Subject Name: DAA
Subject Code: 303105219
B. Tech-CS 3rd Year 5th Semester

PARUL UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL (2025-2026)

Faculty Name:

Name: .....

Enroll. No.: .....

(Roll No.:) .....

Div.: .....

Subject: DAA

Date Of Submission: .....

Faculty Sign: .....

Parul® University | NAAC A++

Subject Name: DAA
Subject Code: 303105219
B. Tech-CS 3rd Year 5th Semester

CERTIFICATE

This is to Certify that Mr/Ms. Tejasvi Raj Singh with Enrolment No. 2303051051372 has successfully completed his Laboratory experiments in **Design and Analysis of Algorithm (303105218)** From the department of **COMPUTER SCIENCE AND ENGINEERING** during the academic year **2025-2026**

Date of Submission : Staff In Charge: .....
.....

Head of department: .....

INDEX

Sr.No.	Name of Practical	Date	Signature
1	Prime Check		
2	Search Insert Position		
3	Minimum Candy Distribution		
4	Largest Minimum Distance Between Cows		
5	Cycle Detection in an Undirected Graph		
6	Find All Critical Connections in a Network		
7	Count the Number of Islands in a Grid		
8	Rot All Oranges in Minimum Time		
9	Minimum Edit Distance Between Two Strings		
10	Minimum Path Sum		
11	Smallest Number After Removing k Digits		
12	Unique Paths in a Grid		

Experiment - 1

Experiment - 1: ***Prime Check***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
Write a C program that determines whether a given positive integer is prime or not.

Input Format:

- A single integer n (where n > 1) representing the number to be checked.

Output Format:

- Print "Prime" if the number is prime.
- Print "Not Prime" if the number is not prime.

Program:

primeCheck.c

```
#include <stdio.h>

int main() {
    int num, i, is_prime = 1;

        scanf("%d", &num);

        if (num <= 1) {
                is_prime =0;
        }
        else {
        for (i = 2; i <= num / 2; i++){
                if(num % i == 0) {
                        is_prime = 0;
                        break;
                }
        }

                }

    if (is_prime) {
        printf("Prime\n");
    } else {
        printf("Not Prime\n");
    }

    return 0;
}
```

Output:

Test case - 1
User Output

9
Not Prime

Test case - 2
User Output
7
Prime

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 2

Experiment - 2: ***Search Insert Position***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
You are given a sorted array and a target value. Your task is to determine the index where the target value should be inserted if it is not found in the array. If the target value exists in the array, return its index.

Input Format:

- The first line of input reads an integer value (n), representing the number of elements in the array.
- The second line reads n space-separated integers representing the sorted array.
- The third line reads an integer value (target) representing the target value to search for.

Output Format:

- Print the index at which the target value is found or should be inserted.

Program:

search_insert_pos.c

```
//write your code here....
#include<stdio.h>
 void main(){
         int size, flag =0;
         scanf("%d", &size);
         int a[size], target;
         for(int i =0; i<size; i++){
                 scanf("%d", &a[i]);

         }
         scanf("%d", &target);
         for(int j=0; j<size; j++){
                 if(a[j]==target){
                         printf("%d",j);
                         flag =1;
                         break;
                 }
                 if (a[j]>target){
                         printf("%d", j);
                         flag =1;
                         break;

                 }
         }
         if(flag==0){
                 printf("%d",size);

         }
 }
```

Output:

Test case - 1
User Output
5
1 3 5 6 9
5
2

Test case - 2
User Output
4
2 4 6 8
1
0

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 3

Experiment - 3: ***Minimum Candy Distribution***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
There are N children standing in a line, each with a given rating value. You need to distribute candies to the children while following these rules:

- Each child must get at least one candy.
- A child with a higher rating than their neighbor(s) must receive more candies than them.

Write a C program to determine the minimum number of candies required to satisfy these conditions.

Input Format:

- The first line contains a positive integer N (number of children).
- The second line contains N space-separated integers, representing the rating values of each child.

Output Format:

- Print a single integer representing the minimum number of candies required.

