EXP. NO	DATE	NAME OF THE EXPERIMENT	PAGE NO	MARKS	SIGN
1		QUICK SORT USING DIVIDE AND CONQUER			
2		STRASSEN'S MATRIX MULTIPLICATION USING DIVIDE AND CONQUER			
3		TRANSITIVE CLOSURE USING WARSHALL ALGORITHM			
4		ALL PAIRS SHORTEST PATH USING FLOYD'S ALGORITHM			
5		TRAVELLING SALESMAN PROBLEM USING DYNAMIC PROGRAMMING			
6		KNAPSACK PROBLEM USING GREEDY METHOD			
7		SHORTEST PATH USING DIJKSTRA'S ALGORITHM			
8		MINIMUM COST SPANNING TREE USING KRUSKAL'S ALGORITHM			
9		N – QUEEN'S PROBLEM USING BACKTRACKING			

DATE:

EXNO:1

QUICK SORT USING DIVIDE AND CONQUER

AIM:

To Implement the Quick Sort Algorithm for Sorting the given Set of elements and to determine the time required to sort the Elements

ALGORITHM:

STEP1:Start

STEP2: function swaps two integer using pointers.

STEP3: It recursively divides the array into smaller sub-arrays based on a pivot element and then sorts those sub-array.

STEP4: initializes variables, asks the user for input, generates random numbers, and then calls the quicksort function.

STEP5: enter the number of elements to be sorted.

STEP6:Random numbers are generated and displayed.

STEP7: The sorted array is displayed

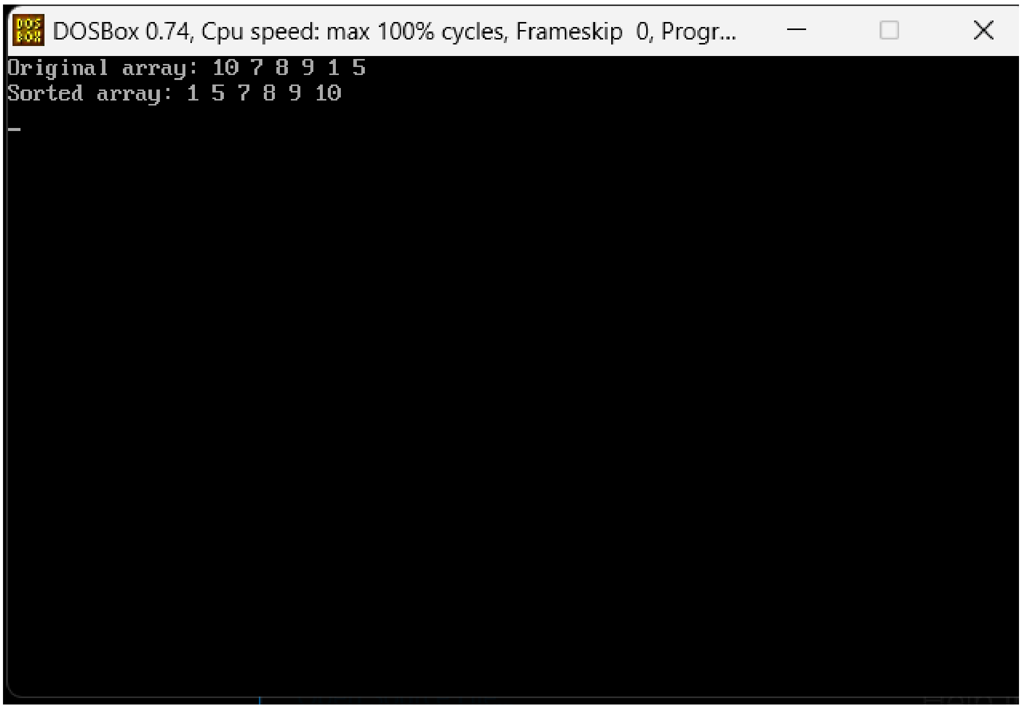
STEP8:End

PROGRAM:

```
#include <iostream.h>
#include <conio.h>
// Function to swap two elements
void swap(int* a, int* b) {
   int t = *a;
   *a = *b;
   *b = t;
}
// This function takes the last element as pivot, places
// the pivot element at its correct position in the sorted
// array, and places all smaller elements to the left of
// the pivot and all greater elements to the right
int partition(int arr[], int low, int high) {
   int pivot = arr[high]; // pivot
   int i = (low - 1); // index of smaller element

   for (int j = low; j <= high - 1; j++) {
         // If the current element is smaller than or equal to pivot
         if (arr[j] <= pivot) {
         i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
         }
   }
swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
// The main function that implements QuickSort
// arr[] --> Array to be sorted,
// low  --> Starting index,
// high  --> Ending index
void quickSort(int arr[], int low, int high) {
   if (low < high) {
         // pi is partitioning index, arr[p] is now at right place
         int pi = partition(arr, low, high);


// Separately sort elements before partition and after partition
         quickSort(arr, low, pi - 1);
         quickSort(arr, pi + 1, high);

 }
}
// Function to print an array
void printArray(int arr[], int size) {
   for (int i = 0; i< size; i++)
         cout<<arr[i] << " ";
cout<<endl;
}
// Main function to test the quick sort algorithm
int main() {
```

```
getch();

  int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
cout<< "Original array: ";
printArray(arr, n);
quickSort(arr, 0, n - 1);
cout<< "Sorted array: ";
printArray(arr, n);
clrscr();
    return 0;

}
```

OUTPUT:



RESULT:

Thus, the program was executed and output is verified successfully.

