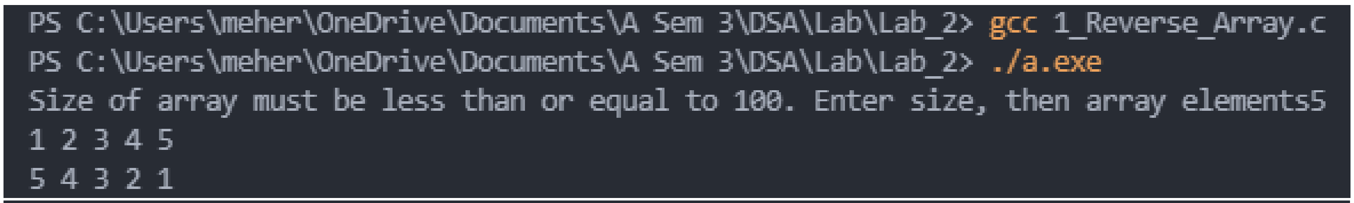
LAB SUBMISSION – 2

Meher Shrishti Nigam – 20BRS1193

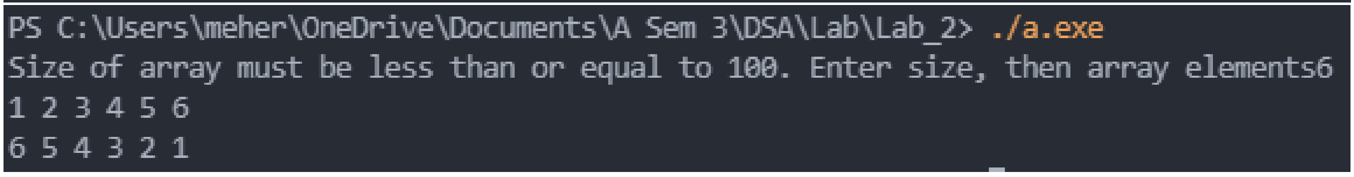
1- Reverse the array of given size ‘n’ by using the temporary variable.

Solution in C

```c
#include <stdio.h>

void reverseArray(int arr[], int start, int end)
{
    end--;
    while (start < end)
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

int main()
{
    int n;
    printf("Size of array must be less than or equal to 100. Enter size, then array elements");
    scanf("%d", &n);
    int arr[100];
    for(int i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    reverseArray(arr, 0, n);
    for(int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
}
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> gcc 1_Reverse_Array.c
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
Size of array must be less than or equal to 100. Enter size, then array elements5
1 2 3 4 5
5 4 3 2 1
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
Size of array must be less than or equal to 100. Enter size, then array elements6
1 2 3 4 5 6
6 5 4 3 2 1
```

Concepts:

Reverse of an array means the first element is at the last, the second element is at the second last… and so on. So, we use two counters, *start* and *end* that initially contain the start and end indices of the array respectively. Using a temp integer, we swap the values in the array at *start* and *end.* We do this until end > start, at which point we have reached the middle of the array and there are no more variables to swap.

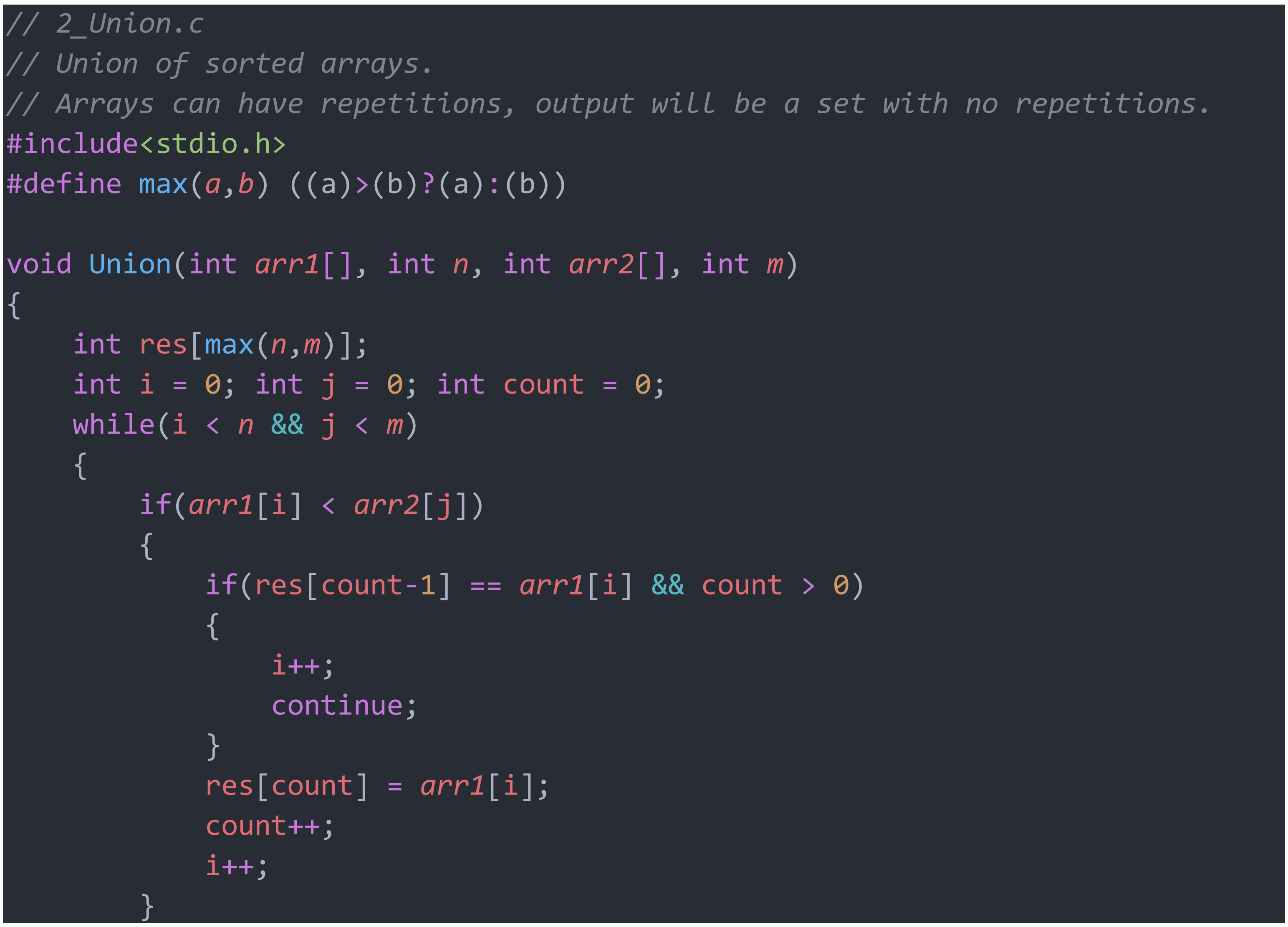
Alternative Methods:

1. We can use a single counter, i, and repeat the swapping till i = n/2, where n is size of array.
2. Create another array of same size and copy all the elements in reverse order. (from n-1 to 0).