Program:

`minimum_candies.c`

```
//  Write your code here

#include<stdio.h>
void main(){
        int n;
        scanf("%d", &n);

         int rating[n];

        for(int i=0; i<n;i++){
                scanf("%d", &rating[i]);

        }
        int candies[n];
        for(int i= 0; i<n;i++){
                candies[i] =1;

        }
        for(int i=1;i<n;i++){
                if(rating[i]>rating[i-1]){
                        candies[i]= candies[i-1]+1;
                }
        }
        for(int i=n-2;i>=0;i--){
                if(rating[i]>rating[i+1]){
                        if(candies[i]<=candies[i+1]){
                                candies[i]= candies[i+1]+1;
                        }
                }
        }
int sumcandies=0;
        for(int i =0; i<n;i++){
                sumcandies += candies[i];
        }
        printf("%d\n",sumcandies);

}
```

Output:

Test case - 1
User Output
5
1 2 3 4 5
15

Test case - 2
User Output
4

4 3 2 1
10

Test case - 3
User Output
5
2 2 2 2 2
5

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 4

Experiment - 4: ***Largest Minimum Distance Between Cows***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
Write a C program to calculate the largest minimum distance that can be achieved when placing all cows in the stalls.

You are given N stalls located along a straight line at positions $x1, x2, \ldots, xN$ (where 0 ≤ x_i ≤ 1,000,000,000) and C cows that must be placed in these stalls.
You need to place the cows in such a way that the minimum distance between any two cows is as large as possible.

Input Format:

- The first line contains two integers, N (number of stalls) and C (number of cows).
- The second line contains N space-separated integers, the positions of the stalls.

Output Format:

- Print a single integer representing the largest minimum distance possible value between any two cows.

Note: Refer to the visible test cases for a better understanding.

Program:

`aggressive_cows.c`

```
//write your code here...
#include<stdio.h>
#include<stdlib.h>

int compare(const void *a, const void *b){
        return (*(int*)a-*(int*)b);

}
int canPlaceCows(int stalls[],int n, int cows,int dist){
        int placed = 1;
        int lastPos = stalls[0];
        for(int i=1; i<n;i++){
                if(stalls[i]-lastPos >= dist){
                        placed++;
                        lastPos = stalls[i];
                        if(placed == cows){
                                return 1;
                        }
                }
        }
        return 0;
}
int largestMinDistance(int stalls[],int n, int cows){
        qsort(stalls, n, sizeof(int), compare);
int low=1;
        int high = stalls[n-1]-stalls[0];
        int best = 0;
        while(low<=high){
                int mid= low + (high - low )/2;
                if(canPlaceCows(stalls,n,cows,mid)){
                        best = mid;
                        low = mid+1;

                } else{
                        high = mid-1;
                }
        }
        return best;
}
int main(){
        int n , cows;
        scanf("%d %d",&n,&cows);
        int stalls[n];
        for(int i =0; i<n;i++){
                scanf("%d",&stalls[i]);
        }
        int result = largestMinDistance(stalls,n,cows);
        printf("%d\n",result);
}
```

Output:

Test case - 1
User Output
5 3
1 2 8 4 9
3

Test case - 2
User Output
6 3
5 1 3 7 9 2
4

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 5

Experiment - 5: ***Cycle Detection in an Undirected Graph***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

Input Format:

- The first line contains two integers, V and E — the number of vertices and edges in the graph.
- The next E lines each contain two integers, u and v, indicating an undirected edge between vertices u and v.

Output Format:

- Print "Cycle Detected" if a cycle is present; otherwise, print "No Cycle".

Program:

`check_cycle.c`

```
//write your code here...
#include<stdio.h>

#define MAX 1000

int graph[MAX][MAX];
int visited[MAX];

int dfs(int current , int parent, int n){
        visited[current] =1;

        for(int i = 0;i<n; i++){
                if(graph[current][i]){
                        if(!visited[i]){
                                if (dfs(i,current,n))
                                        return 1;
                        } else if (i != parent){
                                return 1;
                        }
                }
        }
        return 0;
}
int main(){
        int vertices, edges;
        scanf("%d %d", &vertices, &edges);

        for(int i = 0; i <vertices; i++){
                visited[i] =0;
                for (int j =0; j<vertices; j++){
                        graph[i][j] = 0;

                }
        }
        for(int i = 0; i<edges ; i++){
                int u ,v;
                scanf("%d %d", &u,&v);
                graph[u][v] = 1;
                graph[v][u] = 1;

        }
        int cycleFound = 0;
        for(int i = 0; i< vertices; i++){
                if(!visited[i]){
                        if(dfs(i,-1,vertices)){
                                cycleFound=1;
                                break;
                        }
                }
        }
        if (cycleFound)
                printf("Cycle Detected\n");
        else
                printf("No Cycle\n");
```

```
        return 0;
}
```

Output:

Test case - 1
User Output
5 5
0 1
1 2
2 3
3 4
4 1
Cycle Detected

Test case - 2
User Output
4 2
0 1
2 3
No Cycle

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 6

Experiment - 6: ***Find All Critical Connections in a Network***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
Write a C program to find all critical connections in a network of n servers. The servers are numbered from **0** to $n-1$ and are connected via undirected connections. Each connection is represented as a pair $[a, b]$, where server a is directly connected to server b.

