**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**



**LAB REPORT**

**on**

**Analysis and Design of Algorithms**

***Submitted by***

**AADISHWAR RAMESH (1BM21CS002)**

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**May-2023 to July-2023**

**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**



**CERTIFICATE**

This is to certify that the Lab work entitled “**Analysis and Design of Algorithms**” carried out by

**AADISHWAR RAMESH (1BM21CS002),**who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester May-2023 to July-2023. The Lab report has been approved as it satisfies the academic requirements in respect of **Analysis and Design of Algorithms (22CS4PCADA)** work prescribed for the said degree.

Name of the Lab-In charge:

Sneha S Bagalkot Dr. Jyothi S Nayak

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

|  |  |  |
| --- | --- | --- |
| **Lab Program No.** | **Program Details** | **Page No.** |
| 1 | Write program to do the following:  a. Print all the nodes reachable from a given starting node in a digraph using BFS method.  b. Check whether a given graph is connected or not using DFS method. | 1 - 6 |
| 2 | Write program to obtain the Topological ordering of vertices in a given digraph. | 7 - 9 |
| 3 | Implement Johnson Trotter algorithm to generate permutations. | 10 - 14 |
| 4 | Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort. | 15 - 18 |
| 5 | Sort a given set of N integer elements using Quick Sort technique and compute its time taken. | 19 - 21 |
| 6 | Sort a given set of N integer elements using Heap Sort technique and compute its time taken. | 22 - 25 |
| 7 | Implement 0/1 Knapsack problem using dynamic programming. | 26 - 28 |
| 8 | Implement All Pair Shortest paths problem using Floyd’s algorithm. | 29 - 30 |
| 9 | Find Minimum Cost Spanning Tree of a given undirected graph using Prim’s and Kruskal’s algorithm. | 31 - 36 |
| 10 | From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra’s algorithm. | 37 - 39 |
| 11 | Implement “N-Queens Problem” using Backtracking. | 40 - 43 |

**Course Outcome**

|  |  |
| --- | --- |
| CO1 | Analyze time complexity of Recursive and Non-recursive algorithms using asymptotic notations. |
| CO2 | Apply various design techniques for the given problem. |
| CO3 | Apply the knowledge of complexity classes P, NP, and NP-Complete and prove certain  problems are NP-Complete |
| CO4 | Design efficient algorithms and conduct practical experiments to solve problems. |

**1. a) Breadth First Search**

**Aim:** To print all the nodes reachable from a given starting node in a digraph using BFS method

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

int main(void)

{

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

int n;

scanf("%d", &n);

int i, j;

int \*\*adjMatrix = (int \*\*)malloc(n \* sizeof(int \*));

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

{

adjMatrix[i] = (int \*)malloc(n \* sizeof(int));

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

{

adjMatrix[i][j] = 0;

}

}

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

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

{

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

{

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

}

}

printf("Enter the starting vertex: ");

int src;

scanf("%d", &src);

printf("Breadth First Traversal is as (starting from vertex %d):\n", src);

bool visited[n];

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

{

visited[i] = false;

}

int queue[n];

int front = 0, rear = 0;

visited[src] = true;

queue[rear++] = src;

while (front != rear)

{

int currentVertex = queue[front++];

printf("%d ", currentVertex);

for (int adjacent = 0; adjacent < n; adjacent++)

{

if (adjMatrix[currentVertex][adjacent] && !visited[adjacent])

{

visited[adjacent] = true;

queue[rear++] = adjacent;

}

}

}

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

{

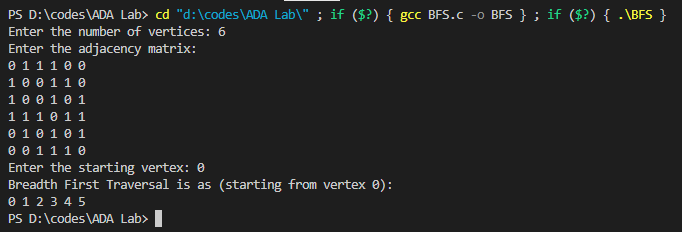
free(adjMatrix[i]);

}

free(adjMatrix);

