5AD4-23: Analysis of Algorithms Lab

Govt. Engineering College Bikaner, Bikaner

Bachelor in Technology Artificial Intelligence and Data Science

Submitted By:

Rahul verma(20EEBAD047)

Course Faculty:

Anita Chandel

INDEX Table

Experiment No. 1

Aim: To Sort a given set of elements using the quick sort method and determine the time required to sort the elements and plot a graph of time taken verses n.

Program Logic: Quick sort is a algorithm based on the divide and conqueror approach where

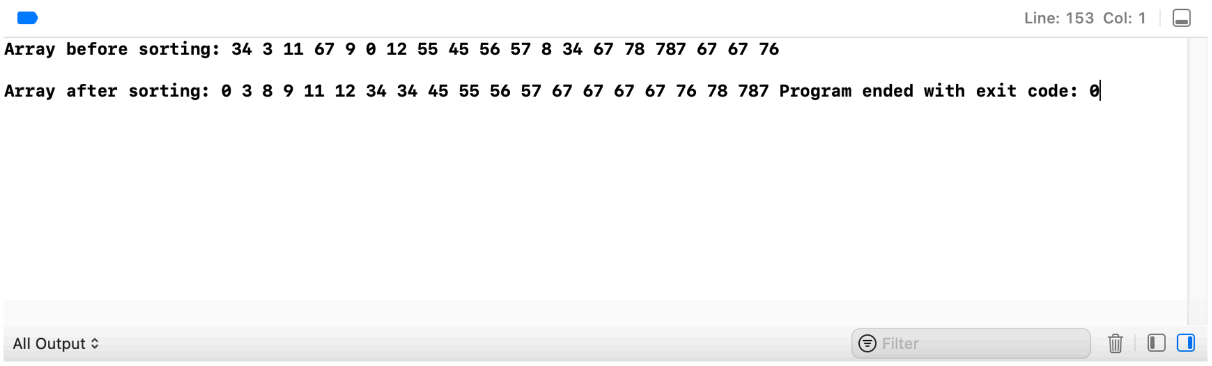
1. An array is divided into subarrays by selecting a pivot element.
2. While dividing array, pivot element should be positioned is such a way that elements less then pivot are kept on left side and elements greater than pivot are on the right side of the pivot.
3. The left and right subarrays are also divided using the same approach. This process continues until subarrays contains a single element.
4. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Source Code:

```
#include<stdio.h>

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

  int temp = *a;

  *a = *b;

  *b = temp;

}

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

  int pivot = arr[high];

  int i = low - 1;

  for(int j = low; j < high; 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); //partition
        quickSort(arr, low, pi-1); // left
        quickSort(arr, pi+1, high); //right
    }
}
void printArray(int arr[], int size){
    for(int k=0; k < size; k++){
        printf("%d ", arr[k]);
    }
}
int main(){
    int arr[] = {34, 3, 11, 67, 9, 0, 12, 55, 88, 1};
    int size = sizeof(arr) / sizeof(arr[1]);
    printf("Array before sorting: "); printArray(arr, size);
    quickSort(arr, 0, size - 1);
    printf("\n\nArray after sorting: "); printArray(arr, size);
    return 0;
}
```

Result:



```
Array before sorting: 34 3 11 67 9 0 12 55 45 56 57 8 34 67 78 787 67 67 76

Array after sorting: 0 3 8 9 11 12 34 34 45 55 56 57 67 67 67 67 76 78 787 Program ended with exit code: 0
```

Experiment No. 2

Aim: To implement a parallelised Merge sort algorithm to sort a given set of elements and determine the time required to sort the elements and plot a graph of the time taken versus n elements.

Program Logic: Merge sort works on divide and conqueror approach.

1. Divide the array half until it's reach to a single element.

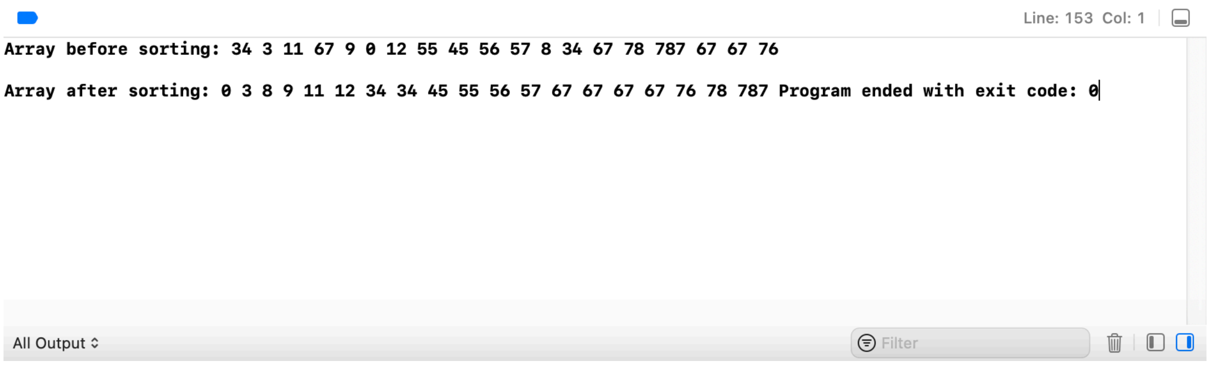
2. After dividing merge every element by comparing.