A connection is considered critical if removing it disconnects the network, meaning at least one server cannot reach another. Your task is to identify and print all such critical connections in any order.

Input Format:

- The first line contains two integers, n and m — the number of servers and the number of connections.
- The next m lines each contain two space-separated integers, a and b — representing a direct connection between server a and server b.

Output Format:

- For each critical connection, print the two endpoints on a new line in the format: a b.

Note: Refer to the visible test cases for a better understanding.

Program:

`critical_connections.c`

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

#define MAXN 10005

#define MAXM 20005

typedef struct Node{
int vertex;
struct Node* next;


} Node;

Node* adj[MAXN];

int ids[MAXN], low[MAXN], visited[MAXN];
int id;
int bridgesA[MAXM], bridgesB[MAXM], bridgeCount;

 void addEdge(int u , int v){
         Node* node  = (Node*)malloc(sizeof(Node));
 node->vertex = v;
         node->next = adj[u];
         adj[u] = node;
         node = (Node*)malloc(sizeof(Node));
         node->vertex = u;
         node->next = adj[v];
         adj[v] = node;
 }

void dfs(int at, int parent){
        visited[at] = 1;
        ids[at] = low[at] = ++id;

          for (Node* cur = adj[at]; cur; cur= cur->next){
                  int to = cur->vertex;
              if (to == parent) continue;
                  if (!visited[to]){
                          dfs(to , at);
                          low[at] = (low[at] < low[to] ) ? low[at] : low[to];


                          if(low[to] > ids[at]){
                                  if(at < to ){
                                          bridgesA[bridgeCount] = at;
                                          bridgesB[bridgeCount] =to;

                                  }
                                  else {
                                          bridgesA[bridgeCount] = to;
                                          bridgesB[bridgeCount] = at;
```

```
                                }

                                bridgeCount++;
                        }

                }
                else {
                        low[at] = (low[at] < ids[to]) ? low[at] : ids[to];
                }
          }
}
int cmp(const void* a, const void* b){
        int* pa = (int*)a;
        int* pb = (int*)b;
        if(pa[0] != pb[0])
                return pb[0] - pa[0];
        return pb[1] - pa[1];
}

int main(){
        int n, m;
        scanf("%d %d", &n , &m);

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

        for (int i = 0;i < m; ++i){
                int u, v;
                scanf("%d %d", &u, &v);
                addEdge(u,v);
        }

        bridgeCount = 0; id = 0;
        memset(visited, 0 , sizeof(visited));

        for(int i = 0; i < n; ++i){
                if(!visited[i])
                        dfs(i, -1);

        }

        int out[MAXM][2];
        for(int i = 0; i < bridgeCount; ++i){
                if(bridgesA[i] < bridgesB[i]){
                out[i][0] = bridgesA[i];
                out[i][1] = bridgesB[i];
        } else {
                out[i][0] = bridgesB[i];
                out[i][1] = bridgesA[i];
         }
        }

        qsort(out, bridgeCount,sizeof(out[0]), cmp);

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

```
                printf("%d %d\n", out[i][0], out[i][1]);
        return 0;
}
```

Output:

Test case - 1
User Output
4 4
0 1
1 2
2 0
1 3
1 3

Test case - 2
User Output
5 5
0 1
1 2
2 3
3 4
4 2
1 2
0 1

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 7

Experiment - 7: ***Count the Number of Islands in a Grid***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
Write a C program to count the number of islands in a given grid.
An island is formed by connecting adjacent lands horizontally or vertically. A cell with value '**1**' represents land, and a cell with value '**0**' represents water.
Two cells are considered connected if they are adjacent horizontally or vertically (not diagonally).
You need to determine how many such islands are present in the entire grid.

Input Format:

- The first line contains two integers, N and M (number of rows and columns).
- The next N lines contain M characters (0 or 1) each, representing the grid.

Output Format:

- Print a single integer denoting the number of islands.

