

## LAB 4

Aniket Sambher

Reg no-190905466

Roll no-58

1.

### **CODE**

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

```
#include <limits.h>
```

```
int n;
```

```
int r1;
```

```
int combinations[1000][10];
```

```
int fact(int n)
```

```
{
```

```
    if (n == 1)
```

```
        return 1;
```

```
    else
```

```
        return fact(n - 1) * n;
```

```
}
```

```
void permute(int numbers[], int l, int r)
```

```
{
```

```
    if (l == r)
```

```
    {
```

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

```
            combinations[r1][i] = numbers[i];
```

```

        r1++;
        return;
    }

    for (int i = l; i <= r; i++)
    {
        int temp = numbers[l];

        numbers[l] = numbers[i];
        numbers[i] = temp;

        permute(numbers, l + 1, r);

        temp = numbers[l];
        numbers[l] = numbers[i];
        numbers[i] = temp;
    }
}

int main()
{
    int opcount = 0;

    printf("Enter n:");
    scanf("%d", &n);

    int square[n][n];

    for (int i = 0; i < n; i++)
    {
        printf("Enter costs for person %d: ", (i + 1));

```

```

        for (int j = 0; j < n; j++)
            scanf("%d", &square[i][j]);
    }

    int numbers[n];

    for (int i = 0; i < n; i++)
        numbers[i] = i;

    r1 = 0;

    permute(numbers, 0, n - 1); //using backtracking for finding permutations
int sum[fact(n)];
for(int i=0;i<fact(n);++i)
    sum[i]=0;
for(int i=0;i<fact(n);++i)
{
    opcount++;
    for(int j=0;j<n;++j){

        sum[i]+=square[j][combinations[i][j]];

    }
}

int min=INT_MAX;
for(int i=0;i<fact(n);++i)
    if(sum[i]<min)
        min=sum[i];

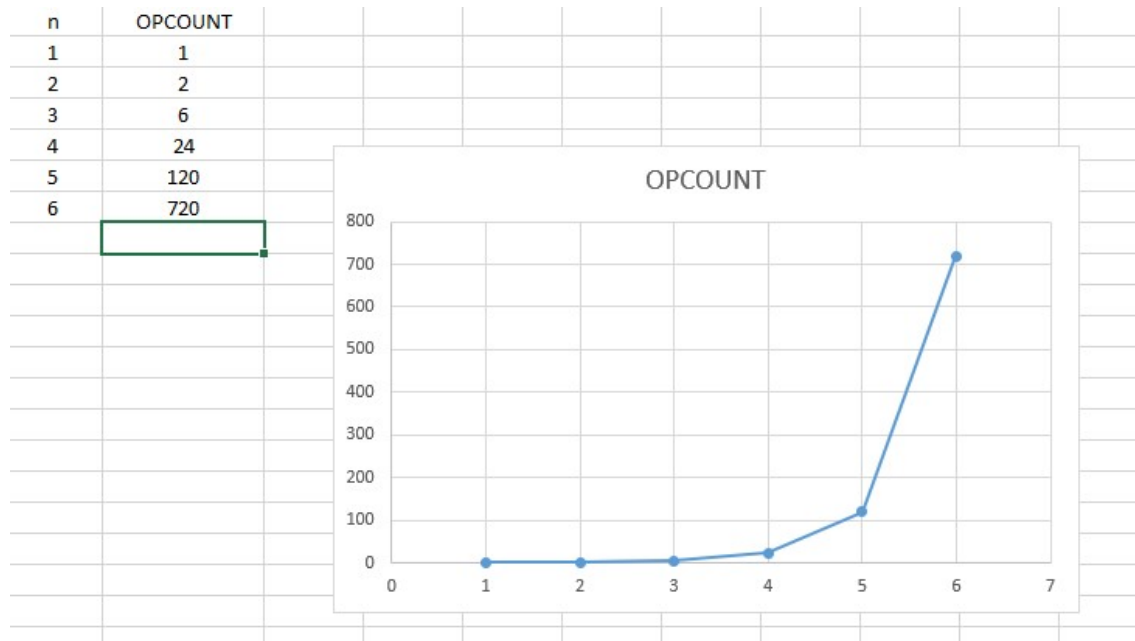
printf("%d",min);

    printf("\nopcount= %d",opcount);
}