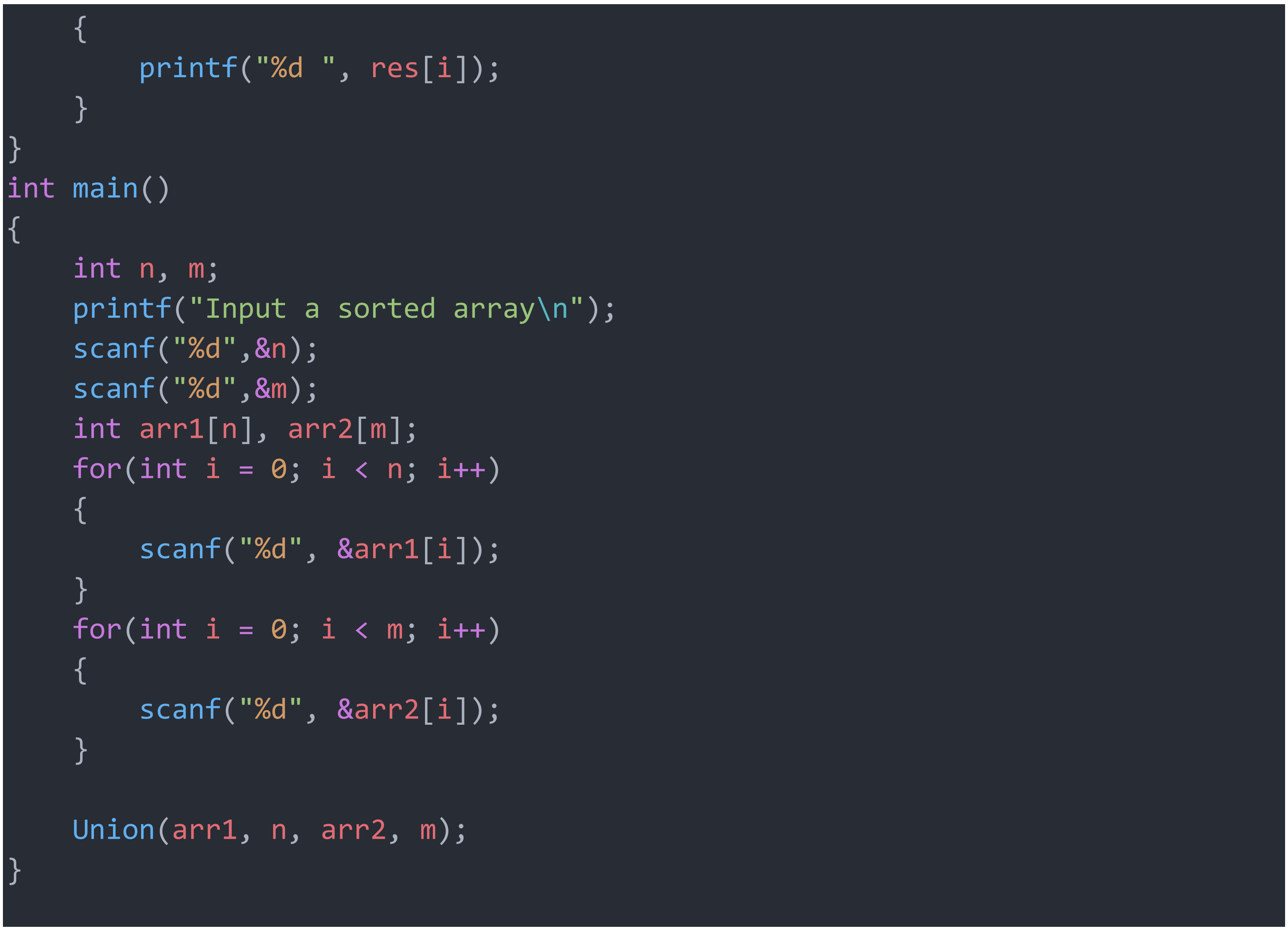
2- Write a program to print union and intersection of two array of ‘n’ dimension.

These solutions (in C) work for sorted arrays with repetitions as well.

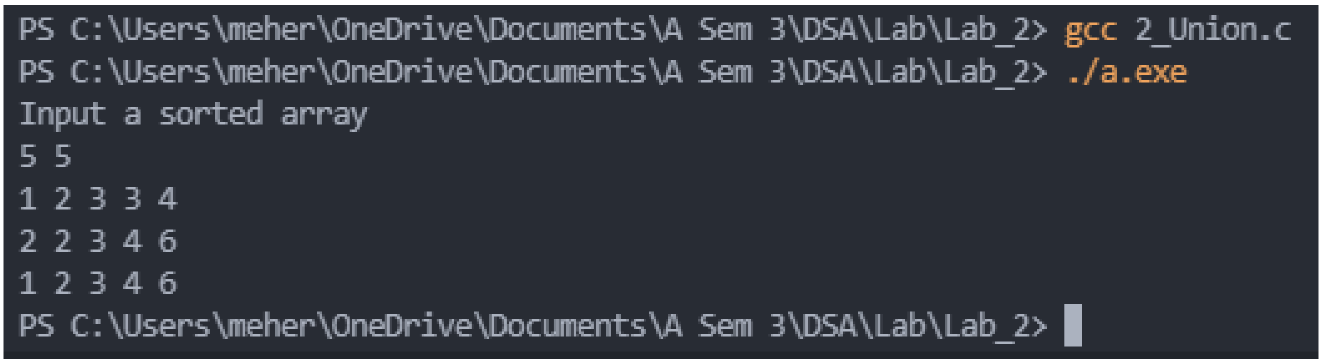
Union –



```c
// 2_Union.c
// Union of sorted arrays.
// Arrays can have repetitions, output will be a set with no repetitions.
#include<stdio.h>
#define max(a,b) ((a)>(b)?(a):(b))

void Union(int arr1[], int n, int arr2[], int m)
{
    int res[max(n,m)];
    int i = 0; int j = 0; int count = 0;
    while(i < n && j < m)
    {
        if(arr1[i] < arr2[j])
        {
            if(res[count-1] == arr1[i] && count > 0)
            {
                i++;
                continue;
            }
            res[count] = arr1[i];
            count++;
            i++;
        }
        else if(arr1[i] > arr2[j])
        {
            if(res[count - 1] == arr2[j] && count > 0)
            {
                j++;
                continue;
            }
            res[count] = arr2[j];
            count++;
            j++;
        }
        else
        {
            if(res[count-1] == arr1[i] && count > 0)
            {
                i++; j++;
                continue;
            }
            res[count] = arr1[i];
            count++;
            i++; j++;
        }
    }
    while(i < n)
    {
        if(res[count-1] == arr1[i] && count > 0)
        {
            i++;
            continue;
        }
        res[count] = arr1[i];
        i++; count++;
    }
    while(j < m)
    {
        if(res[count - 1] == arr2[j] && count > 0)
        {
            j++;
            continue;
        }
        res[count] = arr2[j];
        j++; count++;
    }

    for(i = 0; i < count; i++)
```



```c
    {
        printf("%d ", res[i]);
    }
}
int main()
{
    int n, m;
    printf("Input a sorted array\n");
    scanf("%d",&n);
    scanf("%d",&m);
    int arr1[n], arr2[m];
    for(int i = 0; i < n; i++)
    {
        scanf("%d", &arr1[i]);
    }
    for(int i = 0; i < m; i++)
    {
        scanf("%d", &arr2[i]);
    }

    Union(arr1, n, arr2, m);
}
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> gcc 2_Union.c
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
Input a sorted array
5 5
1 2 3 3 4
2 2 3 4 6
1 2 3 4 6
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2>
```

Concepts:

Initially i and j are zero. We also initialize count to zero, where count is the number of elements in the union set. We compare the value of first two elements in the two arrays.