3. Combined every element until it becomes I it original size.

Source Code:

```
#include<stdio.h>

void merge(int arr[], int first, int mid, int last){

  //coping divide elements in new array left and right

  int x = mid - first + 1;

  int y = last - mid;

  int left[x], right[y];

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

    left[i] = arr[first + i];

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

    right[i] = arr[mid + 1 + i];

  //Compare elements of left with right array

  int i = 0, j = 0, k = first;

  while((i < x) && (j < y)){

    if(left[i] <= right[j]){

      arr[k] = left[i];

      i++;

    }

    else{
          arr[k] = right[j];
          j++;
        }
        k++;
    }

    //merge reaming elements of left and right array
    while(i < x){
        arr[k] = left[i];
        i++; k++;
    }
    while(j < y){
        arr[k] = right[j];
        j++; k++;
    }
}

void mergeSort(int arr[], int first, int last){
      if( first < last){
          int mid = first + (last - 1) / 2;


          mergeSort(arr, first, mid);
          mergeSort(arr, mid+1, last);


          merge(arr, first, mid, last);
      }
}
void printArray(int arr[], int size){
    for(int i=0; i < size; i++){
        printf("%d ", arr[i]);
    }
}
int main(){
    int arr[] = {67, 11, 32, 44, 1, 12, 0, 99};
    int size = sizeof(arr) / sizeof(arr[1]);
    printf("Array before sorting: "); printArray(arr, size);
    mergeSort(arr, 0, size - 1);
    printf("\n\nArray after sorting: "); printArray(arr, size);
    return 0;
}
```

Result:



```
Array before sorting: 34 3 11 67 9 0 12 55 45 56 57 8 34 67 78 787 67 67 76

Array after sorting: 0 3 8 9 11 12 34 34 45 55 56 57 67 67 67 67 76 78 787 Program ended with exit code: 0
```

Experiment No. 3

Aim: a. Obtain the Topological ordering of vertices in a given digraph. b. Compute the transitive closure of a given directed graph using Warshall's algorithm.

Program Logic:

a. Topological sorting for directed acyclic Graph(DAG) is a linear ordering of vertices such that for every discrete directed edge uv, vertex u comes before v in the ordering.

    a. We use a temporary stack.
    b. We don’t print the vertex immediately
    c. We first recursively call topological sorting for all its adjacent vertices, then push it to attack.
    d. Finally prints the content of the stack.

b. If a vertex j is reachable from another vertex I for all vertex pairs I, j in the given graph. The reach-ability matrix is called the transitive closure of a graph.

Source code:

A.

```
#include <iostream>

#include <stack>

#include <unordered_map>

#include <vector>


class Graph {

private:

        std::unordered_map<int, std::vector<int> >

                graph; // adjacency list

        int V; // number of vertices

public:

        Graph(int vertices)

                : V(vertices)
```

```
{
}

void addEdge(int u, int v) { graph[u].push_back(v); }
void nonRecursiveTopologicalSort()
{
        std::vector<bool> visited(
                V,
                false); // mark all the vertices as not visited
        std::stack<int> stack; // result stack
        for (int i = 0; i < V; i++) {
                if (!visited[i]) {
                        nonRecursiveTopologicalSortUtil(i, visited,
                                                                                                                stack);
                }
        }

        std::vector<int> sorted;
        while (!stack.empty()) {
                sorted.push_back(stack.top());
                stack.pop();
        }
        std::cout << "The following is a Topological Sort "
                                "of the given graph:\n";
        for (auto& i : sorted) {
                std::cout << i << " ";
        }
        std::cout << std::endl;
}
```

```
private:
	void nonRecursiveTopologicalSortUtil(
		int v, std::vector<bool>& visited,
		std::stack<int>& stack)
	{
		visited[v] = true;

		for (auto& next_neighbor : graph[v]) {
			if (!visited[next_neighbor]) {
				nonRecursiveTopologicalSortUtil(
					next_neighbor, visited, stack);
			}
		}
		stack.push(v);
	}
};

int main()
{
	Graph g(6);
	g.addEdge(5, 2);
	g.addEdge(5, 0);
	g.addEdge(4, 0);
	g.addEdge(4, 1);
	g.addEdge(2, 3);
	g.addEdge(3, 1);

	g.nonRecursiveTopologicalSort();
```

```
        return 0;
}
```