}

**Output:**



**1. b) Depth First Search**

**Aim:** To check whether a given graph is connected or not using DFS method

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void DFS(int vertex, int \*\*adjMatrix, bool \*visited, int n)

{

printf("%d ", vertex);

visited[vertex] = true;

for (int adjacent = 0; adjacent < n; adjacent++)

{

if (adjMatrix[vertex][adjacent] && !visited[adjacent])

{

DFS(adjacent, adjMatrix, visited, n);

}

}

}

int main(void)

{

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

int n;

scanf("%d", &n);

int i, j;

int \*\*adjMatrix = (int \*\*)malloc(n \* sizeof(int \*));

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

{

adjMatrix[i] = (int \*)malloc(n \* sizeof(int));

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

{

adjMatrix[i][j] = 0;

}

}

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

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

{

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

{

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

}

}

printf("Enter the starting vertex: ");

int src;

scanf("%d", &src);

printf("Depth First Traversal is as (starting from vertex %d):\n", src);

bool visited[n];

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

{

visited[i] = false;

}

DFS(src, adjMatrix, visited, n);

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

{

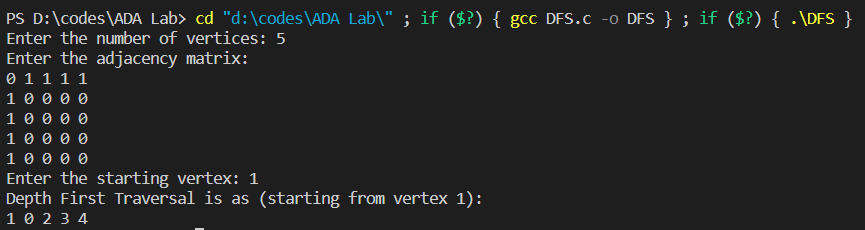
free(adjMatrix[i]);

}

free(adjMatrix);

}

**Output:**



**2. Topological Sorting**

**Aim:** To obtain the Topological ordering of vertices in a given digraph

**Code:**

#include <stdio.h>

int main()

{

int n;

printf("Enter the no of vertices: ");

scanf("%d", &n);

int a[n][n], indeg[n], flag[n];

int i, j, k, count = 0;

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

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

{

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

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

}

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

{

indeg[i] = 0;

flag[i] = 0;

}

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

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

indeg[i] = indeg[i] + a[j][i];

printf("\nThe topological order is: ");

while (count < n)

{

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

{

if ((indeg[k] == 0) && (flag[k] == 0))

{

printf("%d ", (k + 1));

flag[k] = 1;

}

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

{

if (a[i][k] == 1)

indeg[k]--;

}

}

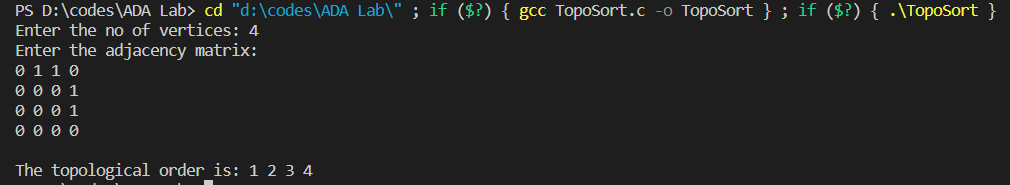
count++;

}

return 0;

}

**Output:**



**3. Johnson Trotter algorithm**

**Aim:** To generate permutations of n numbers using Johnson Trotter algorithm

**Code:**

#include <stdio.h>

#include <stdbool.h>

bool LR = true;

bool RL = false;

int search(int a[], int n, int mobile)

{

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

{

if (a[i] == mobile)

{

return i + 1;

}

}

}

int getMobile(int a[], bool dir[], int n)

{

int i;

int prev = 0, mobile = 0;

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

{

if (dir[a[i] - 1] == RL && i != 0)

{

if (a[i] > a[i - 1] && a[i] > prev)

{

mobile = a[i];

prev = mobile;

}

}

if (dir[a[i] - 1] == LR && i != n - 1)

{

if (a[i] > a[i + 1] && a[i] > prev)

{

mobile = a[i];

prev = mobile;