- If the first array has the smaller number, we put the first element of the into our result array and increment i and count.
- If the first array has the greater number, we put the first element of the into our result array and increment j and count.
- If they are equal, we put that value into the result array and increment i, j and count.

We keep doing this in a loop till the condition (i < n and j < m) is satisfied. Then we break out of the loop fill in the remaining elements, in the case one of the arrays has leftover elements. Here, a check has also been implemented, that checks if the current last element is equal to the element that is going to be addded, if so it does not add that element to the array.

This **will only work for sorted arrays**. So if the array is not sorted, we should sort it first.

```
Array 1: 1 2 3 3 4

Array 2: 2 2 3 4 6

Union: 1 2 3 4 6
```

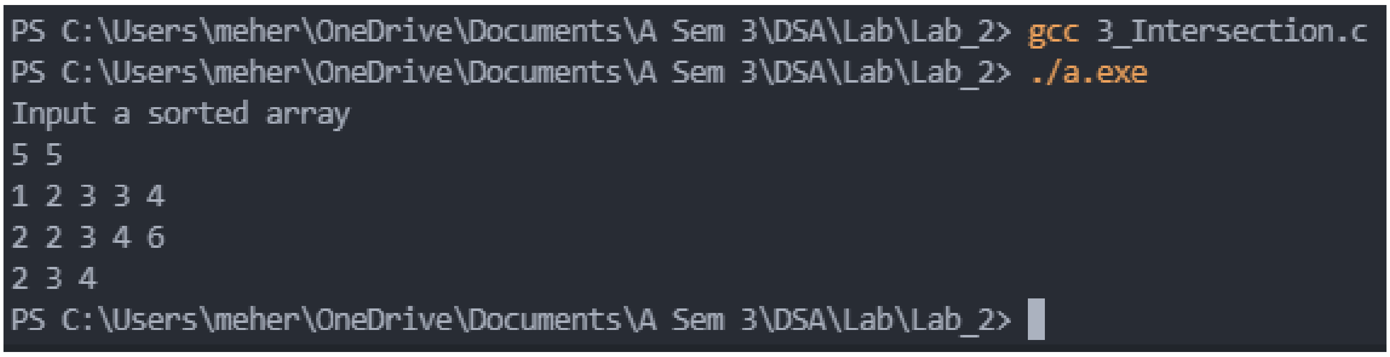
Alternative Methods:

1. Copy the first array into the resultant array (remove repetitions). Then for every element of the second array, traverse the resultant array to check if it already exists in it. If it doesn't add the element at the end.

This can very easily be done in C++ using std::set_union. (Abstraction).

Intersection –

```c
// 3_Intersection.c
// Intersection of a sorted array.
// Arrays can have repetitions, output will be a set with no repetitions.
#include<stdio.h>
#define max(a,b) ((a)>(b)?(a):(b))

void Intersection(int arr1[], int n, int arr2[], int m)
{
    int res[max(n,m)];
    int i = 0; int j = 0; int count = 0;
    while(i < n && j < m)
    {
        if(arr1[i] == arr2[j])
        {
            if(res[count-1] == arr1[i] && count > 0)
            {
                i++; j++;
                continue;
            }
            res[count] = arr1[i];
            count++;
            i++; j++;
        }
        else if(arr1[i] < arr2[j])
        {
            i++;
        }
        else
        {
            j++;
        }
    }
    for(i = 0; i < count; i++)
    {
        printf("%d ", res[i]);
    }
}
int main()
{
    int n, m;
    printf("Input a sorted array\n");
    scanf("%d",&n);
    scanf("%d",&m);
    int arr1[n], arr2[m];
    for(int i = 0; i < n; i++)
    {
        scanf("%d", &arr1[i]);
    }
    for(int i = 0; i < m; i++)
    {
        scanf("%d", &arr2[i]);
    }

    Intersection(arr1, n, arr2, m);
}
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> gcc 3_Intersection.c
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
Input a sorted array
5 5
1 2 3 3 4
2 2 3 4 6
2 3 4
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2>
```

Concepts:

Initially i and j are zero. We also initialize count to zero, where count is the number of elements in the intersection set. We compare the value of first two elements in the two arrays.

- If the first array has the smaller number, we increment i.
- If the first array has the greater number, we increment j.
- If they are equal, we put that value into the result array and increment i, j and count.

We keep doing this in a loop till the condition (i < n and j < m) is satisfied. Then we break out of the loop. Here, a check has also been implemented, that checks if the current last element is equal to the element that is going to be addded, if so it does not add that element to the array.

This **will only work for sorted arrays**. So if the array is not sorted, we should sort it first.

```
Array 1: 1 2 3 3 4

Array 2: 2 2 3 4 6

Intersection: 2 3 4
```

Alternative Methods:

1. Check against every element of the first array every element of the second array, if an element exists in both, add it to the “resultant set” array if not already present.

This can very easily be done in C++ using std::set_intersection. (Abstraction).