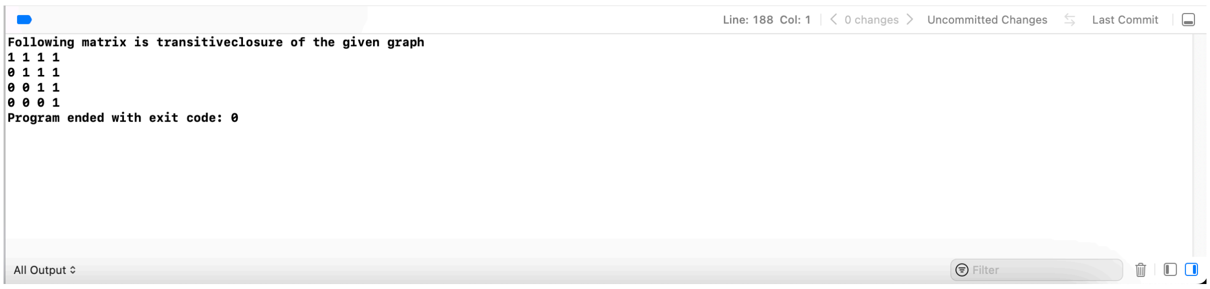
B.

```
#include<stdio.h>
#define V 4
// using Floyd Warshall algorithm
void transitiveClosure(int graph[][V])
{
        int reach[V][V], i, j, k;
        for (i = 0; i < V; i++)
                for (j = 0; j < V; j++)
                        reach[i][j] = graph[i][j];
        for (k = 0; k < V; k++)
        {
                for (i = 0; i < V; i++)
                {
                        for (j = 0; j < V; j++)
                        {
                                reach[i][j] = reach[i][j] ||
                                (reach[i][k] && reach[k][j]);
                        }
                }
        }
        printSolution(reach);
}
void printSolution(int reach[][V])
{
        printf ("Following matrix is transitive");
```

```
    printf("closure of the given graph\n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if(i == j)
                printf("1 ");
            else
                printf ("%d ", reach[i][j]);
        }
        printf("\n");
    }
}
int main()
{
    int graph[V][V] = { {1, 1, 0, 1},
                        {0, 1, 1, 0},
                        {0, 0, 1, 1},
                        {0, 0, 0, 1}
                      };
    transitiveClosure(graph);
    return 0;
}
```

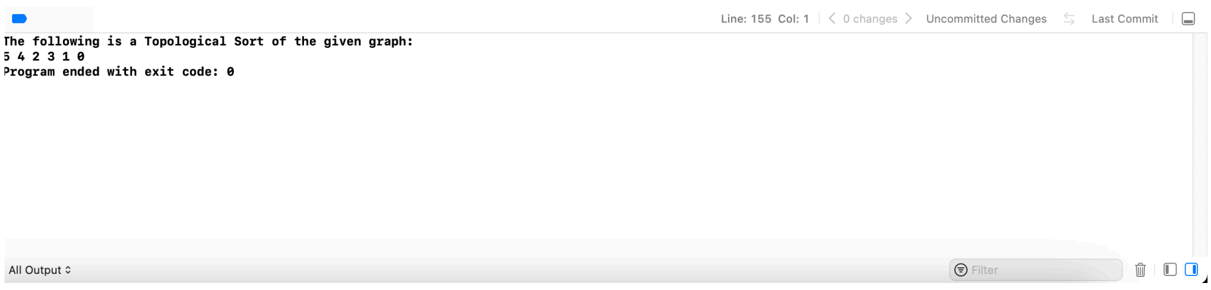
Results:

A.



```
Following matrix is transitiveclosure of the given graph
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1
Program ended with exit code: 0
```

B.



```
The following is a Topological Sort of the given graph:
5 4 2 3 1 0
Program ended with exit code: 0
```

Experiment No. 4

Aim: Implement 0/1 Knapsack problem using Dynamic Programming.

Program Logic:

a. Maximum value obtained by N items and W weigh.

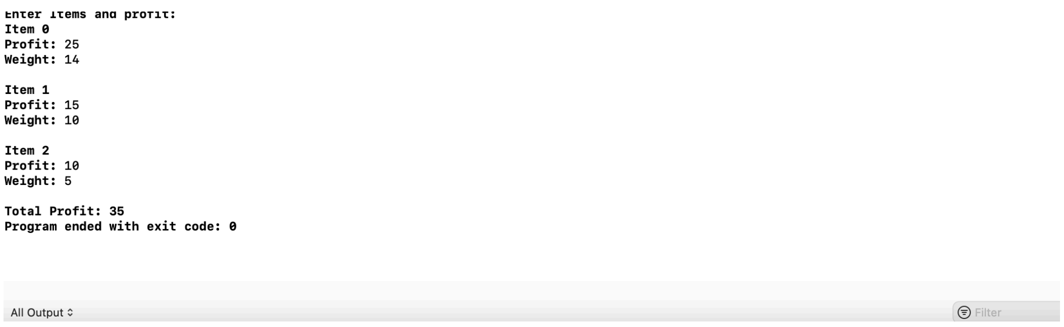
b. Value of nth item plus maximum value obtained by N-1 items minus weight of that.

c. If the weight of Nth item is greater than W then the Nth cannot be included.

Source code:

```
#include<iostream>

void maximizeProfit(int profit, int weight, int *knapsack, int *capacity, int *ptr){
        if((*capacity - weight) > 0){
                *knapsack = profit;
                *capacity = *capacity - weight;
                *ptr = *ptr + 1;
        }
}

int calculateProfit(int knapsack[], int ptr){
        int temp=0;
        for(int i=0; i<ptr; i++){
                temp = temp + knapsack[i];
        }
        return temp;
}

int main(){
}

        int capacity;

        std::cout << "Enter Knapsack capacity: "; std::cin >> capacity;


        int length;

        std::cout << "\nEnter Array size: ";

        std::cin >> length;


        int profit[length], weight[length], knapsack[length];

        std::cout << "\nEnter Items and profit:";


        for(int i=0; i < length; i++){

                std::cout << "\nItem " << i << "\nProfit: "; std::cin >> profit[i];

                std::cout << "Weight: "; std::cin >> weight[i];

        }


        //Assuming profit is sorted in decending order

        //max heap

        int ptr = 0;

        for(int i=0; i < length; i++){

                maximizeProfit(profit[i], weight[i], &knapsack[ptr], &capacity, &ptr);

        }


        int totalProfit = calculateProfit(knapsack, ptr);


        std::cout << "\nTotal Profit: " << totalProfit << std::endl;

   return 0;

}
```