}

}

}

if (mobile == 0 && prev == 0)

return 0;

else

return mobile;

}

int Perm(int a[], bool dir[], int n)

{

int temp;

int mobile = getMobile(a, dir, n);

int pos = search(a, n, mobile);

if (dir[a[pos - 1] - 1] == RL)

{

temp = a[pos - 1];

a[pos - 1] = a[pos - 2];

a[pos - 2] = temp;

}

else if (dir[a[pos - 1] - 1] == LR)

{

temp = a[pos];

a[pos] = a[pos - 1];

a[pos - 1] = temp;

}

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

{

if (a[i] > mobile)

{

if (dir[a[i] - 1] == LR)

dir[a[i] - 1] = RL;

else if (dir[a[i] - 1] == RL)

dir[a[i] - 1] = LR;

}

}

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

{

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

}

printf(" ");

}

int fact(int n)

{

int fact = 1;

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

{

fact = fact \* i;

}

return fact;

}

void perms(int n)

{

int a[n];

bool dir[n];

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

{

a[i] = i + 1;

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

}

printf("\n");

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

dir[i] = RL;

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

Perm(a, dir, n);

}

int main(void)

{

int n;

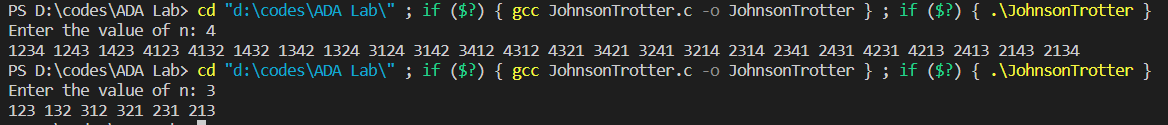
printf("Enter the value of n: ");

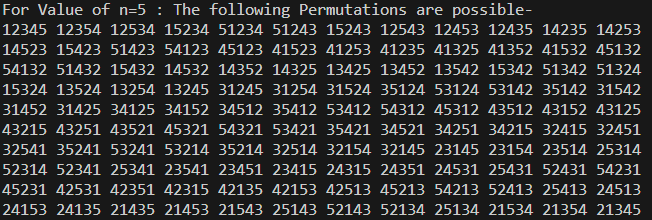
scanf("%d", &n);

perms(n);

}

**Output:**





**4. Merge Sort**

**Aim:** To sort a given set of N integer elements using Merge Sort technique, compute its time taken for different values of N and record the time taken to sort

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void merge(int arr[], int p, int q, int r)

{

int n1 = q - p + 1;

int n2 = r - q;

int L[n1], M[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[p + i];

for (int j = 0; j < n2; j++)

M[j] = arr[q + 1 + j];

int i, j, k;

i = 0;

j = 0;

k = p;

while (i < n1 && j < n2)

{

if (L[i] <= M[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = M[j];

j++;

}

k++;

}

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

while (j < n2)

{

arr[k] = M[j];

j++;

k++;

}

}

void mergeSort(int arr[], int l, int r)

{

if (l < r)

{

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

int main(void)

{

int n;

printf("Enter the no of elements: ");

scanf("%d", &n);

int arr[n];

// printf("Enter the elements: ");

srand(time(0));

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

{

arr[i] = rand();

}

clock\_t st, end;

st = clock();

mergeSort(arr, 0, n - 1);

end = clock();

double time\_taken = (((double)(end - st)) / CLOCKS\_PER\_SEC);

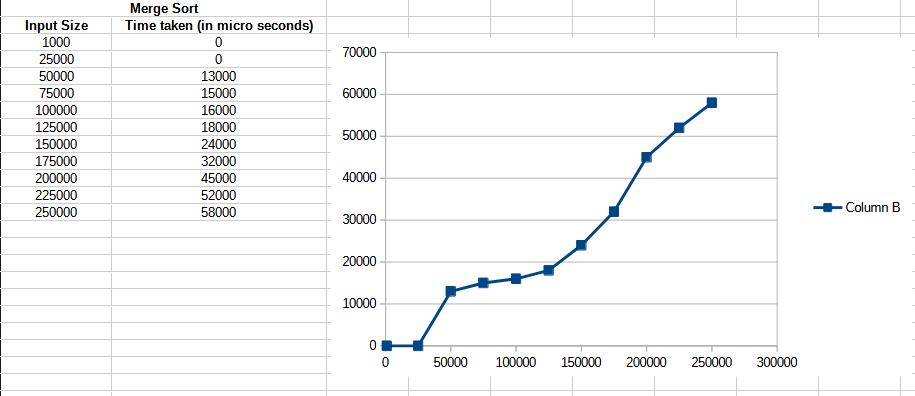
printf("\nSorted array: ");