3- Consider two-dimensional array of MXN dimension. Insert the element in matrix and perform the following operations on array

a) Display the transpose of array

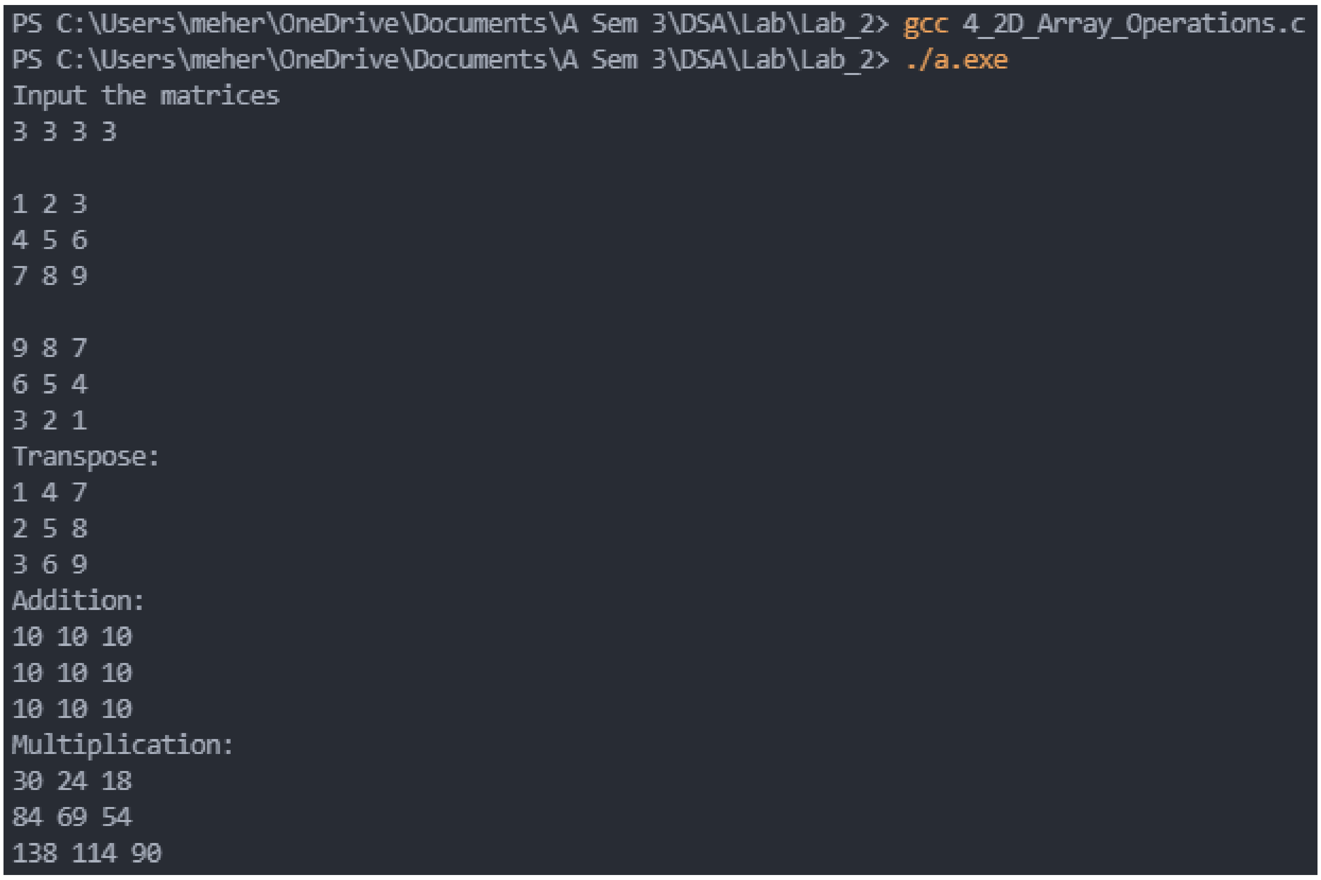
b) Calculate and display the addition of two 2D Matrix

c) Calculate and display the multiplication of 2D array

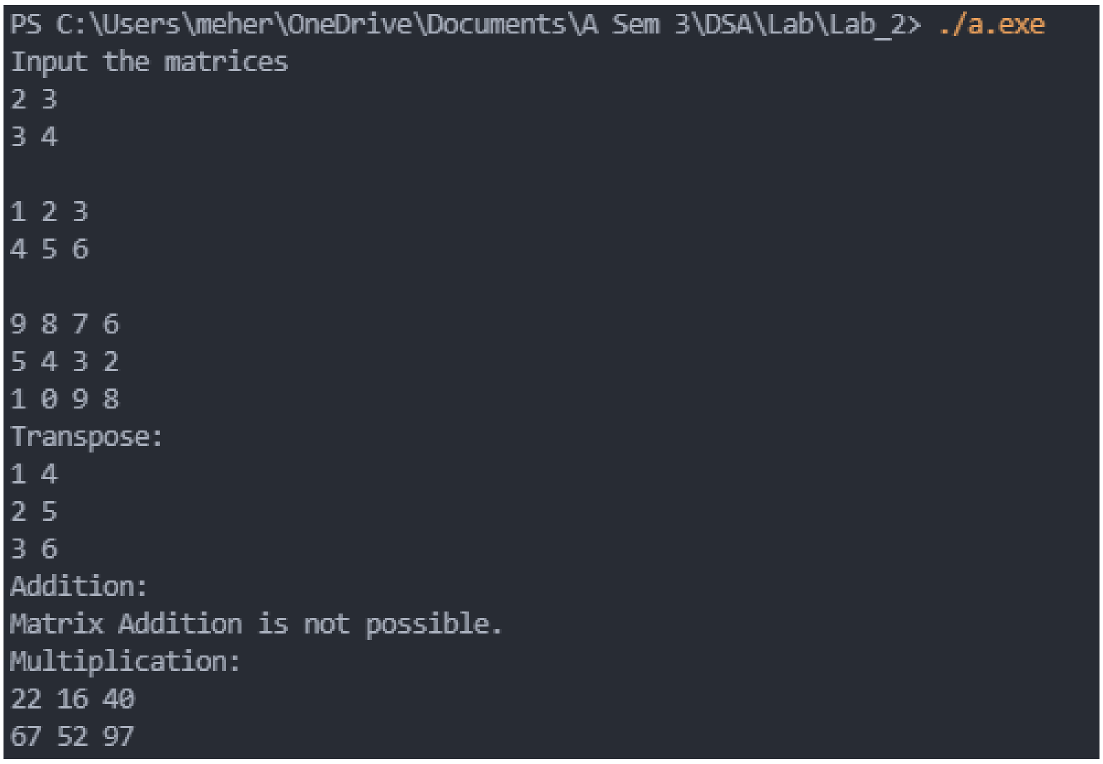
Solution in C –

```c
// 4_2D_Array_Operations.c
#include <stdio.h>
#include <stdbool.h>

void Transpose(int r, int c, int * matrix, int * result) // Stores result in a separate matrix
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < c; j++)
        {
            *((result + j*r) + i) = *((matrix + i*c) + j);
        }
    }
}


bool Addition(int r1, int c1, int r2, int c2, int * matrix1, int * matrix2, int * result)
{
    if(r1 != r2 || c1 != c2)
    {
        printf("Matrix Addition is not possible.\n");
        return false;
    }
    for(int i = 0; i < r1; i++)
    {
        for(int j = 0; j < c1; j++)
        {
            *((result + i*c1) + j) = *((matrix1 + i*c1) + j) + *((matrix2 + i*c1) + j);
        }
    }
    return true;
}
```