DATE:

EXNO: 2

STRASSEN'S MATRIX MULTIPLICATION USING DIVIDE AND CONQUER

AIM:

To write a program to analyse all the complexity of strassen's matrix with minimum matrix size of 4*4

ALGORITHM:

STEP1:Start

STEP2: Initialize matrices A, B.

STEP3: Divide each input matrix into four equal-sized submatrices.

STEP4: Compute different matrix products using the submatrices.

STEP5: Take input for the matrices A and B from the user.

STEP6:Callthe strassen function to perform matrix multiplication.

STEP7:Print the resultant matrix C.

STEP8: The resultant matrix C is displayed.

STEP9:End.

PROGRAM:

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

void multiply(int A[4][4], int B[4][4], int C[4][4]) {
    //Aoo*Boo
    int M1 = (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]);
    int M2 = (A[1][0] + A[1][1]) * B[0][0];
    int M3 = A[0][0] * (B[0][1] - B[1][1]);
    int M4 = A[1][1] * (B[1][0] - B[0][0]);
    int M5 = (A[0][0] + A[0][1]) * B[1][1];
    int M6 = (A[1][0] - A[0][0]) * (B[0][0] + B[0][1]);
    int M7 = (A[0][1] - A[1][1]) * (B[1][0] + B[1][1]);

    int S[2][2]={0};
S[0][0] = M1 + M4 - M5 + M7;
S[0][1] = M3 + M5;
S[1][0] = M2 + M4;
S[1][1] = M1+M3-M2+M6;


     //Ao1*B1o
    int M8 = (A[0][2] + A[1][3]) * (B[2][0] + B[3][1]);
    int M9 = (A[1][2] + A[1][3]) * B[2][0];
    int M10 = A[0][2] * (B[2][1] - B[3][1]);
    int M11 = A[1][3] * (B[3][0] - B[2][0]);
    int M12 = (A[0][2] + A[0][3]) * B[3][1];
    int M13 = (A[1][2] - A[0][2]) * (B[2][0] + B[2][1]);
    int M14= (A[0][3] - A[1][3]) * (B[3][0] + B[3][1]);

            int T[2][2]={0};

T[0][0] = M8 + M11 - M12 + M14;
T[0][1] = M10 + M12;
T[1][0] = M9+ M11;
T[1][1] = M8+M10-M9+M13;
cout<<endl;

    //ADDING TWO SUB MATRIXES

    int H[2][2]={0};
    for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
            H[i][j]=S[i][j]+T[i][j];

}
    }

             //Aoo*Bo1
    int M15 = (A[0][0] + A[1][1]) * (B[0][2] + B[1][3]);
    int M16= (A[1][0] + A[1][1]) * B[0][2];
    int M17 = A[0][0] * (B[0][3] - B[1][3]);
    int M18 = A[1][1] * (B[1][2] - B[0][2]);
```

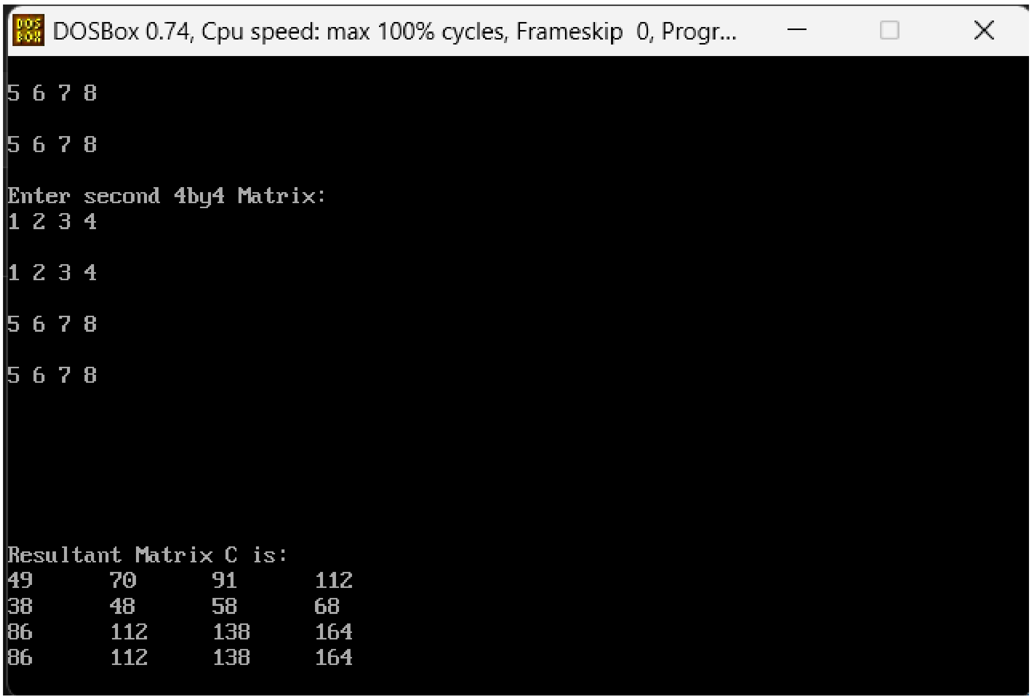
```
   int M19 = (A[0][0] + A[0][1]) * B[1][3];
    int M20 = (A[1][0] - A[0][0]) * (B[0][2] + B[0][3]);
    int M21= (A[0][1] - A[1][1]) * (B[1][2] + B[1][3]);

