**LAB-4**

**NAME – TUSHAR MAROO**

**SECTION - A**

**REGISTRATION NUMBER – 190905238**

**ROLL NUMBER – 40**

**Q1.**

**ALGORITHM**

We generate n! possible job assignments and for each such assignment, we compute its total cost and return the less expensive assignment. Since the solution is a permutation of the n jobs, its complexity is O(n!).

**CODE**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int ans[1000], min = INT\_MAX, opcount1 = 0, opcount2 = 0;

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

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void permute(int r, int arr[][r + 1], int per[], int l)

{

int i;

if (l == r)

{

int sum = 0;

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

{

opcount2++;

int idx = per[i];

sum += arr[i][idx];

}

if (sum < min)

{

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

{

int idx = per[i];

ans[i] = arr[i][per[i]];

}

min = sum;

}

}

else

{

for (i = l; i <= r; i++)

{

opcount1++;

swap((per + l), (per + i));

permute(r, arr, per, l + 1);

swap((per + l), (per + i));

}

}

}

int main()

{

int i, j, n;

printf("Enter the size of the square matrix : ");

scanf("%d", &n);

int arr[n][n];

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

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

{

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

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

// arr[i][j]=1;

}

int per[n];

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

per[i] = i;

permute(n - 1, arr, per, 0);

printf("Combination for minimum cost : ");

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

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

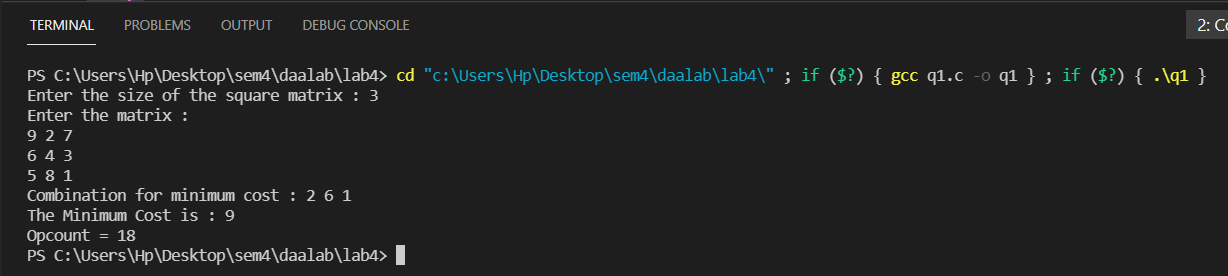
printf("\nThe Minimum Cost is : %d\n", min);

printf("Opcount = %d\n", opcount1 > opcount2 ? opcount1 : opcount2);

return 0;

}

**OUTPUT**

****

**ANALYSIS**

We will always need to check sum of time for all n! permutations. So the time complexity in the best as well as the worst case would be O(n!). So basically there is no best and worst case all cases take similar time depending on n.

**GRAPH**

**Q2.**

**ALGORITHM**

Implements a breadth-first search traversal of a given graph

Input: Graph G = V, E

Output: Graph G with its vertices marked with consecutive integers

in the order they are visited by the BFS traversal

mark each vertex in V with 0 as a mark of being “unvisited”

count ←0

for each vertex v in V do

if v is marked with 0

bfs(v)

bfs(v)

visits all the unvisited vertices connected to vertex v

by a path and numbers them in the order they are visited

via global variable count

count ←count + 1; mark v with count and initialize a queue with v

while the queue is not empty do

for each vertex w in V adjacent to the front vertex do

if w is marked with 0

count ←count + 1; mark w with count

add w to the queue

remove the front vertex from the queue

**CODE**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

int top = -1, stack[MAX];

void push(int val)

{

if (top == MAX - 1)

{

printf("\nStack is full!!");

}

else

{

top = top + 1;

stack[top] = val;

printf("Pushed %d.\n", stack[top]);

}

}

int pop()

{

if (top == -1)

{

printf("\nStack is empty!!");

}

else

{

printf("Popped %d.\n", stack[top]);

top = top - 1;

return stack[top + 1];

}

}

typedef struct listNode ListNode;

struct listNode

{

ListNode \*next;

int data;

};

ListNode \*createNode(int x)

{

ListNode \*temp = (ListNode \*)malloc(sizeof(ListNode));

temp->data = x;

temp->next = NULL;

return temp;

}

ListNode \*insert(ListNode \*head, int x)

{

if (head == NULL)

{

head = createNode(x);

return head;

}

ListNode \*temp = head;

while (temp->next != NULL)

temp = temp->next;

temp->next = createNode(x);

return head;

}

void print(ListNode \*head)

{

ListNode \*temp = head;

while (temp != NULL)

{

printf("%d --> ", temp->data);

temp = temp->next;

}

printf("NULL\n");

}

void dfs(ListNode \*\*adjList, int visited[],int DFS[], int v)

{

int idx=0;

push(v);

while (top != -1)

{

v = pop();

if (visited[v])

{

continue;

}

visited[v] = 1;