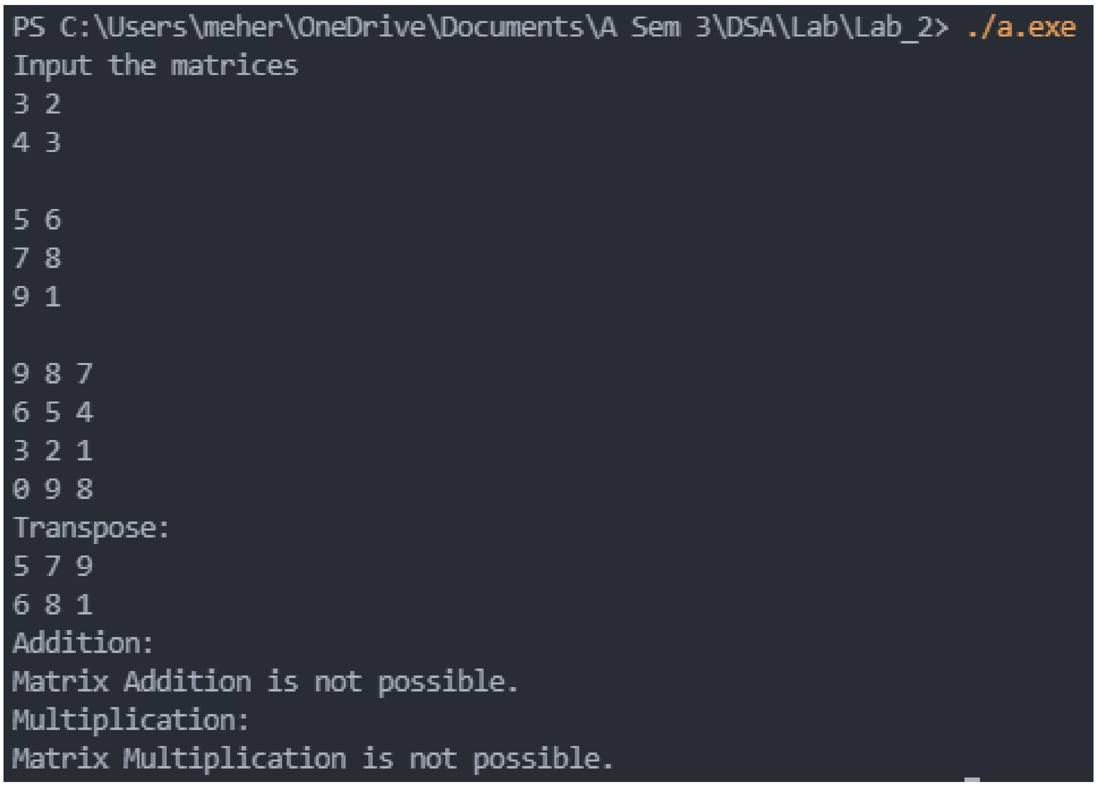
```c
bool Multiplication(int r1, int c1, int r2, int c2, int * matrix1, int * matrix2, int * result)
{
    if(c1 != r2)
    {
        printf("Matrix Multiplication is not possible.\n");
        return false;
    }

    for(int i = 0; i < r1; i++)
    {
        for(int j = 0; j < c2; j++)
        {
            *((result + i*c1) + j) = 0;
            for(int k = 0; k < c1; k++)
            {
                *((result + i*c1) + j) += ((*((matrix1 + i*c1) + k)) * (*((matrix2 + k*c2) + j)));
            }
        }
    }
    return true;
}


void display1(int r, int c, int * matrix)
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < c; j++)
            printf("%d ", *((matrix + i*c) + j));
        printf("\n");
    }
}

int main()
{
    // Getting two matrices as inputs
    printf("Input the matrices\n");
    int r1, c1;
    scanf("%d%d",&r1,&c1);
    int r2, c2;
    scanf("%d%d",&r2,&c2);
    int matrixA[r1][c1];
    int matrixB[r2][c2];
    for(int i = 0; i < r1; i++)
    {
        for(int j = 0; j < c1; j++)
        {
            scanf("%d", &matrixA[i][j]);
        }
    }
    for(int i = 0; i < r2; i++)
    {
        for(int j = 0; j < c2; j++)
        {
            scanf("%d", &matrixB[i][j]);
        }
    }

    // Transpose of Matrix
    // Lets find the transpose of the first matrix
    int matrixT[c1][r1]; // cols become rows and vice versa
    printf("Transpose: \n");
    Transpose(r1, c1, (int *)matrixA, (int *)matrixT);
    display1(c1, r1, (int *)matrixT);



// Matrix Addition
    int matrixC[r1][c1]; // Result of the addition will be stored here
    printf("Addition: \n");
    if(Addition(r1, c1, r2, c2, (int *)matrixA, (int *)matrixB, (int *)matrixC))
    {
        display1(r1, c1, (int *)matrixC);
    }

    // Matrix Multiplication
    int matrixM[r1][c2];
    printf("Multiplication: \n");
    if(Multiplication(r1, c1, r2, c2, (int *)matrixA, (int *)matrixB, (int *)matrixM))
    {
        display1(r1, c1, (int *)matrixM);
    }
}
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> gcc 4_2D_Array_Operations.c
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
Input the matrices
3 3 3 3

1 2 3
4 5 6
7 8 9

9 8 7
6 5 4
3 2 1
Transpose:
1 4 7
2 5 8
3 6 9
Addition:
10 10 10
10 10 10
10 10 10
Multiplication:
30 24 18
84 69 54
138 114 90
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
Input the matrices
2 3
3 4

1 2 3
4 5 6

9 8 7 6
5 4 3 2
1 0 9 8
Transpose:
1 4
2 5
3 6
Addition:
Matrix Addition is not possible.
Multiplication:
22 16 40
67 52 97
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
Input the matrices
3 2
4 3

5 6
7 8
9 1

9 8 7
6 5 4
3 2 1
0 9 8
Transpose:
5 7 9
6 8 1
Addition:
Matrix Addition is not possible.
Multiplication:
Matrix Multiplication is not possible.
```

Concepts

Transpose of a matrix is taking the rows of a matrix and making them the columns or vice versa. So, an element at position **r,c** where r is the row number and c is column number, is now at **c,r**. So we make a new matrix and olace the element at r,c in the original matrix at c,r in the new matrix.

Addition of two matrices is only possible when both matrices are of same dimensions. So we check that first. To add two matrices, we add the corressponding elements. So the r,c position in the resulting matrix will have the sum of the r,c element in the first matrix and the r,c element in the second matrix.

Multiplication of two matrices is only possible if the column of the first matrix is equal to the row of the second matrix. The result will have the same number of rows as the 1st matrix, and the same number of columns as the 2nd matrix. We multiply the corresponding elements of rows of the first matrix with the columns of the second matrix, and then we add them together, to get the elements of the resulting matrix.