            int Q[2][2]={0};

Q[0][0] = M15 + M18 - M19 + M21;
Q[0][1] = M17 + M19;
Q[1][0] = M16+ M18;
Q[1][1] = M15+M17-M16+M20;

    //Ao1*B11

    int M22 = (A[0][2] + A[1][3]) * (B[2][2] + B[3][3]);
    int M23= (A[1][2] + A[1][3]) * B[2][2];
    int M24 = A[0][2] * (B[2][3] - B[3][3]);
    int M25 = A[1][3] * (B[3][2] - B[2][2]);
    int M26 = (A[0][2] + A[0][3]) * B[3][3];
    int M27 = (A[1][2] - A[0][2]) * (B[2][2] + B[2][3]);
    int M28= (A[0][3] - A[1][3]) * (B[3][2] + B[3][3]);

            int R[2][2]={0};

R[0][0] = M22 + M25 - M26 + M28;
R[0][1] = M24 + M26;
R[1][0] = M23+ M25;
R[1][1] = M22+M24-M23+M27;

            cout<<endl;
    //ADDING TWO SUB MATRIXES

    int V[2][2]={0};
    for (int u = 0; u < 2; u++) {
            for (int j = 0; j < 2; j++) {
            V[u][j]=Q[u][j]+R[u][j];
            }
    }

              //A1o*Boo

    int M29 = (A[2][0] + A[3][1]) * (B[0][0] + B[1][1]);
    int M30= (A[3][0] + A[3][1]) * B[0][0];
    int M31 = A[2][0] * (B[0][1] - B[1][1]);
    int M32 = A[3][1] * (B[1][0] - B[0][0]);
    int M33= (A[2][0] + A[2][1]) * B[1][1];
    int M34= (A[3][0] - A[2][0]) * (B[0][0] + B[0][1]);
    int M35 = (A[2][1] - A[3][1]) * (B[1][0] + B[1][1]);

            int D[2][2]={0};

D[0][0] = M29 + M32 - M33 + M35;
D[0][1] = M31 + M33;
D[1][0] = M30+ M32;
D[1][1] = M29+M31-M30+M34;
cout<<endl;
```

```
 //A11*B1o


   int M36 = (A[2][2] + A[3][3]) * (B[2][0] + B[3][1]);
   int M37= (A[3][2] + A[3][3]) * B[2][0];
   int M38 = A[2][2] * (B[2][1] - B[3][1]);
   int M39 = A[3][3] * (B[3][0] - B[2][0]);
   int M40 = (A[2][2] + A[2][3]) * B[3][1];
   int M41 = (A[3][2] - A[2][2]) * (B[2][0] + B[2][1]);
   int M42= (A[2][3] - A[3][3]) * (B[3][0] + B[3][1]);

	int P[2][2]={0};

P[0][0] = M36 + M39 - M40 + M42;
P[0][1] = M38 + M40;
P[1][0] = M37+ M39;
P[1][1] = M36+M38-M37+M41;
cout<<endl;
	//ADDING TWO SUB MATRIXES

int I[2][2]={0};
   for (int p = 0; p < 2; p++) {
	for (int j = 0; j < 2; j++) {
	I[p][j]=P[p][j]+D[p][j];
	}
   }

	//A1o*Bo1

   int M43 = (A[2][0] + A[3][1]) * (B[0][2] + B[1][3]);
   int M44= (A[3][0] + A[3][1]) * B[0][2];
   int M45 = A[2][0] * (B[0][3] - B[1][3]);
   int M46 = A[3][1] * (B[1][2] - B[0][2]);
   int M47= (A[2][0] + A[2][1]) * B[1][3];
   int M48= (A[3][0] - A[2][0]) * (B[0][2] + B[0][3]);
   int M49 = (A[2][1] - A[3][1]) * (B[1][2] + B[1][3]);

	int O[2][2]={0};

O[0][0] = M43 + M46 - M47 + M49;
O[0][1] = M45 + M47;
O[1][0] = M44+ M46;
O[1][1] = M43+M45-M44+M48;

	cout<<endl;

   int M50= (A[2][2] + A[3][3]) * (B[2][2] + B[3][3]);
   int M51 = (A[3][2] + A[3][3]) * B[2][2];
   int M52 = A[2][2] * (B[2][3] - B[3][3]);
   int M53 = A[3][3] * (B[3][2] - B[2][2]);
   int M54 = (A[2][2] + A[2][3]) * B[3][3];
   int M55 = (A[3][2] - A[2][2]) * (B[2][2] + B[2][3]);
   int M56= (A[2][3] - A[3][3]) * (B[3][2] + B[3][3]);
   int X[2][2]={0};
```