Results:



```
Enter items and profit:
Item 0
Profit: 25
Weight: 14

Item 1
Profit: 15
Weight: 10

Item 2
Profit: 10
Weight: 5

Total Profit: 35
Program ended with exit code: 0
```

Experiment No. 5

Aim: From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

Program Logic: The Dijkstra’s algorithm works on the basis that if there is no direct path Available from vertex i to j then there exist a path from i to k and k to j

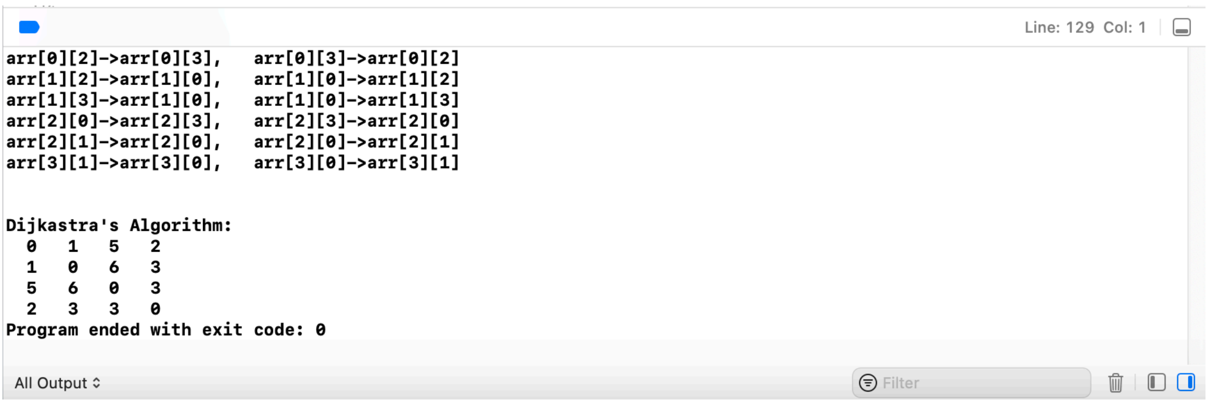
Or

If there is long(more coast) path from vertex I to j then there exists a path from I to k and k to j which is shorter.

Source code:

```
#include<stdio.h>

void dijkastrs(int col, int arr[][col], int size){

   for(int i = 0; i < size; i++){

      for(int j = 0; j < size; j++){

         for(int k = 0; k < size; k++){

            if(arr[i][j] > (arr[i][k] + arr[k][j])){

               printf("arr[%d][%d]->arr[%d][%d],   arr[%d][%d]->arr[%d][%d]\n", i, j, i, k, i, k, i, j);

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

            }

         }

      }

   }

}


void printArray(int col, int arr[][col], int size){

   printf("\n\nDijkastra's Algorithm: \n");

   for(int i = 0; i < size; i++){

      for(int j = 0; j < size; j++){
            printf("%3d ", arr[i][j]);
        }
        printf("\n");
    }
}

int main(){
    int arr[][4] = {{0, 1, 999, 2},
                    {1, 0, 999, 999},
                    {999, 999, 0, 3},
                    {2, 999, 3, 0}};

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

    dijkastrs(size, arr, size);

    printArray(size, arr, size);

    return 0;
}
```

Results:



```
arr[0][2]->arr[0][3],   arr[0][3]->arr[0][2]
arr[1][2]->arr[1][0],   arr[1][0]->arr[1][2]
arr[1][3]->arr[1][0],   arr[1][0]->arr[1][3]
arr[2][0]->arr[2][3],   arr[2][3]->arr[2][0]
arr[2][1]->arr[2][0],   arr[2][0]->arr[2][1]
arr[3][1]->arr[3][0],   arr[3][0]->arr[3][1]


Dijkastra's Algorithm:
  0   1   5   2
  1   0   6   3
  5   6   0   3
  2   3   3   0
Program ended with exit code: 0
```

Experiment No. 6

Aim: Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

Program Logic:

1. Sort all the edges from low weight to high.
2. Take the edge withe lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edge until we reach all vertices.

Source code:

```
#include <stdio.h>
#define MAX 30

typedef struct edge {
  int u, v, w;
} edge;

typedef struct edge_list {
  edge data[MAX];
  int n;
} edge_list;

edge_list elist;

int Graph[MAX][MAX], n;
edge_list spanlist;

void kruskalAlgo();
```