4- Consider the following scenario for a square matrix

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

Once the square matrix is rotated

i) by 90 degree in a clockwise direction then the transformed matrix will be as follows

$$\begin{matrix} 7 & 4 & 1 \\ 8 & 5 & 2 \\ 9 & 6 & 3 \end{matrix}$$

ii) by 180 degree in a clockwise direction will lead to following

$$\begin{matrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{matrix}$$

iii) by 90 degree in a anti-clockwise direction then the transformed matrix will be as follows

$$\begin{matrix} 3 & 6 & 9 \\ 2 & 5 & 8 \\ 1 & 4 & 7 \end{matrix}$$

ii) Matrix rotation by 180 degree in anti-clockwise direction will lead to following

$$\begin{matrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{matrix}$$

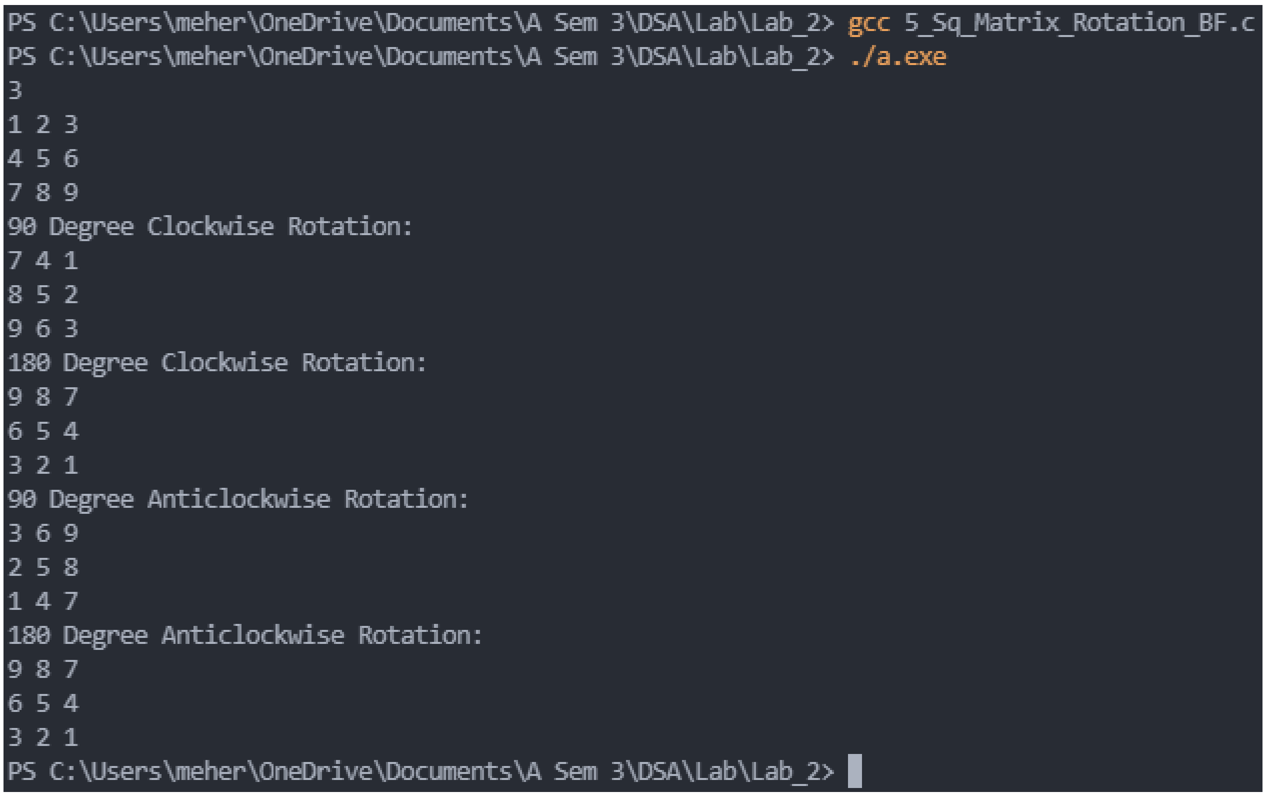
Write a program for the above-mentioned scenarios. (Use both brute force and in-place approach):

Soultion in C:

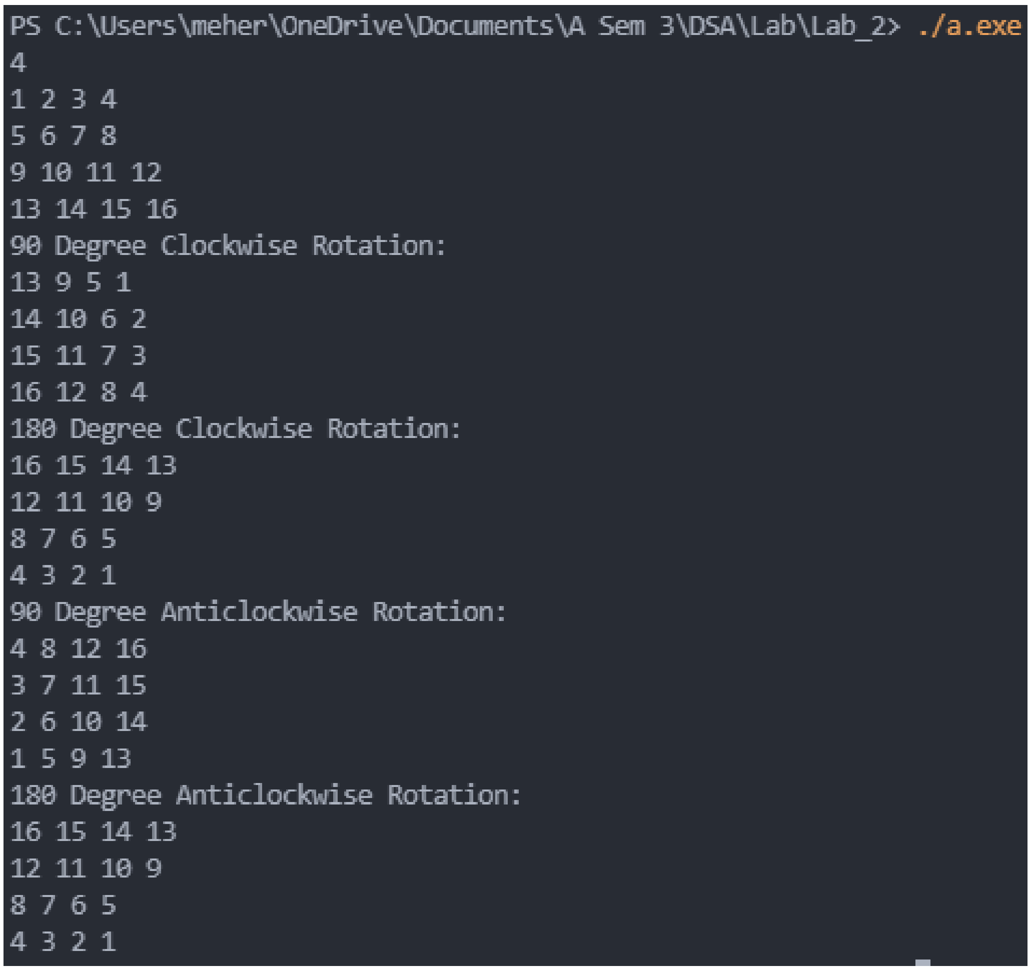
Brute Force Approach:

```
// 5_Sq_Matrix_Rotation_BF.c
// Matrix rotation by given degree and angle by brute force method


#include <stdio.h>


void Clockwise_90(int r, int * matrix, int * result)
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < r; j++)
        {
            *((result + j*r) + (r-i-1)) = *((matrix + i*r) + j);
        }
    }
}
void Clockwise_180(int r, int * matrix, int * result)
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < r; j++)
        {
            *((result + (r-i-1)*r) + (r-j-1)) = *((matrix + i*r) + j);
        }
    }
}

void Anti_Clockwise_90(int r, int * matrix, int * result)
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < r; j++)
        {
            *((result + (r-j-1)*r) + i) = *((matrix + i*r) + j);
        }
    }
}
// Anti-Clockwise 180 is the same as Clockwise 180
void Anti_Clockwise_180(int r, int * matrix, int * result)
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < r; j++)
        {
            *((result + (r-i-1)*r) + (r-j-1)) = *((matrix + i*r) + j);
        }
    }
}
void display(int r, int c, int * matrix)
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < c; j++)
```

```c
            printf("%d ", *((matrix + i*c) + j));
        printf("\n");
    }
}
int main()
{
    int r;
    scanf("%d",&r);
    int matrixA[r][r];
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < r; j++)
        {
            scanf("%d", &matrixA[i][j]);
        }
    }
    int matrixB[r][r];
    int matrixC[r][r];
    int matrixD[r][r];
    int matrixE[r][r];

    printf("90 Degree Clockwise Rotation: \n");
    Clockwise_90(r, (int * )matrixA, (int *) matrixB);
    display(r, r, (int *) matrixB);

    printf("180 Degree Clockwise Rotation: \n");
    Clockwise_180(r, (int * )matrixA, (int *) matrixC);
    display(r, r, (int *) matrixC);

    printf("90 Degree Anticlockwise Rotation: \n");
    Anti_Clockwise_90(r, (int * )matrixA, (int *) matrixD);
    display(r, r, (int *) matrixD);

    printf("180 Degree Anticlockwise Rotation: \n");
    Anti_Clockwise_180(r, (int * )matrixA, (int *) matrixE);
    display(r, r, (int *) matrixE);
}
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> gcc 5_Sq_Matrix_Rotation_BF.c
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
3
1 2 3
4 5 6
7 8 9
90 Degree Clockwise Rotation:
7 4 1
8 5 2
9 6 3
180 Degree Clockwise Rotation:
9 8 7
6 5 4
3 2 1
90 Degree Anticlockwise Rotation:
3 6 9
2 5 8
1 4 7
180 Degree Anticlockwise Rotation:
9 8 7
6 5 4
3 2 1
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2>
```