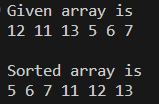
for (int i = 0; i < n; i++)

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

printf("\nTime taken: %lf micro seconds\n", time\_taken \* 1000000);

}

**Output with input size vs time graph:**



**5. Quick Sort**

**Aim:** To sort a given set of N integer elements using Quick Sort technique and compute its time taken

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition(int arr[], int low, int high)

{

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++)

{

if (arr[j] < pivot)

{

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

void quickSort(int arr[], int low, int high)

{

if (low < high)

{

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main(void)

{

int n;

printf("Enter the no of elements: ");

scanf("%d", &n);

int arr[n];

// printf("Enter the elements: ");

srand(time(0));

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

{

arr[i] = rand();

}

clock\_t st, end;

st = clock();

quickSort(arr, 0, n - 1);

end = clock();

double time\_taken = (((double)(end - st)) / CLOCKS\_PER\_SEC);

printf("\nSorted array: ");

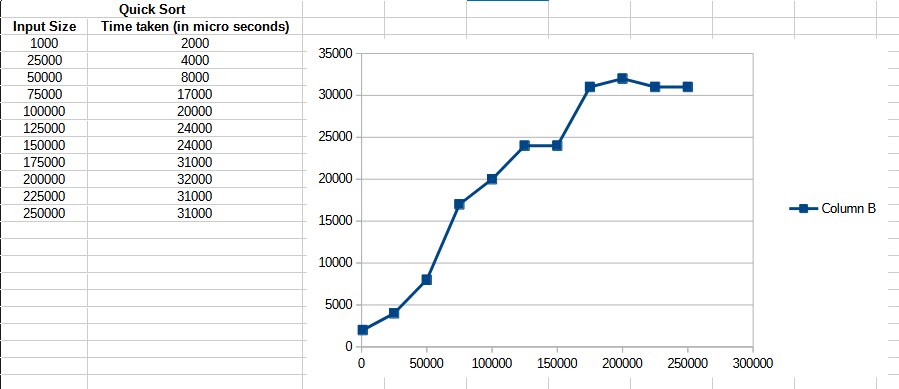
for (int i = 0; i < n; i++)

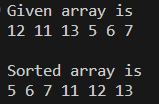
printf("%d ", arr[i]);

printf("\nTime taken: %lf micro seconds\n", time\_taken \* 1000000);

}

**Output with input size vs time graph:**



****

**6. Heap Sort**

**Aim:** To sort a given set of N integer elements using Heap Sort technique and compute its time taken

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void swap(int \*a, int \*b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void heapify(int arr[], int N, int i)

{

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < N && arr[left] > arr[largest])

{

largest = left;

}

if (right < N && arr[right] > arr[largest])

{

largest = right;

}

if (largest != i)

{

swap(&arr[i], &arr[largest]);

heapify(arr, N, largest);

}

}

void heapSort(int arr[], int N)

{

for (int i = N / 2 - 1; i >= 0; i--)

{

heapify(arr, N, i);

}

for (int i = N - 1; i >= 0; i--)

{

swap(&arr[0], &arr[i]);

heapify(arr, i, 0);

}

}

int main(void)

{

int n;

printf("Enter the size of array: ");

scanf("%d", &n);

int arr[n];

// printf("Enter the elements: ");

srand(time(0));

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

{

arr[i] = rand();

}

clock\_t st, end;

st = clock();

heapSort(arr, n);

end = clock();

double time\_taken = (((double)(end - st)) / CLOCKS\_PER\_SEC);

printf("\nSorted array: ");