```

## OUTPUT

```
PS C:\Users\aniket\Desktop\desktop\sem4\daa\lab\daa\week4> cd "c:\Users\aniket\Desktop\desktop\sem4\daa\lab\daa\week4\" ; if ($?) { gcc assignment
.c -o assignment } ; if ($?) { .\assignment }
Enter n:4
Enter costs for person 1: 1 2 3 4
Enter costs for person 2: 5 3 2 1
Enter costs for person 3: 2 4 2 1
Enter costs for person 4: 1 2 4 5
6
opcount= 24
PS C:\Users\aniket\Desktop\desktop\sem4\daa\lab\daa\week4> |
```



## TIME COMPLEXITY ANALYSIS

For brute force solution we check for all possible combinations of the jobs i.e  $n!$

Thereby time complexity  $O(n!)$

2.

### **ALGORITHM DFS(G)**

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  $\leftarrow 0$

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  $\leftarrow$  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>
```

```
//global variables for dfs
```

```
int v;
```

```
int matrix[100][100];
```

```
int visited[100];
```

```
int push[100], pop[100];
```

```
int pushcount = 0, popcount = 0;
```

```

void dfsv(int vertex){
    printf("Pushing vertex : %d\n", vertex);
    push[pushcount++] = vertex;
    visited[vertex] = 1;
    for(int i=0; i<v; ++i){
        if((!visited[i]) && (matrix[vertex][i]==1) && (i!=vertex)){
            dfsv(i);
        }
    }
    printf("Popping vertex : %d\n", vertex);
    pop[popcount++] = vertex;
}

```

```

void dfs(){
    int i;
    for(i=0; i<v; ++i){
        if(!visited[i]){
            dfsv(i);
        }
    }
}

```

```

int main(){
    printf("Enter number of vertices:");
    scanf("%d", &v);
    printf("Enter adjacency matrix:\n");
    for(int i=0; i<v; i++){
        for(int j=0; j<v; j++){
            scanf("%d", &matrix[i][j]);
        }
    }
}

```

```

    }

    dfs();

    printf("Push order:");

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

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

    }

    printf("\nPop order:");

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

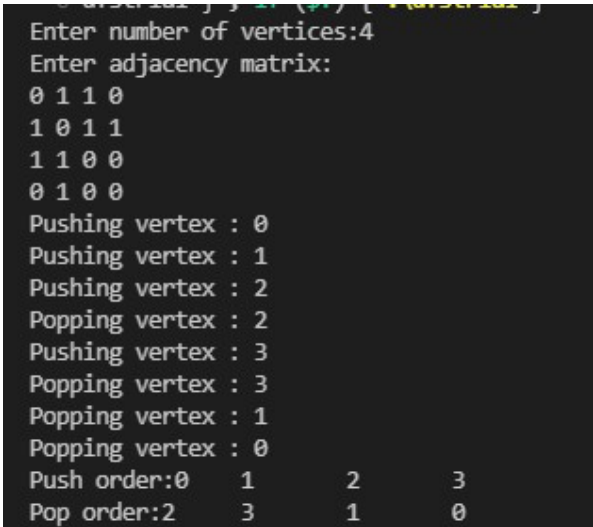
        printf("%d\t", pop[i]);