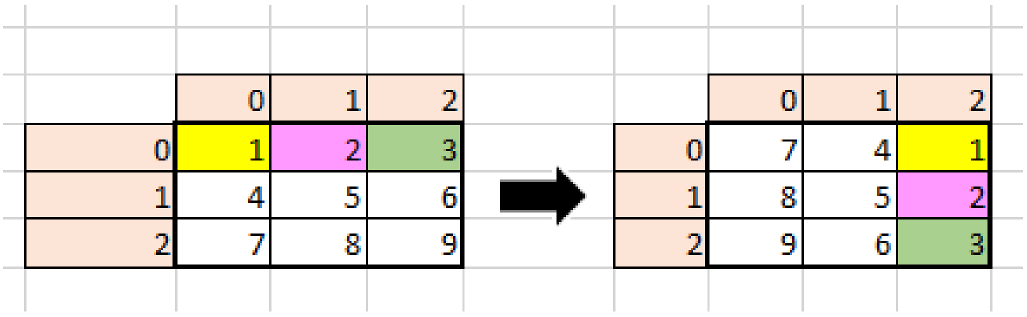


```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
4
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
90 Degree Clockwise Rotation:
13 9 5 1
14 10 6 2
15 11 7 3
16 12 8 4
180 Degree Clockwise Rotation:
16 15 14 13
12 11 10 9
8 7 6 5
4 3 2 1
90 Degree Anticlockwise Rotation:
4 8 12 16
3 7 11 15
2 6 10 14
1 5 9 13
180 Degree Anticlockwise Rotation:
16 15 14 13
12 11 10 9
8 7 6 5
4 3 2 1
```

Concepts

Brute force method to rotate matrices includes analyzing the position fo the elements of the original matrix and the rotated matrix.

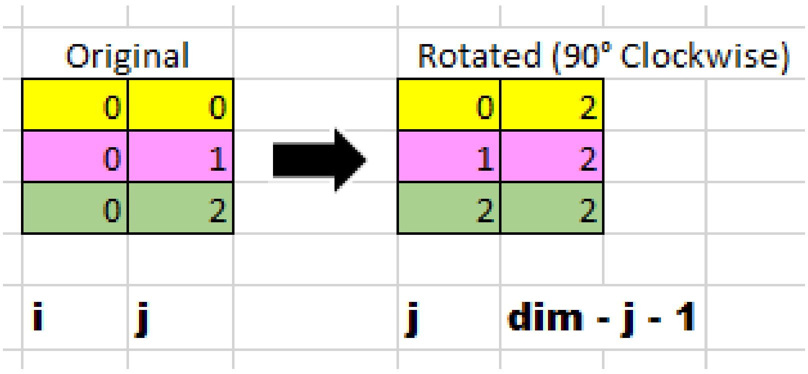
Taking the example of 90 degree rotation in clockwise direction –



	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

	0	1	2
0	7	4	1
1	8	5	2
2	9	6	3

Looking at how the postions of the first row changed **we can deduce the logic in which the numbers have to be shifted –**