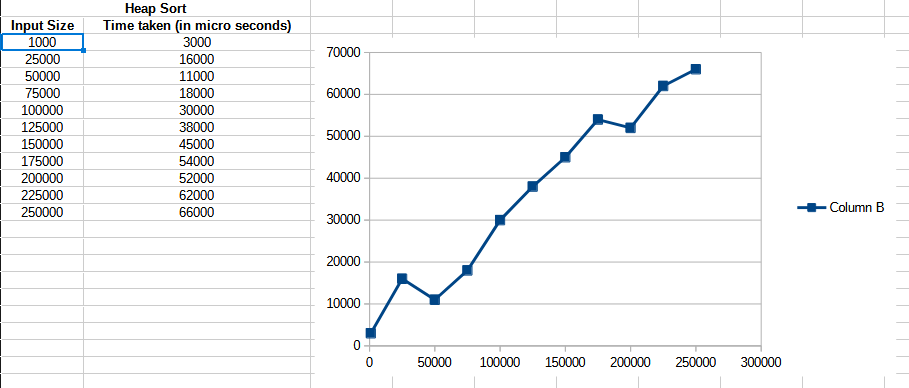
for (int i = 0; i < n; i++)

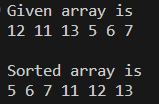
printf("%d ", arr[i]);

printf("\nTime taken: %lf micro seconds\n", time\_taken \* 1000000);

}

**Output with input size vs time graph:**





**7. 0/1 Knapsack Problem**

**Aim:** To optimize(maximize) the items in the knapsack for our requirement using 0/1 Knapsack algorithm

**Code:**

[

#include <stdio.h>

int main(void)

{

printf("Enter the number of items: ");

int n;

scanf("%d", &n);

printf("Enter the price of each item: ");

int price[n];

int i;

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

{

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

}

printf("Enter the weight of each item: ");

int weight[n];

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

{

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

}

printf("Enter the max weight: ");

int W;

scanf("%d", &W);

printf("\nThe dp table is:\n");

int dp[n + 1][W + 1];

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

{

for (int j = 0; j <= W; j++)

{

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

{

dp[i][j] = 0;

}

else if (weight[i - 1] <= j)

{

dp[i][j] = (price[i - 1] + dp[i - 1][j - weight[i - 1]]) > dp[i - 1][j] ? (price[i - 1] + dp[i - 1][j - weight[i - 1]]) : dp[i - 1][j];

}

else

{

dp[i][j] = dp[i - 1][j];

}

printf("%d ", dp[i][j]);

}

printf("\n");

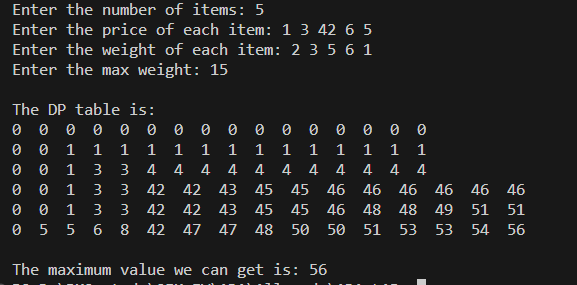
}

printf("\nThe maximum value we can get is: %d", dp[n][W]);

return 0;

}

**Output:**



**8. Floyd’s Algorithm**

**Aim:** To find out the shortest path between all pairs of vertices

**Code:**

#include <stdio.h>

int main(void)

{

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

int n;

scanf("%d", &n);

printf("Enter the adjacency matrix(use 999 as infinity):\n");

int adj[n][n];

int i, j, k;

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

{

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

{

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

}

}

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

{

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

{

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

{

if (adj[i][j] > adj[i][k] + adj[k][j])

{

adj[i][j] = adj[i][k] + adj[k][j];

}

}

}

}

printf("The shortest path matrix is:\n");