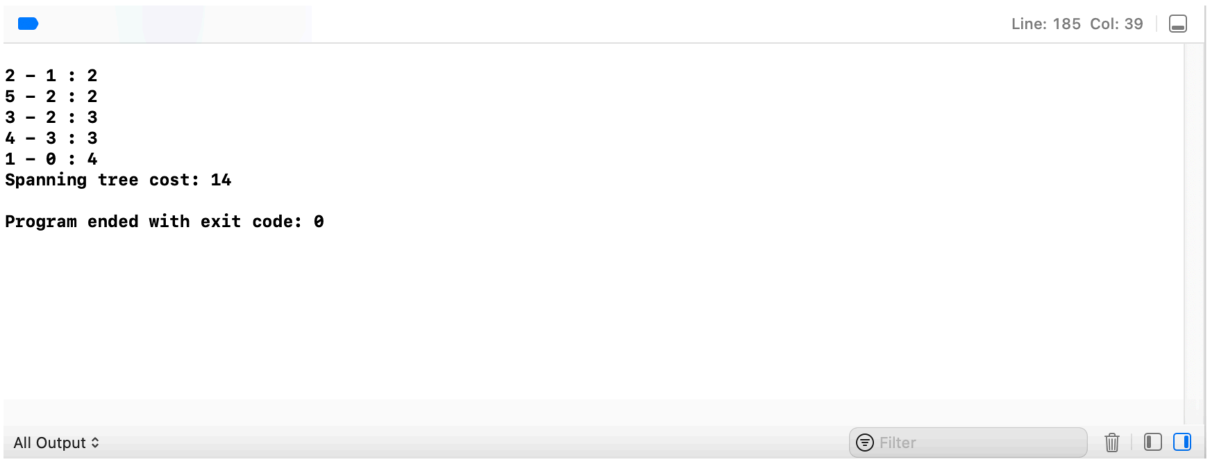
```
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();

void kruskalAlgo() {
  int belongs[MAX], i, j, cno1, cno2;
  elist.n = 0;

  for (i = 1; i < n; i++)
    for (j = 0; j < i; j++) {
      if (Graph[i][j] != 0) {
        elist.data[elist.n].u = i;
        elist.data[elist.n].v = j;
        elist.data[elist.n].w = Graph[i][j];
        elist.n++;
      }
    }
  sort();
  for (i = 0; i < n; i++)
    belongs[i] = i;
  spanlist.n = 0;
  for (i = 0; i < elist.n; i++) {
    cno1 = find(belongs, elist.data[i].u);
    cno2 = find(belongs, elist.data[i].v);

    if (cno1 != cno2) {
      spanlist.data[spanlist.n] = elist.data[i];
      spanlist.n = spanlist.n + 1;
      applyUnion(belongs, cno1, cno2);
    }
  }
}

int find(int belongs[], int vertexno) {
  return (belongs[vertexno]);
}

void applyUnion(int belongs[], int c1, int c2) {
  int i;

  for (i = 0; i < n; i++)
    if (belongs[i] == c2)
      belongs[i] = c1;
}

void sort() {
  int i, j;
  edge temp;

  for (i = 1; i < elist.n; i++)
    for (j = 0; j < elist.n - 1; j++)
      if (elist.data[j].w > elist.data[j + 1].w) {
        temp = elist.data[j];
        elist.data[j] = elist.data[j + 1];
        elist.data[j + 1] = temp;
      }
}
```

```
void print() {
  int i, cost = 0;

  for (i = 0; i < spanlist.n; i++) {
    printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
    cost = cost + spanlist.data[i].w;
  }

  printf("\nSpanning tree cost: %d", cost);
}

int main() {
  int i, j, total_cost;

  n = 6;

  Graph[0][0] = 0;
  Graph[0][1] = 4;
  Graph[0][2] = 4;
  Graph[0][3] = 0;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
  Graph[0][6] = 0;

  Graph[1][0] = 4;
  Graph[1][1] = 0;
  Graph[1][2] = 2;
  Graph[1][3] = 0;
```

```
Graph[1][4] = 0;

Graph[1][5] = 0;

Graph[1][6] = 0;


Graph[2][0] = 4;

Graph[2][1] = 2;

Graph[2][2] = 0;

Graph[2][3] = 3;

Graph[2][4] = 4;

Graph[2][5] = 0;

Graph[2][6] = 0;


Graph[3][0] = 0;

Graph[3][1] = 0;

Graph[3][2] = 3;

Graph[3][3] = 0;

Graph[3][4] = 3;

Graph[3][5] = 0;

Graph[3][6] = 0;


Graph[4][0] = 0;

Graph[4][1] = 0;

Graph[4][2] = 4;

Graph[4][3] = 3;

Graph[4][4] = 0;

Graph[4][5] = 0;

Graph[4][6] = 0;

Graph[5][0] = 0;

Graph[5][1] = 0;
```

```
 Graph[5][2] = 2;

 Graph[5][3] = 0;

 Graph[5][4] = 3;

 Graph[5][5] = 0;

 Graph[5][6] = 0;


 kruskalAlgo();

 print();

}
```

Result:



```
2 - 1 : 2
5 - 2 : 2
3 - 2 : 3
4 - 3 : 3
1 - 0 : 4
Spanning tree cost: 14

Program ended with exit code: 0
```

Experiment No. 7

Aim: a. Print all the nodes reachable from a given starting node in a digraph using the BFS method.
b. Check whether a given graph is connected or not using the DFS method.

Program Logic:

A. Breadth First Search -

   A. BFS uses queue as data structure.

   B. In BFS, if we want to reach from I to j then we first adds vertex I and its adjacent vertices and remove according to queue algorithm and removing any vertex we add it's adjacent vertex and it keeps going until we insert jth vertex in queue.

   C. We never adds any vertex in queue that is already traversed.

B. Depth First Search -

   A. DFS uses stack as data structure.

   B. In DFS, if we want to reach from I to j then we first push vertex I and its adjacent vertices and remove according to stack algorithm and after piping any vertex we add it's adjacent vertex and it keeps going until we push jth vertex in stack.