DFS[idx++]=v;

ListNode \*temp = adjList[v];

while (temp != NULL)

{

if (!visited[temp->data])

push(temp->data);

temp = temp->next;

}

}

}

int main()

{

int v, e, i, j;

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

scanf("%d", &v);

ListNode \*\*adjList = malloc(v \* sizeof(ListNode \*));

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

adjList[i] = NULL;

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

scanf("%d", &e);

int start, end, visited[v],DFS[v];

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

visited[i] = 0;

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

{

printf("Enter start and end vertex of edge %d : ",i+1);

scanf("%d%d", &start,&end);

adjList[start] = insert(adjList[start], end);

adjList[end] = insert(adjList[end], start);

}

printf("\nAdjacency List : \n");

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

{

printf("%d : ", i);

print(adjList[i]);

}

printf("\n");

dfs(adjList, visited,DFS, 0);

printf("\nDFS : ");

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

{

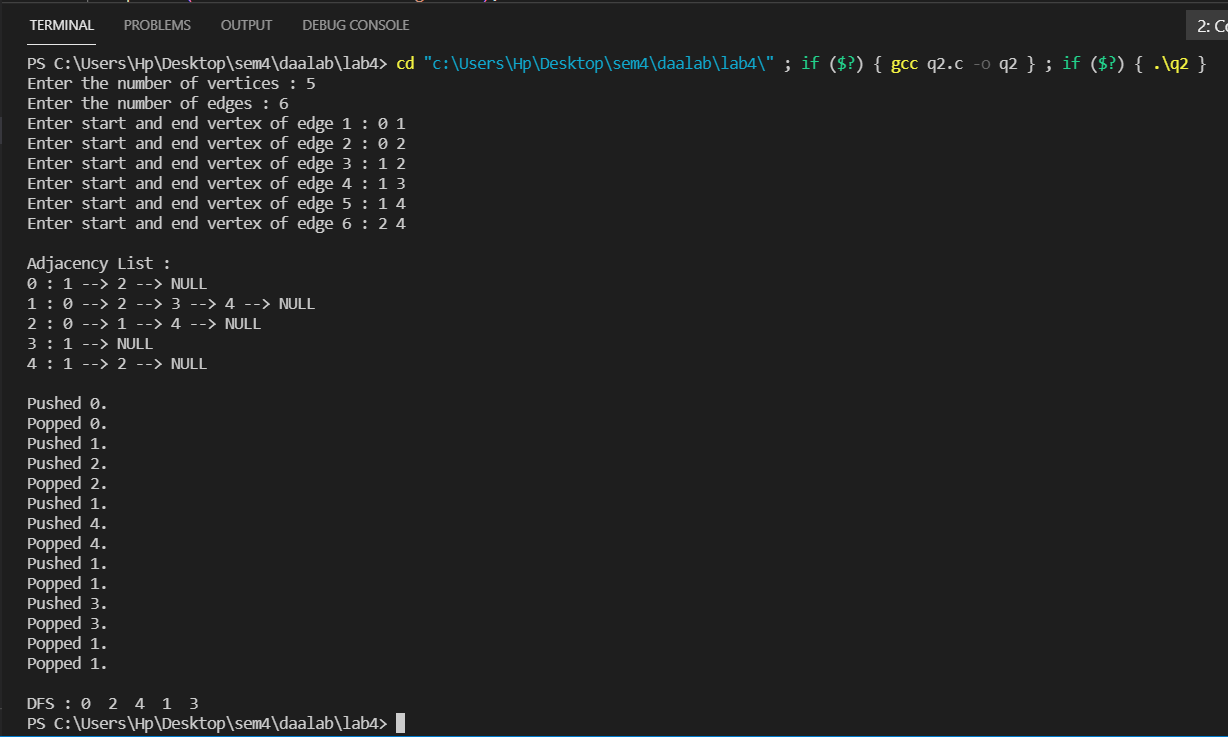
printf("%d ", DFS[i]);

}

printf("\n");

}

**OUTPUT**

****

**ANALYSIS**

The time complexity of DFS if the entire tree is traversed is O(V) where V is the number of nodes.

Here, each node maintains a list of all its adjacent edges. Let’s assume that there are V number of nodes and E number of edges in the graph.

For each node, we discover all its neighbors by traversing its adjacency list just once in linear time.