```
X[0][0] = M50 + M53 - M54 + M56;
X[0][1] = M52 + M54;
X[1][0] = M51+ M53;
X[1][1] = M50+M52-M51+M55;

                 //ADDING TWO SUB MATRIXES



    int Y[2][2]={0};
    for (int m = 0; m < 2; m++) {


for (int j = 0; j < 2; j++) {
             Y[m][j]=O[m][j]+X[m][j];
             }
    } C[0][0]=H[0][0];
    C[0][1]=H[0][1];
    C[0][2]=V[0][0];
    C[0][3]=V[0][1];
    C[1][0]=H[1][0];
    C[1][1]=H[1][1];

C[1][2]=V[1][0];
    C[1][3]=V[1][1];
    C[2][0]=I[0][0];
    C[2][1]=I[0][1];
    C[2][2]=Y[0][0];
    C[2][3]=Y[0][1];
    C[3][0]=I[1][0];
    C[3][1]=I[1][1];
    C[3][2]=Y[1][0];
    C[3][3]=Y[1][1];

}

int main() {
clrscr();
    int A[4][4] = {0};
    int B[4][4] = {0};
    int C[4][4] = {0};
cout<<"Enter First 4by4 Matrix: \n";
    for (int i = 0; i < 4; i++) {
             for (int j = 0; j < 4; j++) {
             cin>>A[i][j];

             }
             cout<<"\n";
    }
             cout<<"Enter second 4by4 Matrix: \n";

    for (int r = 0; r <4;r++) {
             for (int j = 0; j < 4; j++) {
             cin>>B[r][j];
```

```
            }
            cout<<"\n";
    }


multiply(A, B, C);

cout<< "Resultant Matrix C is: " <<endl;
    for (int l = 0; l < 4; l++) {
            for (int j = 0; j < 4; j++) {



cout<< C[l][j] << " ";
            cout<<"\t";
            }
            cout<<endl;
    }
getch();
return 0;

}
```

OUTPUT:



RESULT:

Thus,the program is executed and output is verified successfully.

DATE:

EXNO:3

TRANSITIVE CLOSURE USING WARSHALL ALGORITHM

AIM:

To compute the transitive closure of a given directed graph by warshall’s algorithm.

ALGORITHM:

STEP 1: Define a constant MAX_NODES to represent the maximum number of nodes in the graph.

STEP 2: Create a function warshall that computes the transitive closure of a graph using the Warshall's algorithm. It takes a 2D array graph representing the adjacency matrix of the graph and an integer num_nodes representing the number of nodes in the graph as input

.

STEP 3: Create a 2D array dist to store the distances between nodes. Initialize dist with the values from the adjacency matrix graph.

STEP 4: Use three nested loops to iterate over all pairs of nodes in the graph.

STEP 5: Print the transitive closure matrix dist after computing the transitive closure of the graph.

STEP 6: In the main function, prompt the user to enter the number of nodes in the graph and then input the adjacency matrix of the graph.

STEP 7: Call the warshall function with the adjacency matrix and the number of nodes to compute and output the transitive closure of the graph

STEP8:Stop

PROGRAM:

```
#include <iostream.h>

const int MAX_NODES = 100;

void warshall(int graph[MAX_NODES][MAX_NODES], int num_nodes) {
   int dist[MAX_NODES][MAX_NODES];

   // Copy the original graph to dist matrix
   for (int i = 0; i < num_nodes; ++i) {
      for (int j = 0; j < num_nodes; ++j) {
         dist[i][j] = graph[i][j];
      }
   }

   // Applying Warshall's algorithm
   for (int k = 0; k < num_nodes; ++k) {
      for (int i = 0; i < num_nodes; ++i) {
         for (int j = 0; j < num_nodes; ++j) {
            if (dist[i][k] && dist[k][j])
               dist[i][j] = 1;
         }
      }
   }

   // Printing the transitive closure matrix
   cout << "Transitive Closure Matrix:\n";
   for ( i = 0; i < num_nodes; ++i) {
      for (int j = 0; j < num_nodes; ++j) {
         cout << dist[i][j] << " ";
      }
      cout << endl;
   }
}

int main() {
   int num_nodes;
   cout << "Enter the number of nodes: ";
   cin >> num_nodes;

   int graph[MAX_NODES][MAX_NODES];
   cout << "Enter the adjacency matrix:\n";
   for (int i = 0; i < num_nodes; ++i) {
      for (int j = 0; j < num_nodes; ++j) {
         cin >> graph[i][j];
      }
   }

   warshall(graph, num_nodes);

   return 0;
}
```

OUTPUT:

```
C:\TURBOC3\BIN>TC
Enter the number of nodes: 4
Enter the adjacency matrix:
1 2 3 4
5 6 7 8
1 2 3 4
5 6 7 8
Transitive Closure Matrix:
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
Enter the number of nodes:
```

RESULT:

Thus, the program was executed and output is verified successfully.

DATE:

EXNO: 4

ALL PAIRS SHORTEST PATH USING FLOYD'S ALGORITHM

AIM:

To implement the transitive closure of a given directed graph by using Floyd's Algorithm.

ALGORITHM:

STEP 1: Include the necessary libraries, iostream and conio for input/output operations and console functions..

STEP 2: Define a constant MAX to represent a large value, used as infinity for initializing distances.

STEP 3: Define a function createDistanceMatrix that takes the adjacency matrix A, the distance matrix D, and the number of vertices n as input. Initialize the distance matrix D based on the adjacency matrix A, setting the diagonal to 0 and unreachable paths to -1.

STEP 4: Define a function floydWarshall that takes the adjacency matrix A, the distance matrix D, and the number of vertices n as input. Use the Floyd-Warshall algorithm to compute the shortest distances between all pairs of vertices.

STEP 5: Define a function printMatrix that takes the distance matrix D and the number of vertices n as input. Print the shortest distances between all pairs of vertices.

STEP 6: In the main function, initialize the adjacency matrix A and the distance matrix D. Call the floydWarshall function to compute the shortest distances. Call the printMatrix function to print the results.

STEP 7: Call the main function to execute the Floyd-Warshall algorithm and print the shortest distances between all pairs of vertices.
Requisition

STEP8:Stop

PROGRAM:

```
#include <iostream.h>

#define MAX 10 // Define the maximum size of the graph

// Function to find the shortest paths using Floyd's algorithm
void floyd(int graph[MAX][MAX], int n) {
    int dist[MAX][MAX];

    // Initialize the distance matrix
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            dist[i][j] = graph[i][j];

    // Update the distance matrix
    for (int k = 0; k < n; k++)
            for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)
                if (dist[i][k] != -1 && dist[k][j] != -1 &&
                    (dist[i][j] == -1 || dist[i][k] + dist[k][j] < dist[i][j]))
                    dist[i][j] = dist[i][k] + dist[k][j];

    // Print the shortest distances
    cout << "Shortest distances between every pair of vertices:\n";
    for (i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dist[i][j] == -1)
                        cout << "INF\t";
                else
                        cout << dist[i][j] << "\t";
            }

            cout << endl;
    }
}

int main() {
    int n; // Number of vertices
    cout << "Enter the number of vertices: ";
    cin >> n;

    // Sample graph
    int graph[MAX][MAX];

    cout << "Enter the adjacency matrix of the graph (enter -1 for infinity):\n";
    for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                cin >> graph[i][j];

    // Call the Floyd's algorithm function
    floyd(graph, n);

    return 0;
}
```

OUTPUT:

```
C:\TURBOC3\BIN>TC
Enter the number of vertices: 4
Enter the adjacency matrix of the graph (enter -1 for infinity):
0 3 -1 5
2 0 -1 4
-1 1 0 -1
-1 -1 2 0
Shortest distances between every pair of vertices:
0       3       7       5
2       0       6       4
3       1       0       5
5       3       2       0
Enter the number of vertices:
```

RESULT:

Thus,the program was executed and the output is verified successfully.

DATE:

EXNO: 5

TRAVELLING SALESMAN PROBLEM USING DYNAMIC PROGRAMMING

AIM:

To implement Dynamic Programming for Travelling Sales person problem .

ALGORITHM:

STEP 1: Start by initializing variables c and cost to 0 and 999 respectively. Declare a 2D array graph representing the cost matrix.

STEP 2: Define a function swap that takes two integers as input and swaps their values.

STEP 3: Define a function permute that generates all permutations of the array a using recursion. For each permutation, call copy_array to calculate the cost.

STEP 4: Define a function copy_array that takes an array a and its size n as input. Calculate the cost of the path represented by the permutation of a and update cost if the calculated cost is lower.

STEP 5: In the main function, initialize an array a representing the nodes (0, 1, 2, 3) and call permute to find the minimum cost of traversal.

STEP 6: Print the minimum cost calculated in the copy_array function.

STEP 7: End the program after displaying the minimum cost.

STEP 8: Stop

PROGRAM:

```
#include <stdio.h>
#include<conio.h>
#include<iostream.h>
int c=0,cost=999;
int graph[4][4]={{0,10,15,20},
{10,0,35,25},
{15,35,0,30},
{20,25,30,0}
};
void swap (int*x,int*y)
{
int temp;
temp=*x;
*x=*y;
*y=temp;
}
void copy_array(int *a,int n)
{
int i,sum=0;
for(i=0;i<=n;i++)
{
sum+= graph[a[i % 4]][a[(i + 1)% 4]];
}
if (cost> sum)
{
cost =sum;
}
}
void permute(int *a,int i,int n)
{
int j,k;
if(i==n)
{
copy_array(a,n);
}
else
{
for(j=i;j<=n;j++)
{
swap((a+i),(a+j));
permute(a,i+1,n);
swap((a+i),(a+j));
}
}
}
int main()
{
int i,j;
int a[]={0,1,2,3};
permute(a,0,3);
cout<<"minimun cost:"<<cost<<endl;
getch();
return 0;
```

}

OUTPUT:

RESULT:

Thus,the program was executed and output is verified successfully.

DATE:

EXNO:6

KNAPSACK PROBLEM USING GREEDY METHOD

AIM:

To Implement Knapsack problem using Greedy Method.

ALGORITHM:

STEP 1: Declare variables weight, profit, ratio, Totalvalue, temp, capacity, and amount of appropriate types to store the weight, profit, ratio, total value, temporary values, capacity of the knapsack, and amount of a single item.

STEP 2: Prompt the user to enter the number of items n.

STEP 3: Use a loop to input the weight and profit for each item.

STEP 4: Prompt the user to enter the capacity of the knapsack.

STEP 5: Calculate the profit-to-weight ratio for each item.

STEP 6: Sort the items in non-increasing order of their profit-to-weight ratios using a bubble sort or similar technique, while also rearranging their weights and profits accordingly.

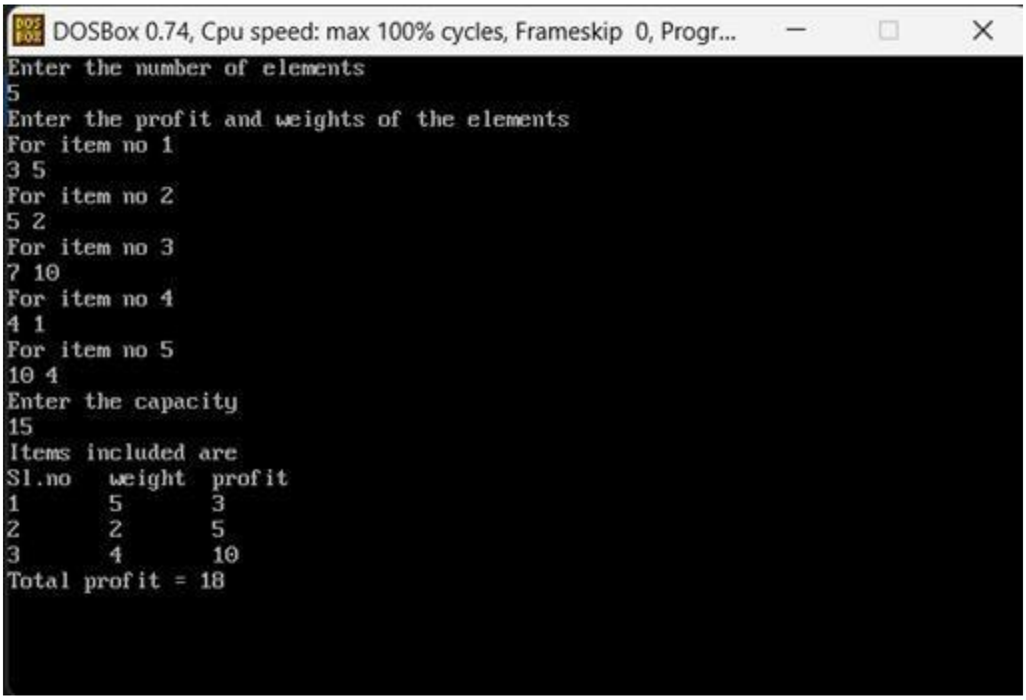
STEP 7: End Program

PROGRAM:

```
#include<iostream.h>
#include<conio.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]);
}

int main() {
   int profit, count = 0;

clrscr();
cout<< "Enter the number of elements" <<endl;
cin>> n;
cout<< "Enter the profit and weights of the elements" <<endl;
   for (i = 1; i <= n; i++) {
          cout<< "For item no " <<i<<endl;
          cin>> p[i] >> w[i];
   }
cout<< "Enter the capacity" <<endl;
cin>> cap;
   for (i = 0; i <= n; i++)
          for (j = 0; j <= cap; j++)
             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--;
    }