   C. We never adds any vertex in queue that is already traversed.

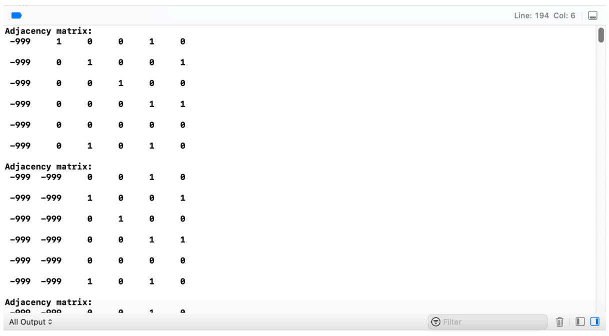
Source code:

```
#include<stdio.h>

void printArray(int col, int arr[][col], int row){

   printf("Adjacency matrix: \n");

   for(int p = 0; p < row; p++){

      for(int q = 0; q < col; q++){

         printf("%5d ", arr[p][q]);

      }

      printf("\n\n");

   }

}
```

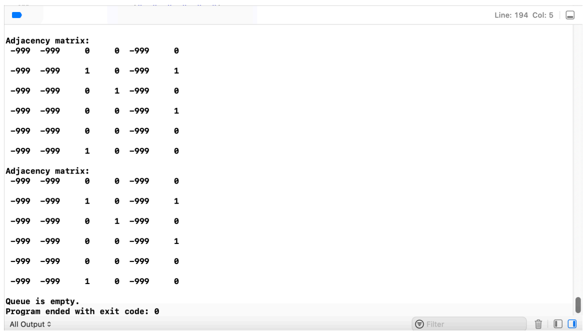
```
void removeTraversedVertice(int col, int arr[][col], int vertice){
    for(int k = 0; k < col; k++){
        arr[k][vertice] = -999;
    }
}

void push(int stack[], int vertice, int *ptr){
    *ptr = *ptr + 1;
    stack[*ptr] = vertice;
}
int pop(int stack[], int *ptr){
    int v = stack[*ptr];
    *ptr = *ptr - 1;
    return v;
}

void dfs(int col, int arr[][col], int row){

    int stack[row], ptr = -1;
    int vertice = 0;

    //initialization with first vertice
    push(stack, vertice, &ptr);
    removeTraversedVertice(col, arr, vertice);
    vertice = pop(stack, &ptr);

    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            if(arr[vertice][j] != -999){
                if(arr[vertice][j] == 1){
                    push(stack, j, &ptr);
                    removeTraversedVertice(col, arr, j);
                }
            }
            printArray(col, arr, row);
        }
        vertice = pop(stack, &ptr);
        printf("\n");
    }
}

void insertV(int queue[], int *rear, int vertice){
    *rear = *rear + 1;
    queue[*rear] = vertice;
}

int removeV(int queue[], int *front, int *rear){
    if(*front > *rear){ //this condition will never occur because no queue will not empty while bfs
        printf("Queue is empty.");
        return -999;
    }
    else{
        *front = *front + 1;
        int v = queue[*front];
        return v;
    }
}
```

```
void bfs(int col, int arr[][col], int row){
   int vertice = 0;
   int queue[row], front = -1, rear = -1;

   //initialize queue by inserting first vertice
   insertV(queue, &rear, vertice);
   removeTraversedVertice(col, arr, vertice);
   vertice = removeV(queue, &front, &rear);

   for(int i = 0; i < row; i++){
      for(int j = 0; j < col; j++){
         if(arr[vertice][j] != -999){
            if(arr[vertice][j] == 1){
               insertV(queue, &rear, vertice);
               removeTraversedVertice(col, arr, j);
            }
         }
         printArray(col, arr, row);
      }
      vertice = removeV(queue, &front, &rear);
      printf("\n");
   }
}

int main(){

   int arr[][6] ={{0, 1, 0, 0, 1, 0},
              {0, 0, 1, 0, 0, 1},
                {0, 0, 0, 1, 0, 0},

                {0, 0, 0, 0, 1, 1},

                {0, 0, 0, 0, 0, 0},

                {1, 0, 1, 0, 1, 0}};


    //dfs(6, arr, 6);

    //bfs(6, arr, 6);

    return 0;

}
```

Results:



```
Adjacency matrix:
 -999     1     0     0     1     0

 -999     0     1     0     0     1

 -999     0     0     1     0     0

 -999     0     0     0     1     1

 -999     0     0     0     0     0

 -999     0     1     0     1     0

Adjacency matrix:
 -999  -999     0     0     1     0

 -999  -999     1     0     0     1

 -999  -999     0     1     0     0

 -999  -999     0     0     1     1

 -999  -999     0     0     0     0

 -999  -999     1     0     1     0

Adjacency matrix:
 -999  -999     0     0     1     0
```



```
Adjacency matrix:
 -999  -999     0     0  -999     0

 -999  -999     1     0  -999     1

 -999  -999     0     1  -999     0

 -999  -999     0     0  -999     1

 -999  -999     0     0  -999     0

 -999  -999     1     0  -999     0

Adjacency matrix:
 -999  -999     0     0  -999     0

 -999  -999     1     0  -999     1

 -999  -999     0     1  -999     0

 -999  -999     0     0  -999     1

 -999  -999     0     0  -999     0

 -999  -999     1     0  -999     0

Queue is empty.
Program ended with exit code: 0
```

Expriment No. 8

Aim: Find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm.

Program Logic: prim’s algorithm works on greedy approach.