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

{

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

{

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

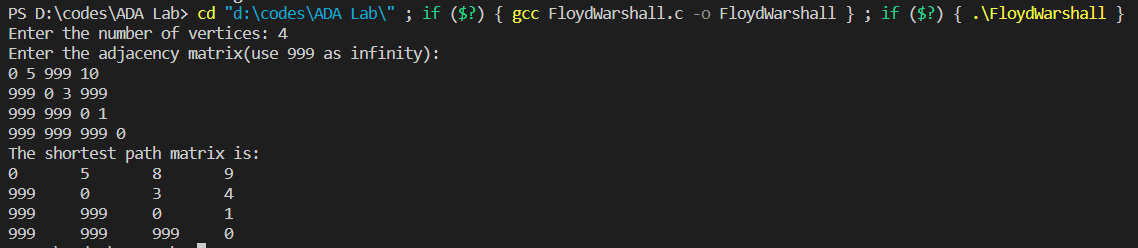
}

printf("\n");

}

}

**Output:**



**9. Prim’s and Kruskal’s algorithm**

**Aim:** To find minimal spanning tree of a graph using Prim’s and Kruskal’s algorithms

**Prim’s Algorithm Code:**

#include <stdio.h>

int main(void)

{

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

int n;

scanf("%d", &n);

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

int adj[n][n];

int i, j, k;

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

{

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

{

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

}

}

int visited[n];

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

{

visited[i] = 0;

}

printf("Enter the starting vertex: ");

int start;

scanf("%d", &start);

visited[start] = 1;

printf("\nThe minimal spanning tree is:\nEdge : Weight\n");

for (k = 0; k < n - 1; k++)

{

int min = 999;

int u = 0;

int v = 0;

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

{

if (visited[i])

{

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

{

if (!visited[j] && adj[i][j])

{

if (min > adj[i][j])

{

min = adj[i][j];

u = i;

v = j;

}

}

}

}

}

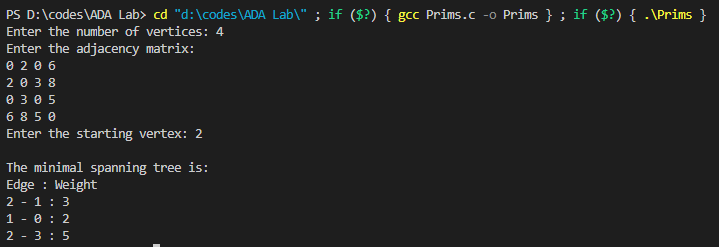
printf("%d - %d : %d\n", u, v, adj[u][v]);

visited[v] = 1;

}

}

**Output:**



**Kruskal’s Algorithm Code:**

#include <stdio.h>

int find(int v, int \*parent)

{

while (parent[v] != v)

{

v = parent[v];

}

return v;

}

void union1(int i, int j, int \*parent)

{

if (i < j)

parent[j] = i;

else

parent[i] = j;

}

int main(void)

{

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

int n;

scanf("%d", &n);

printf("Enter the adjacency matrix(use 999 as infinity):\n");

int adj[n][n];

int i;

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

{

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

{

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

}

}

int parent[n];

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

{

parent[i] = i;

}

int count = 0, k = 0, min, sum = 0, j, t[n][n], u, v;

while (count != n - 1)

{

min = 999;

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

{

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

{

if (adj[i][j] < min && adj[i][j] != 0)

{

min = adj[i][j];

u = i;

v = j;

}

}

}

i = find(u, parent);

j = find(v, parent);

if (i != j)

{

union1(i, j, parent);

t[k][0] = u;

t[k][1] = v;

k++;

count++;

sum = sum + adj[u][v];

}

adj[u][v] = adj[v][u] = 999;

}

if (count == n - 1)

{

printf("The minimal spanning tree is as:\n");

for (i = 0; i < n - 1; i++)

{

printf("%d -> %d\n", t[i][0], t[i][1]);

}

printf("Cost of spanning tree = %d\n", sum);

}

else

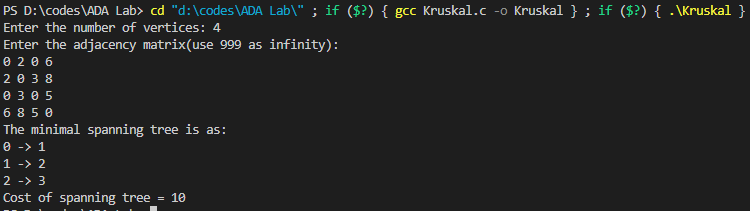
{

printf("\nSpanning tree does not exist!");