cout<< "Items included are" <<endl;
cout<< "Sl.no\tweight\tprofit" <<endl;
    for (i = 1; i <= n; i++)
            if (x[i])
            cout<< ++count << "\t" << w[i] << "\t" << p[i] <<endl;
cout<< "Total profit = " << profit <<endl;
getch();
    return 0;
}
```

OUTPUT:



RESULT:

Thus,the program was executed and output is verified successfully.

DATE:

EXNO:7

SHORTEST PATH USING DIJKSTRA'S ALGORITHM

AIM:

To find Shortest Path to other vertices using Dijkstra's Algorithm using c++.

ALGORITHM:

STEP1:Start

STEP2: Initialize arrays and variables to keep track of visited vertices and shortest distances

STEP3: Update the shortest distances iteratively by selecting the vertex with the minimum distance from the source vertex and relaxing its adjacent vertices.

STEP4: Take input for the number of nodes, the cost matrix representing the weighted graph, and the source vertex.

STEP5: Print the shortest paths .

STEP6:compute the shortest paths from the source vertex to all other.

STEP7 :shortest paths from the source vertex to all other vertices, along with their respective costs,
are printed

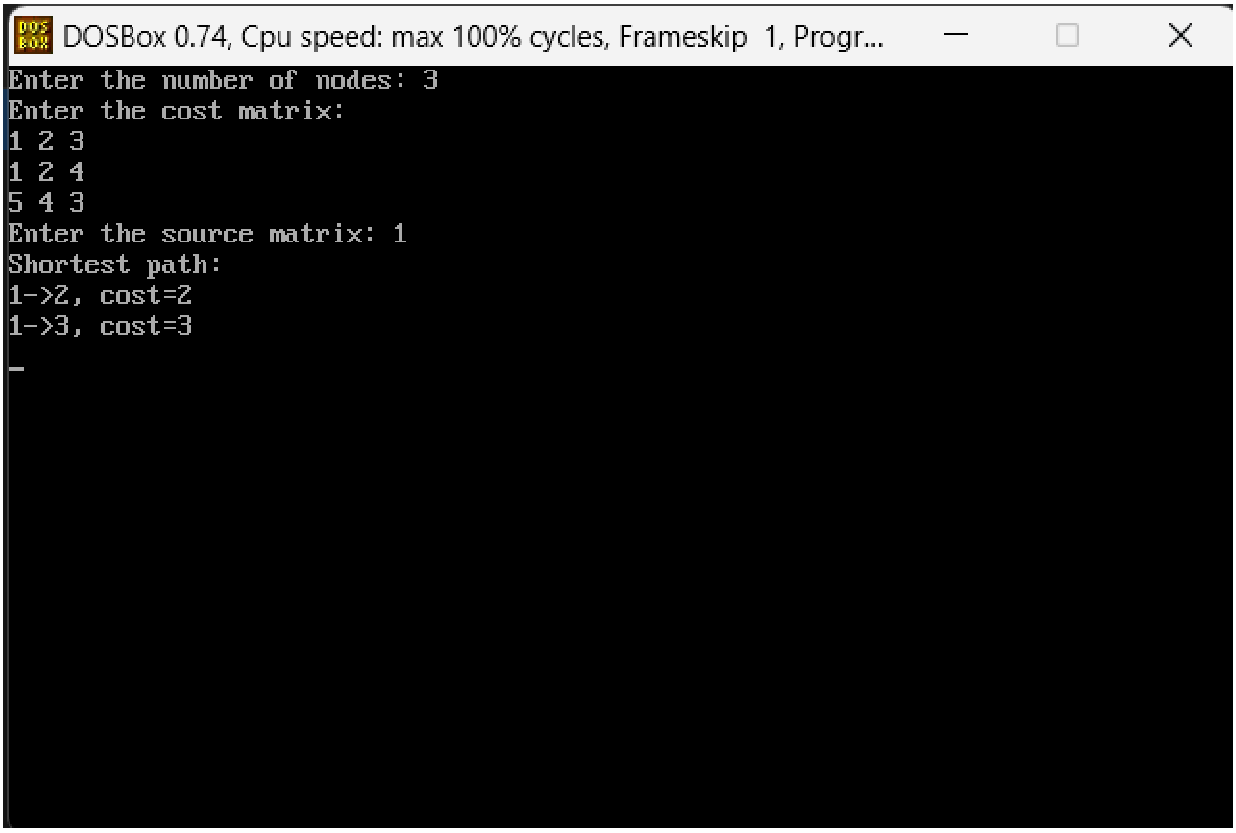
STEP8: End

PROGRAM:

```
#include <iostream.h>
#define infinity 999
#include <conio.h>

void dij(int n, int v, int cost[10][10], int dist[10]) {
   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 = infinity;
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];                          }
   }
}

int main() {
   int n, v, i, j, cost[10][10], dist[10];
cout<< "Enter the number of nodes: ";
cin>> n;
cout<< "Enter the cost matrix:\n";
   for (i = 1; i<= n; i++)
   {
          for (j = 1; j <= n; j++) {
          cin>>cost[i][j];

             if(cost[i][j] == 0){
                    cost[i][j] = infinity;
                    }
      }
          }
cout<< "Enter the source matrix: ";
cin>>v;
dij(n, v, cost, dist);
cout<< "Shortest path:\n";
   for (i = 1; i<= n; i++)
   {
          if (i != v)
          {
          cout<< v << "->" <<i<< ", cost=" <<dist[i] <<endl;
```