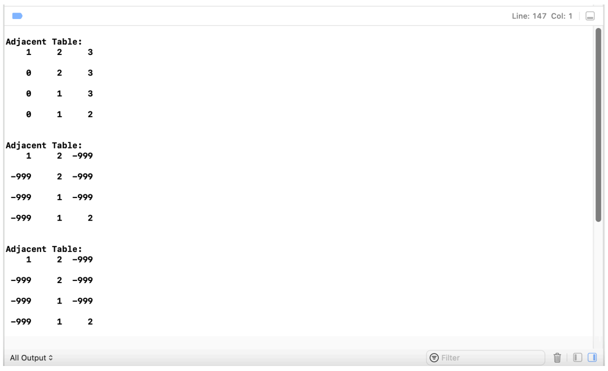
1. Start from any vertex I, and find its adjacent vertex with minimum cost and add it in array.
2. Repeat this process for every vertex.

Source code:

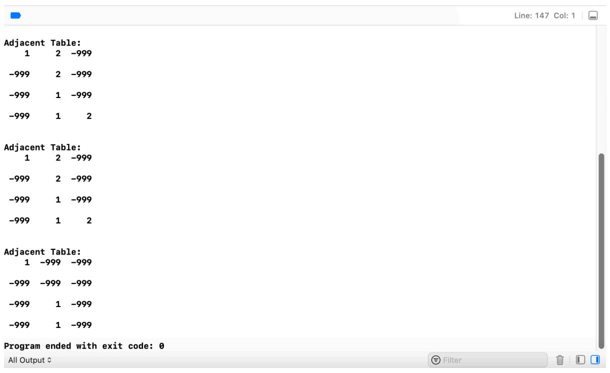
```
#include<stdio.h>

void printArray(int col, int adjTble[][col-1], int row){
   printf("\nAdjacent Table: \n");
   for(int i = 0; i < row; i++){
      for(int j = 0; j < col - 1; j++){
         printf("%5d ", adjTble[i][j]);
      }
      printf("\n\n");
   }
}

void removeTraversedVertice(int col, int adjTble[][col-1], int row, int saveI, int saveJ){
   int remove = adjTble[saveI][saveJ];
   for(int p = 0; p < row; p++){
      for(int q = 0; q < col - 1; q++){
         if(remove == adjTble[p][q]){
            adjTble[p][q] = -999;
         }
      }
```

```
    }
}

void minSpanningTree(int col, int arr[][col], int row, int adjTble[][col-1]){
    printArray(col, adjTble, row);
    removeTraversedVertice(col, adjTble, row, 1, 0);
    int saveI = 0, saveJ = 0;
    int min = 999; //initially
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col - 1; j++){
            if(adjTble[i][j] != -999){
                if(min > arr[i][adjTble[i][j]]){
                    min = arr[i][adjTble[i][j]];
                    saveI = i; saveJ = j;
                }
            }
        }
        removeTraversedVertice(col, adjTble, row, saveI, saveJ);
        printArray(col, adjTble, row);
    }
}

int main(){

    int arr[][4] = {{0, 5, 4, 3},
                    {5, 0, 9, 2},
                    {4, 9, 0, 1},
                    {3, 2, 1, 0}};
    int adjTble[][4-1] = {{1, 2, 3},
                          {0, 2, 3},
                          {0, 1, 3},
                          {0, 1, 2}}; //adjacent table in which row value is used as a vertice number.

    minSpanningTree(4, arr, 4, adjTble);

    return 0;
}
```

Results:



```
Adjacent Table:
    1     2     3

    0     2     3

    0     1     3

    0     1     2


Adjacent Table:
    1     2  -999

 -999     2  -999

 -999     1  -999

 -999     1     2


Adjacent Table:
    1     2  -999

 -999     2  -999

 -999     1  -999

 -999     1     2
```



```
Adjacent Table:
    1     2  -999

 -999     2  -999

 -999     1  -999

 -999     1     2


Adjacent Table:
    1     2  -999

 -999     2  -999

 -999     1  -999

 -999     1     2


Adjacent Table:
    1  -999  -999

 -999  -999  -999

 -999     1  -999

 -999     1  -999

Program ended with exit code: 0
```

Experiment No. 9

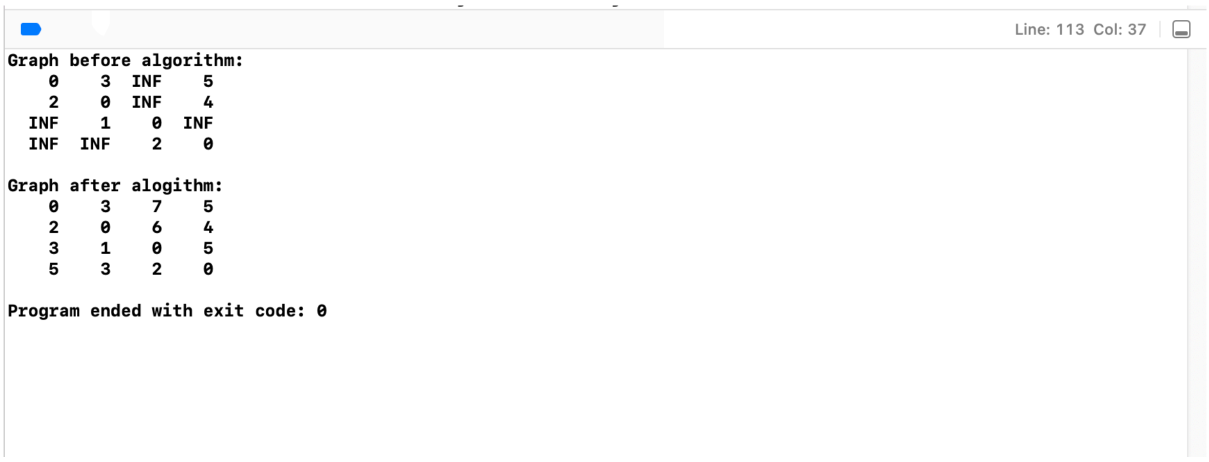
Aim: Implement All-Pairs Shortest Paths Problem using Floyd's algorithm.

