 What is a Randomized Algorithm?

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm. For example, in Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array). And in Karger's algorithm, we randomly pick an edge.

How to analyse Randomized Algorithms?

Some randomized algorithms have deterministic time complexity. For example, this implementation of Karger's algorithm has time complexity is O(E). Such algorithms are called Monte Carlo Algorithms and are easier to analyse for worst case.

On the other hand, time complexity of other randomized algorithms (other than Las Vegas) is dependent on value of random variable. Such Randomized algorithms are called Las Vegas Algorithms. These algorithms are typically analysed for expected worst case. To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible value needs to be evaluated. Average of all evaluated times is the expected worst case time complexity. Below facts are generally helpful in analysis os such algorithms.

Linearity of Expectation :Expected Number of Trials until Success.

For example consider below a randomized version of QuickSort.

A Central Pivot is a pivot that divides the array in such a way that one side has at-least 1/4 elements.

// Sorts an array arr[low..high]

randQuickSort(arr[], low, high)

1. If low >= high, then EXIT.

2. While pivot 'x' is not a Central Pivot.

 (i)   Choose uniformly at random a number from [low..high].

    Let the randomly picked number number be x.

(ii) Count elements in arr[low..high] that are smaller

than arr[x]. Let this count be sc.

(iii) Count elements in arr[low..high] that are greater

than arr[x]. Let this count be gc.

(iv) Let n = (high-low+1). If sc >= n/4 and

gc >= n/4, then x is a central pivot.

3. Partition arr[low..high] around the pivot x.

4. // Recur for smaller elements

randQuickSort(arr, low, sc-1)

5. // Recur for greater elements

randQuickSort(arr, high-gc+1, high)

Program:

```c
#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<conio.h>

int arr[10000], n;

int partition(int arr[], int m, int p);

void interchange(int arr[], int i, int j);

void QuickSort(int arr[], int p, int q);

void show();

int main()

{

// int arr[100], n;

int i;

time_t start, end;
```

```c
time_t diff;

printf("\nEnter the number of elements: ");

scanf("%d", &n);

rand();    //Generates random elements

// printf("\nEnter the elements: ");

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

{

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

arr[i]=rand();

}

printf("\nThe elements are ");

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

{

  printf("%d ", arr[i]);

}
start=time(NULL);

printf("\n\nThe passes are:");

QuickSort(arr, 0, n-1);

end=time(NULL);

diff=difftime(end, start);

printf("\n\nThe sorted array is ");

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

  printf("%d ", arr[i]);

printf("\n\nTime taken is %ld seconds", diff);

getch();

return 0;

}
//-------FIND THE POSITION OF THE PARTITION-------
```

```
int partition(int arr[], int m, int p)

{

int v=arr[m];

int i=m;

int j=p;

do

{

  do

   i++;

  while(arr[i]<v);

  do

   j--;

  while(arr[j]>v);

  if(i<j)

   interchange(arr, i, j);

}while(i<=j);

arr[m]=arr[j];

arr[j]=v;

return j;

}
//-------INTERCHANGE THE ELEMENTS-------
void interchange(int arr[], int i, int j)

{

int p;

p=arr[i];

arr[i]=arr[j];

arr[j]=p;

}
```

```
//-------QUICK SORT-------

void QuickSort(int arr[], int p, int q)

{

int j, k, temp;

if(p<q)    //If there are more than 1 element, divide p into 2 sub-problems

{

  k=rand() % (q-p+1)+p;

  interchange(arr, k, p);
/*  temp=arr[k];


  arr[k]=arr[p];

  arr[p]=temp;  */

  j=partition(arr, p, q+1);  //j is the position of partioning element

  show();

  //Solve sub-problems

  QuickSort(arr, p, j-1);

  QuickSort(arr, j+1, q);

}

}

//-------PRINT ELEMENTS OF THE ARRAY FOR EACH PASS-------

void show()
```

```
{

   int j;

//   printf("\nPasses are: ");

   printf("\n");

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

printf("   %d", arr[j]);

}
```

Output:

```
Enter the number of elements: 5

The elements are 18467 6334 26500 19169 15724

The passes are:
   18467    6334    15724    19169    26500
   15724    6334    18467    19169    26500
    6334   15724    18467    19169    26500

The sorted array is 6334 15724 18467 19169 26500

Time taken is 0 seconds
```

Time complexity:O(n log n)