}

}

**Output:**



**10. Dijkstra’s Algorithm**

**Aim:** To find shortest paths to other vertices from a given vertex in a weighted connected graph using Dijkstra’s algorithm

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int main(void)

{

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

int n;

scanf("%d", &n);

int \*\*arr = (int \*\*)malloc(n \* sizeof(int \*));

int i, j;

printf("Enter cost matrix(use 999 for infinity):\n");

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

{

arr[i] = (int \*)malloc(n \* sizeof(int));

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

{

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

}

}

printf("Enter the source vertex: ");

int src;

scanf("%d", &src);

int dist[n];

int visited[n];

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

{

dist[i] = INT\_MAX;

visited[i] = 0;

}

dist[src] = 0;

for (int count = 0; count < n - 1; count++)

{

int min = INT\_MAX, min\_index;

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

{

if (!visited[i] && dist[i] <= min)

{

min = dist[i], min\_index = i;

}

}

visited[min\_index] = 1;

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

{

if (!visited[i] && arr[min\_index][i] && dist[min\_index] != INT\_MAX && dist[min\_index] + arr[min\_index][i] < dist[i])

{

dist[i] = dist[min\_index] + arr[min\_index][i];

}

}

}

printf("The shortest path from source vertex %d to all other vertices is:\n", src);

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

{

printf("%d -> %d: %d\n", src, i, dist[i]);

}

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

{

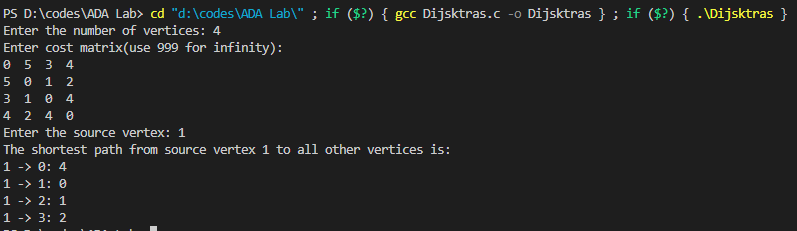
free(arr[i]);

}

free(arr);

}

**Output:**



**11. N – Queen’s Problem**

**Aim:** To calculate a solution to place N queens in an N x N chess board such that no two queens cancel each other

**Code:**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

int n;

bool isSafe(int \*\*arr, int x, int y)

{

int row, col;

for (row = 0; row < x; row++)

{

if (arr[row][y] == 1)

{

return false;

}

}

for (row = x, col = y; row >= 0 && col >= 0; row--, col--)

{

if (arr[row][col] == 1)

{

return false;

}

}

for (row = x, col = y; row >= 0 && col < n; row--, col++)

{

if (arr[row][col] == 1)

{

return false;

}

}

return true;

}

bool nQueen(int \*\*arr, int x)

{

if (x >= n)

{

return true;

}

for (int col = 0; col < n; col++)

{

if (isSafe(arr, x, col))

{

arr[x][col] = 1;

if (nQueen(arr, x + 1))

{

return true;

}

arr[x][col] = 0;

}

}

return false;

}

int main(void)

{

printf("Enter the size of board: ");

scanf("%d", &n);

int \*\*arr = (int \*\*)malloc(n \* sizeof(int \*));

int i, j;

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

{

arr[i] = (int \*)malloc(n \* sizeof(int));

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

{

arr[i][j] = 0;

}

}

if (nQueen(arr, 0))

{

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

{

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

{

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

}

printf("\n");

}

}

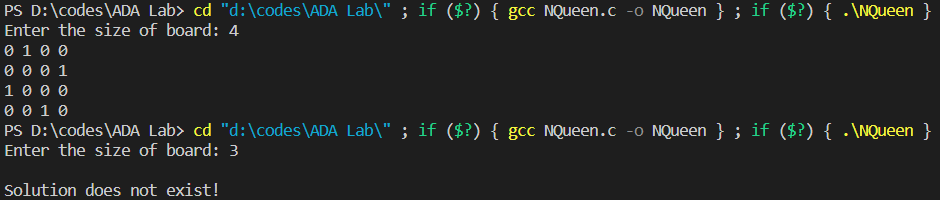
else

{

printf("\nSolution does not exist!");

}

}

**Output:**