Program:

`count_islands.c`

```c
#include <stdio.h>

int n, m;
int grid[105][105];
int visited[105][105];

int dr[4] = {-1, 1, 0, 0};
int dc[4] = {0, 0, -1, 1};

void dfs(int r, int c) {
    visited[r][c] = 1;
    for (int k = 0; k < 4; k++) {
        int nr = r + dr[k];
        int nc = c + dc[k];
        if (nr >= 0 && nr < n && nc >= 0 && nc < m && !visited[nr][nc] &&
grid[nr][nc] == 1) {
            dfs(nr, nc);
        }
    }
}

int main() {
    scanf("%d %d", &n, &m);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d", &grid[i][j]);
        }
    }
    int islands = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (!visited[i][j] && grid[i][j] == 1) {
                dfs(i, j);
                islands++;
            }
        }
    }
    printf("%d\n", islands);
    return 0;
}
```

Output:

Test case - 1
User Output
4 5
1 1 0 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1
3

Test case - 2
User Output
3 3
1 1 1
1 1 1
1 1 1
1

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 8

Experiment - 8: ***Rot All Oranges in Minimum Time***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
Given a grid of dimension NxM where each cell in the grid can have the following values:

- **0**: Empty cell
- **1**: Fresh orange
- **2**: Rotten orange

A rotten orange at position $[i][j]$ can rot other fresh oranges at indexes $[i-1, j]$, $[i+1, j]$, $[i, j-1]$, $[i, j+1]$ (up, down, left and right) in unit time.

Your task is to calculate the minimum time required to rot all oranges. If it's not possible to rot all the fresh oranges, print **-1**.

Input Format:

- The first line of input contains two integers N and M — number of rows and columns of the grid.
- The next N lines contain M space-separated integers representing the grid.

Output Format:

- Print a single integer representing the minimum time to rot all oranges; if not all oranges can be rotted, print **-1**.

Program:

`rotten_oranges.c`

```
#include <stdio.h>

#define MAX 1005

int grid[MAX][MAX];
int visited[MAX][MAX];
int qx[MAX*MAX], qy[MAX*MAX], qt[MAX*MAX];
int front = 0, rear = 0;
int n, m;

int dr[4] = {-1, 1, 0, 0};
int dc[4] = {0, 0, -1, 1};

void enqueue(int x, int y, int t) {
    qx[rear] = x;
    qy[rear] = y;
    qt[rear] = t;
    rear++;
}

int main() {
    scanf("%d %d", &n, &m);
    int fresh = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d", &grid[i][j]);
            if (grid[i][j] == 1) fresh++;
            if (grid[i][j] == 2) {
                enqueue(i, j, 0);
                visited[i][j] = 1;
            }
        }
    }
    int time = 0, rotted = 0;
    while (front < rear) {
        int x = qx[front];
        int y = qy[front];
        int t = qt[front];
        front++;
        time = t;
        for (int k = 0; k < 4; k++) {
            int nx = x + dr[k];
            int ny = y + dc[k];
            if (nx >= 0 && nx < n && ny >= 0 && ny < m && !visited[nx][ny] &&
grid[nx][ny] == 1) {
                visited[nx][ny] = 1;
                enqueue(nx, ny, t + 1);
                rotted++;
            }
        }
    }
    if (rotted == fresh) printf("%d\n", time);
    else printf("-1\n");
```

```
    return 0;
}
```

Output:

Test case - 1
User Output
3 3
2 1 1
1 1 0
0 1 1
4

Test case - 2
User Output
3 3
2 1 1
0 0 0
1 1 1
-1

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 9

Experiment - 9: ***Minimum Edit Distance Between Two Strings***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
Write a C program to find the minimum number of operations required to convert one string (**str1**) into another string (**str2**) using the following operations:

- Insert a character
- Remove a character
- Replace a character

All operations have equal cost. You are given two strings, **str1** and **str2**. Your program should compute the minimum number of edit operations to convert **str1** into **str2**.

Input Format:

- The first line contains a string, **str1**.
- The second line contains a string, **str2**.

Output Format:

- Print a single integer representing the **minimum number of edit operations** required.

Program:

`edit_distance.c`

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

#define MAX 1005

int min(int a, int b) {
    return a < b ? a : b;
}

int main() {
    char str1[MAX], str2[MAX];
    scanf("%s", str1);
    scanf("%s", str2);

    int n = strlen(str1);
    int m = strlen(str2);

    int dp[MAX][MAX];

    for (int i = 0; i <= n; i++) dp[i][0] = i;
    for (int j = 0; j <= m; j++) dp[0][j] = j;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (str1[i-1] == str2[j-1]) dp[i][j] = dp[i-1][j-1];
            else dp[i][j] = 1 + min(dp[i-1][j-1], min(dp[i-1][j], dp[i][j-1]));
        }
    }