```
            }
              }


  return 0;
getch();
clrscr();
   }
```

OUTPUT:



RESULT:
Thus,the program was executed and output is verified successfully.

DATE:

EXNO : 8

MINIMUM LOST SPANNING TREE USING KRUSKAL'S ALGORITHM

AIM:

To find the minimum cost spanning tree of a given undirected graph using kruskal's algorithm.

ALGORITHM:

STEP1:Start

STEP2: Sort the edges by their weights in non-decreasing order.

STEP3: Initialize an empty graph as the minimum spanning tree.

STEP4: Iterate through the sorted edges.For each edge, if adding it to the spanning tree doesn't form a cycle, include it in the tree.

STEP5: Return the minimum spanning tree

STEP6: Input the number of vertices and the cost adjacency matrix.

STEP7: Repeat step 2 until (V - 1) edges are included in the spanning tree.

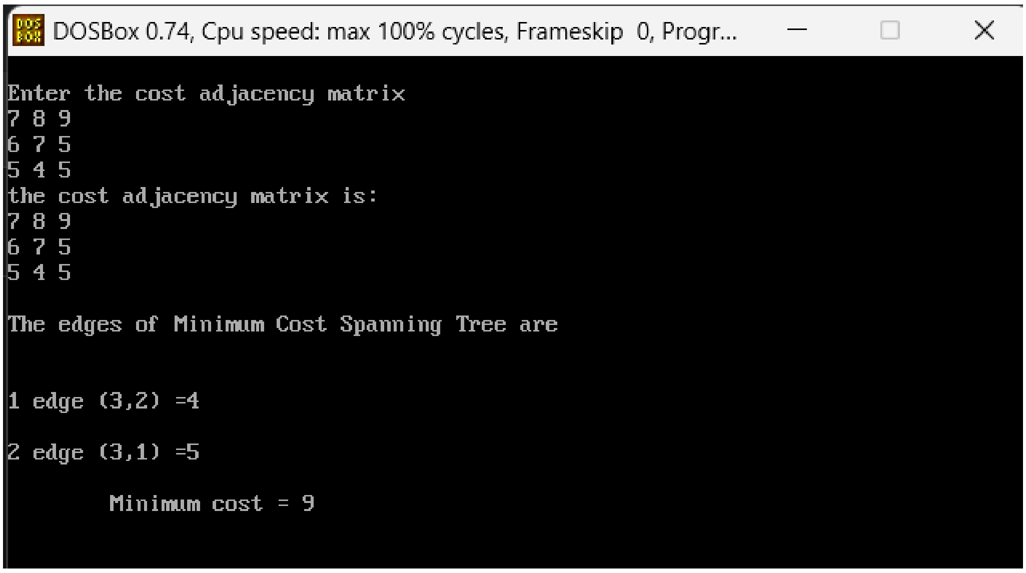
STEP8: End

PROGRAM:

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

int i, j, k, a, b, u, v, n, ne = 1;
 int mincost = 0, min;
int cost[9][9], parent[9];

int find(int);

int uni(int, int);

int main() {
cout<< "\n\n\tImplementation of Kruskal's algorithm\n\n";
cout<< "\nEnter the no. of vertices\n";
cin>> n;
cout<< "\nEnter the cost adjacency matrix\n";
   for (i = 1; i<= n; i++) {
            for (j = 1; j <= n; j++) {
            cin>> cost[i][j];
if (cost[i][j] == 0)
                        cost[i][j] = 999;
            }
    }
cout<<"the cost adjacency matrix is:\n";
    for(i=1;i<=n;i++)
    {
    for(j=1;j<=n;j++)
    {
cout<<cost[i][j]<<"  ";
    }
cout<<endl;
    }
cout<< "\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);
         if (uni(u, v)) {
cout<< "\n" << ne << " edge (" << a << "," << b << ") =" << min <<endl;
   ne++;
            mincost += min;
            }
```

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



    }
cout<< "\n\tMinimum cost = " <<mincost<<endl;
    return 0;
}



 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:



RESULT:

Thus,the program was executed and output is verified successfully.

DATE:

EXNO:9

N-QUEEN’S PROBLEM USING BACKTRACKING

AIM:

To Implement N Queen’s problem using Backtracking.

ALGORITHM:

STEP 1: Start with the first queen and set its position to 0.

STEP 2: Increment the current queen's position.

STEP 3: Check if the current position is valid (no conflicts with other queens).

STEP 4: If valid and the last queen, print the solution. If not the last queen, move to the next queen.

STEP 5: If the current position is not valid, backtrack to the previous queen.

STEP 6: Continue the process until all queens are placed or all possibilities are exhausted.

STEP 7: Output the total number of solutions found.

STEP 8: Stop

PROGRAM

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

int a[30], count=0;

int place(int pos, int n) {
    for(int 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++;
    cout << "\n\nSolution #" << count << ":\n";
    for(i=1;i<=n;i++) {
        for(j=1;j<=n;j++) {
            if(a[i]==j)
                cout << "Q\t";
            else
                cout << "*\t";
        }
        cout << endl;
    }
}

void queen(int n) {
    int k=1;
    a[k]=0;
    while(k!=0) {
        a[k]=a[k]+1;
        while((a[k]<=n) && !place(k, n))
            a[k]++;
        if(a[k]<=n) {
            if(k==n)
                print_sol(n);
            else {
                k++;
                a[k]=0;
            }
```

```
    }
        else
            k--;
    }
}

int main() {
    int n;
    clrscr();
    cout << "Enter the number of Queens\n";
    cin >> n;
    queen(n);
    cout << "\nTotal solutions=" << count;
    getch();
    return 0;
}
```

OUTPUT:

```
Enter the number of Queens
4


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


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

Total solutions=2
```

RESULT:

Thus,the program was executed and output is verified successfully.