For a directed graph, the sum of the sizes of the adjacency lists of all the nodes is E. So, the time complexity in this case is O(V) + O(E) = O(V + E)

**Q3.**

**ALGORITHM**

Implements a depth-first search traversal of a given graph

Input: Graph G = V, E

Output: Graph G with its vertices marked with consecutive integers

in the order they are first encountered by the DFS traversal

mark each vertex in V with 0 as a mark of being “unvisited”

count ←0

for each vertex v in V do

if v is marked with 0

dfs(v)

dfs(v)

visits recursively all the unvisited vertices connected to vertex v

by a path and numbers them in the order they are encountered

via global variable count

count ←count + 1; mark v with count

for each vertex w in V adjacent to v do

if w is marked with 0

dfs(w)

**CODE**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

int queue[MAX];

int rear = -1;

int front = -1;

void enqueue(int item)

{

if (rear == MAX - 1)

printf("Queue Overflow \n");

else

{

if (front == -1)

front = 0;

rear++;

queue[rear] = item;

printf("Inserted %d.\n",item);

}

}

int dequeue ()

{

if (front == -1 || front > rear)

{

printf("Queue Underflow \n");

return -1;

}

else

{

printf("Deleted %d.\n",queue[front]);

front++;

return queue[front-1];

}

}

typedef struct listNode ListNode;

struct listNode

{

ListNode \*next;

int data;

};

ListNode \*createNode(int x)

{

ListNode \*temp = (ListNode \*)malloc(sizeof(ListNode));

temp->data = x;

temp->next = NULL;

return temp;

}

ListNode \*insert(ListNode \*head, int x)

{

if (head == NULL)

{

head = createNode(x);

return head;

}

ListNode \*temp = head;

while (temp->next != NULL)

temp = temp->next;

temp->next = createNode(x);

return head;

}

void print(ListNode \*head)

{

ListNode \*temp = head;

while (temp != NULL)

{

printf("%d --> ", temp->data);

temp = temp->next;

}

printf("NULL\n");

}

void bfs(ListNode \*\*adjList,int visited[],int BFS[],int v)

{

int idx=0;

visited[v]=1;

enqueue(v);

while(!(front == -1 || front > rear))

{

v=dequeue();

BFS[idx++]=v;

ListNode \*temp = adjList[v];

while (temp != NULL)

{

if (!visited[temp->data])

{

visited[temp->data]=1;

enqueue(temp->data);

}

temp = temp->next;

}

}

}

int main()

{

int v, e, i, j;

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

scanf("%d", &v);

ListNode \*\*adjList = malloc(v \* sizeof(ListNode \*));

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

adjList[i] = NULL;

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

scanf("%d", &e);

int start, end, visited[v], BFS[v];

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

visited[i] = 0;

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

{

printf("Enter start and end vertex of edge %d : ", i + 1);

scanf("%d%d", &start, &end);

adjList[start] = insert(adjList[start], end);

adjList[end] = insert(adjList[end], start);

}

printf("\nAdjacency List : \n");

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

{

printf("%d : ", i);

print(adjList[i]);

}

printf("\n");

bfs(adjList, visited, BFS, 0);

printf("\nBFS : ");

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

{

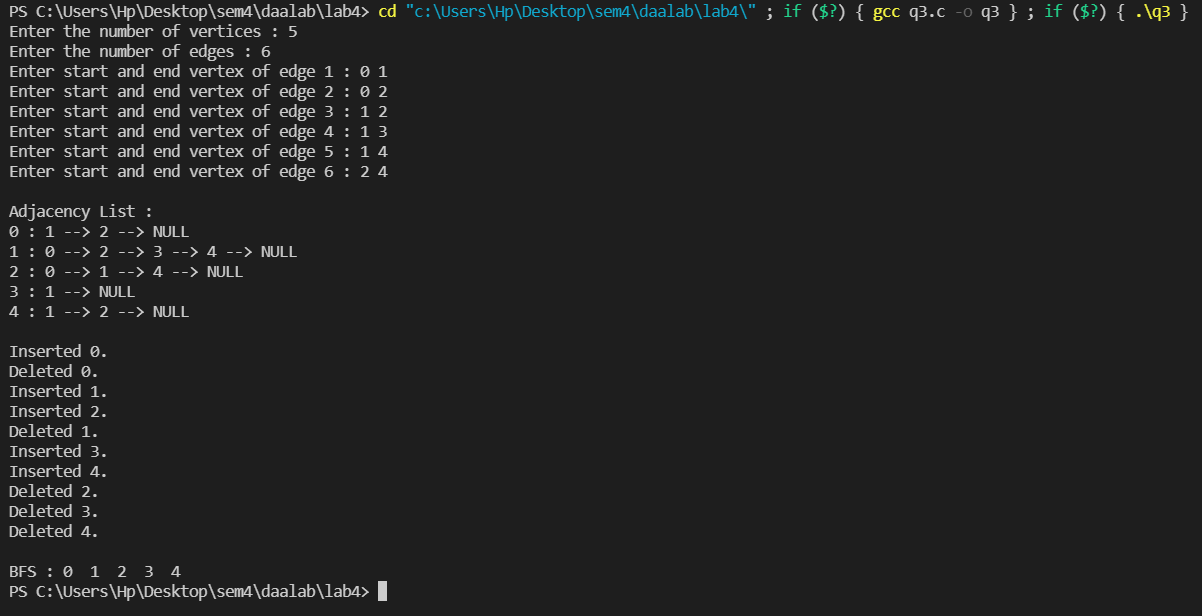
printf("%d ", BFS[i]);

}

printf("\n");

}

**OUTPUT**

****

**ANALYSIS**

The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.