    }

}

```

## OUTPUT



```

Enter number of vertices:4
Enter adjacency matrix:
0 1 1 0
1 0 1 1
1 1 0 0
0 1 0 0
Pushing vertex : 0
Pushing vertex : 1
Pushing vertex : 2
Popping vertex : 2
Pushing vertex : 3
Popping vertex : 3
Popping vertex : 1
Popping vertex : 0
Push order:0    1    2    3
Pop order:2    3    1    0

```

## TIME COMPLEXITY ANALYSIS

If the graph is represented as an **adjacency matrix** (a  $V \times V$  array):

For each node, we will have to traverse an entire row of length  $V$  in the matrix to discover all its outgoing edges.

Note that each row in an adjacency matrix corresponds to a node in the graph, and that row stores information about edges emerging from the node. Hence, the time complexity of DFS in this case is  $O(V * V) = O(V^2)$ .

3.

**ALGORITHM** *BFS(G)*

```
//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>

//global variables

int matrix[100][100];

int V;

int visited[100];

int queue[100], f = 0, r = 0;

//functions for implementation of queue

void enqueue(int v)
```



```

{
queue[r++] = v;
}

int dequeue()
{
if(f == r)
    return -1;
return queue[f++];
}

void bfsv(int v)
{
printf("Visiting %d\n", v);
visited[v] = 1;
int i;
for(i = 0; i < V; ++i)
{
if(!visited[i] && matrix[v][i] && i != v)

    enqueue(i);
}
}

void bfs(){
int i, x;
enqueue(0);
do
{
x = dequeue();
if(x != -1 && !visited[x])
    bfsv(x);
}while (x != -1);
}

```

```

void main()
{
printf("Enter the Number of Vertices : \n");

scanf("%d", &V);

int i, j;

printf("Enter the Adjacency Matrix: \n");

for (i = 0; i < V; ++i)
{
for (j = 0; j < V; ++j)

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

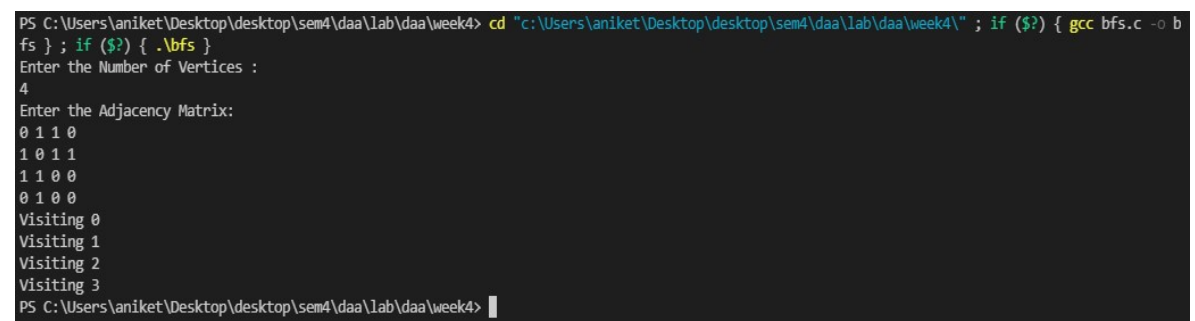
}

bfs();

}

```

## OUTPUT



```

PS C:\Users\aniket\Desktop\desktop\sem4\lab\daa\week4> cd "c:\Users\aniket\Desktop\desktop\sem4\lab\daa\week4\" ; if ($?) { gcc bfs.c -o b
fs } ; if ($?) { .\bfs }
Enter the Number of Vertices :
4
Enter the Adjacency Matrix:
0 1 1 0
1 0 1 1
1 1 0 0
0 1 0 0
Visiting 0
Visiting 1
Visiting 2
Visiting 3
PS C:\Users\aniket\Desktop\desktop\sem4\lab\daa\week4>

```

## TIME COMPLEXITY ANALYSIS

For BFS we visit each vertex's neighbours and push them into the queue till all the vertex are visited i.e we traverse the adjacency matrix for each vertex till all neighbours are visited

**Time complexity  $O(v^2)$**  as we are using adjacency matrix