Program logic: 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).

Source code:

```
#include <stdio.h>

#define nV 4
#define INF 999

void printMatrix(int matrix[][nV]);

void floydWarshall(int graph[][nV]) {
  int matrix[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      matrix[i][j] = graph[i][j];

  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] + matrix[k][j] < matrix[i][j])
          matrix[i][j] = matrix[i][k] + matrix[k][j];
      }
    }
  }
    printf("Graph after alogithm: \n"); printMatrix(matrix);
}

void printMatrix(int matrix[][nV]) {
  for (int i = 0; i < nV; i++) {
    for (int j = 0; j < nV; j++) {
      if (matrix[i][j] == INF)
        printf("%5s", "INF");
      else
        printf("%5d", matrix[i][j]);
    }
    printf("\n");
  }
    printf("\n");
}

int main() {
  int graph[nV][nV] = {{0, 3, INF, 5},
            {2, 0, INF, 4},
            {INF, 1, 0, INF},
            {INF, INF, 2, 0}};

    printf("Graph before algorithm: \n"); printMatrix(graph);
    floydWarshall(graph);
}
```

Results:



```
Line: 113  Col: 37
```

```
Graph before algorithm:
    0    3  INF    5
    2    0  INF    4
  INF    1    0  INF
  INF  INF    2    0

Graph after alogithm:
    0    3    7    5
    2    0    6    4
    3    1    0    5
    5    3    2    0

Program ended with exit code: 0
```

Experiment No. 10

Aim: Implement N Queen's problem using Backtracking.

Program logic:

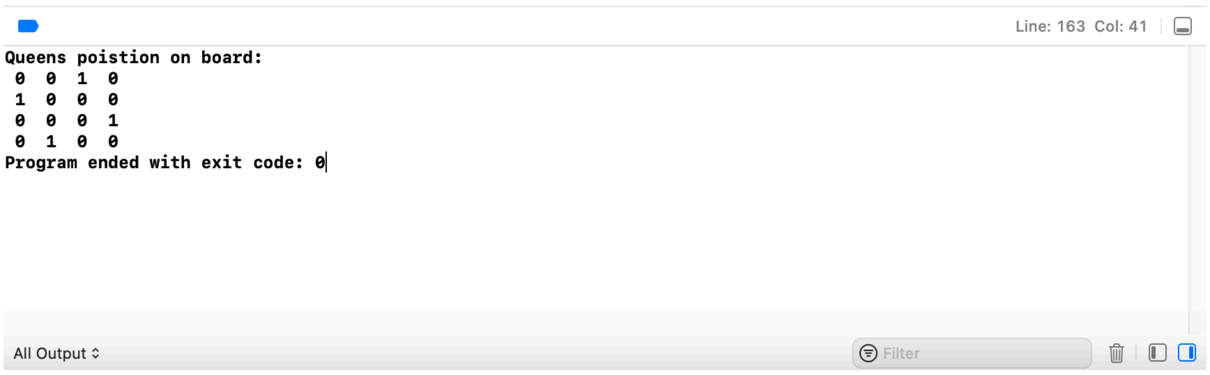
1. Initialize an empty chessboard of size NxN
2. Start with the leftmost column and place a queen in the first row of that column.
3. Move to the next column and place a queen in the first row of that column.
4. Repeat step 3 until either all N queens have been placed or it is impossible to place a queen in the current column without violating the rules of the problem.
5. If all N queens have been placed, print the solution.
6. If it is not possible to place a queen in the current column without violating the rules of the problem, backtrack to the previous column.
7. Remove the queen from the previous column and move it down one row.
8. Repeat steps 4-7 until all possible configurations have been tried.

Source code:

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

#define N 4

void printSolution(int board[N][N])
{
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++)
         printf(" %d ", board[i][j]);
      printf("\n");
   }
}

bool isSafe(int board[N][N], int row, int col)
{
   int i, j;

   /* Check this row on left side */
   for (i = 0; i < col; i++)
      if (board[row][i])
         return false;
```

```
    /* Check upper diagonal on left side */
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    /* Check lower diagonal on left side */
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

bool solveNQUtil(int board[N][N], int col)
{
    /* base case: If all queens are placed
    then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
    this queen in all rows one by one */
    for (int i = 0; i < N; i++) {
        /* Check if the queen can be placed on
        board[i][col] */
        if (isSafe(board, i, col)) {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1))
                return true;

            board[i][col] = 0; // BACKTRACK
        }
    }
    return false;
}

bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }

    printf("Queens poistion on board: \n"); printSolution(board);
    return true;
}

int main()
{
    solveNQ();
    return 0;
}
```

Results:



```
Queens poistion on board:
 0  0  1  0
 1  0  0  0
 0  0  0  1
 0  1  0  0
Program ended with exit code: 0
```