    printf("%d\n", dp[n][m]);
    return 0;
}
```

Output:

Test case - 1
User Output
apple
apple
0

Test case - 2
User Output
cat
cut
1

Test case - 3

User Output
bat
cats
2

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 10

Experiment - 10: ***Minimum Path Sum***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
Write a C program to find the minimum path sum from the top-left to the bottom-right corner of a given $N \times M$ grid.

- Each cell in the grid contains a non-negative integer.
- You can only move **either down or right** at any point in time.

Input Format:

- The first line contains two integers, N and M, representing the number of rows and columns in the grid.
- The next N lines contain M space-separated non-negative integers representing the grid.

Output Format:

- Print a single integer — the minimum path sum from the top left to the bottom-right of the grid.

Note: Refer to the visible test cases for a better understanding.

Program:

`minimum_path_sum.c`

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

#define min(a,b) ((a < b) ? a : b)

int main(){
        int rows, cols;
        scanf("%d %d", &rows, &cols);

        int grid[rows] [cols];
        for(int i =0; i < rows; i++){
                for(int j = 0; j<cols;j++){
                        scanf("%d", &grid[i][j]);
                }
        }
        int dp[rows][cols];

        dp[0][0] = grid[0][0];
        for(int j =1; j <cols; j++){
                dp[0][j] = dp[0] [j -1] + grid[0][j];
        }
        for(int i =1; i <rows;i++){
                dp[i][0] = dp[i-1][0] + grid[i][0];
        }

        for (int i =1; i<rows; i++){
                for(int j =1; j <cols; j++){
                        dp[i][j] = grid[i][j] + min(dp[i -1][j], dp[i][j -1]);
                }
        }

        printf("%d\n", dp[rows -1] [cols -1]);
        return 0;

}
```

Output:

Test case - 1
User Output
2 2
1 2
1 1
3

Test case - 2
User Output
1 4
1 2 3 4
10

Test case - 3
User Output
4 4
1 2 3 4
4 8 2 1
1 5 3 1
6 2 9 5
15

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 11

Experiment - 11: ***Smallest Number After Removing k Digits***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
Write a C program that takes a string **num** representing a non-negative integer and an integer k and returns the smallest possible integer after removing k digits from num.
You must ensure the result does not contain leading zeroes unless the result is "**0**".

Input Format:

- The first line contains a string representing the non-negative integer **num**.
- The second line contains an integer k.

Output Format:

- Print the smallest possible integer after removing k digits.

Program:

`remove_k_digits.c`

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

void removeKdigits(char *num, int k){
        int n = strlen(num);
        char stack[n];
        int top = -1;

         for (int i =0; i < n; i++){
                 while (k > 0 && top >= 0 && stack[top] > num[i]){
                         top--;
                         k--;
                 }
                 stack[++top] = num[i];
         }
        top -=k;

        int start = 0;
        while (start <= top && stack[start] == '0'){
                start++;

        }

        if (start>top){
                printf("0\n");
                return;
        }
        for (int i =start; i<= top; i++){
                printf("%c", stack[i]);
        }

        printf("\n");
}
int main(){
        char num[1000];
        int k;
        scanf("%s",num);
        scanf("%d", &k);

        removeKdigits(num, k);
        return 0;

}
```

Output:

Test case - 1
User Output
1432219
3
1219

Test case - 2
User Output
10
2
0

Test case - 3
User Output
9621458
2
21458

Result:

Thus the above program is executed successfully and the output has been verified

Experiment - 12

Experiment - 12: ***Unique Paths in a Grid***

Name: Tejasvi Raj Singh
UUID: 2303051051372
Date of performance:

Aim:
There is a robot on an $m \times n$ grid. The robot is initially located at the top-left corner (i.e., grid$[0][0]$). The robot tries to move to the bottom-right corner (i.e., grid $[m-1][n-1]$). The robot can only move either down or right at any point in time.
Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

Input Format:

- The first line of input contains an integer m (number of rows).
- The second line of input contains an integer n (number of columns).

Output Format:

- Print a single integer representing the number of unique paths the robot can take to reach the bottom-right corner.

Program:

unique_paths.c

```
#include<stdio.h>
int grid(int m,int n, int i, int j){
        if(i ==m -1 || j==n-1){
                return 1;

        }
        else {
                return grid(m,n,i,j+1) + grid(m,n,i+1,j);

        }
}
void main(){
        int n, m;
        scanf("%d%d", &n,&m);
        printf("%d", grid(m,n,0,0));
}
```

Output:

Test case - 1
User Output
2
2
2

Test case - 2

User Output
3
2
3

Result:

Thus the above program is executed successfully and the output has been verified