Original	
0	0
0	1
0	2
i	**j**

Rotated (90° Clockwise)	
0	2
1	2
2	2
j	**dim - j - 1**

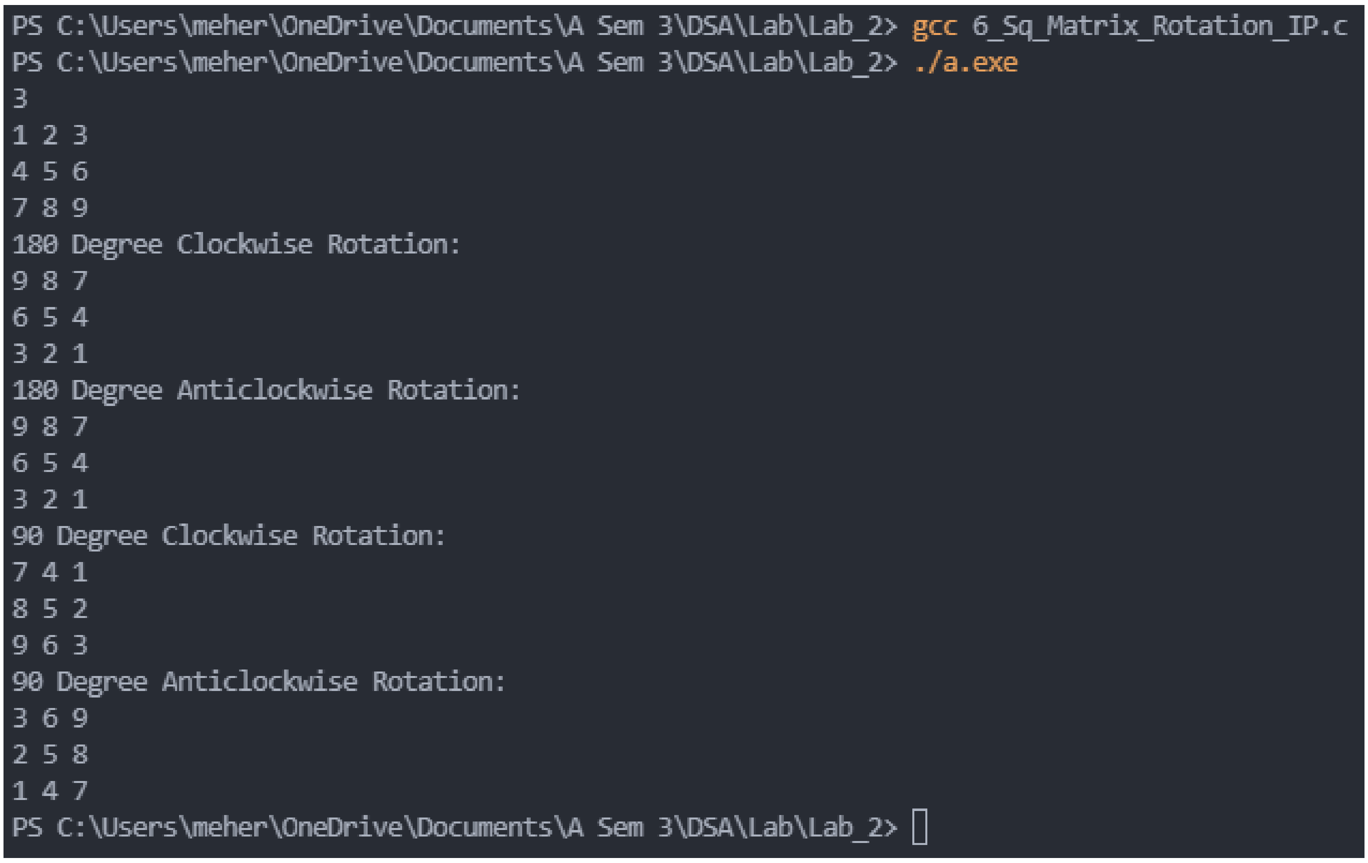
Other rotations have been deduced in a similar way. 180 degree rotation in clockwise and anti-clockwise direction are the same.

<u>In-Place Approach:</u>

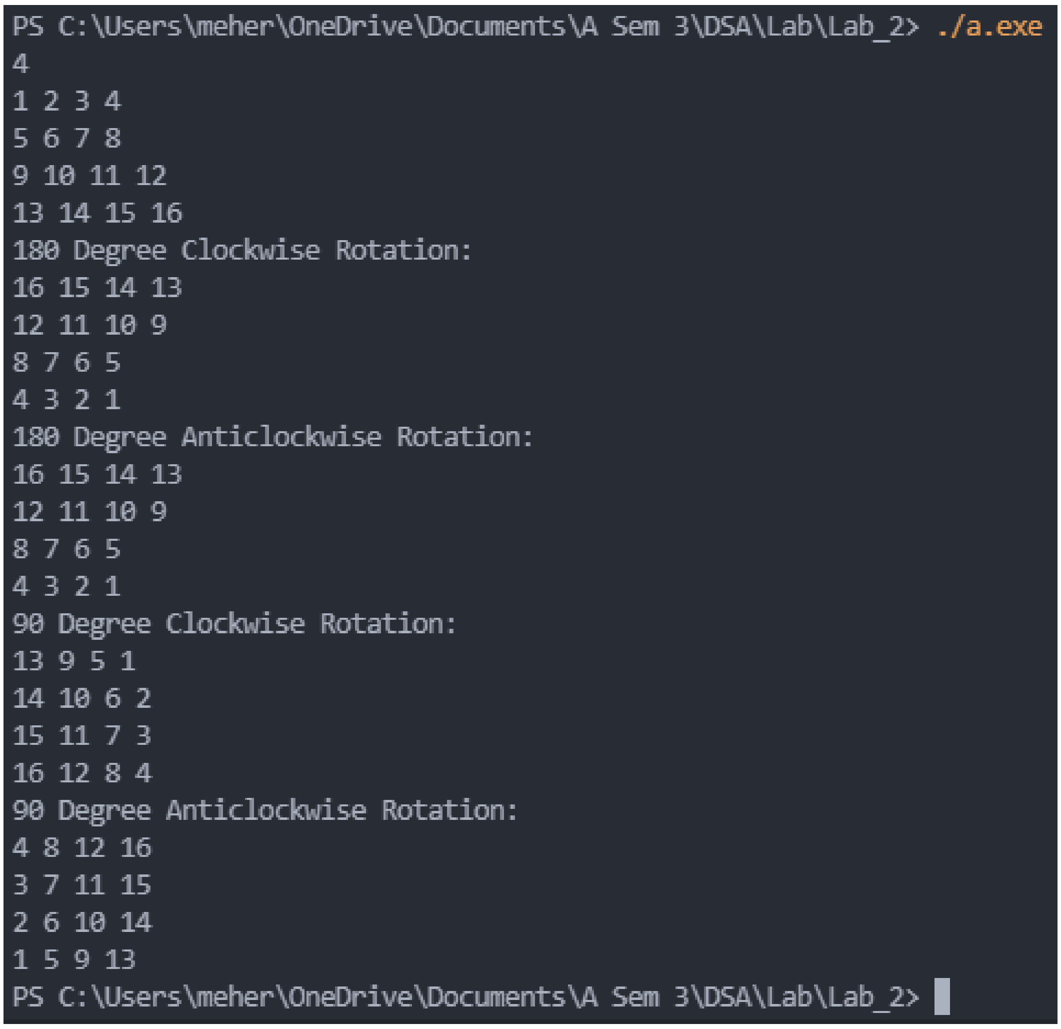
```c
// 6_Sq_Matrix_Rotation_IP.c
#include <stdio.h>
void display(int r, int c, int * matrix)
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < c; j++)
            printf("%d ", *((matrix + i*c) + j));
        printf("\n");
    }
}
void Transpose(int r, int c, int * matrix) // Stores result in a separate matrix
{
    int result1[r][r];
    int * result = (int *) result1;
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < c; j++)
        {
            *((result + j*r) + i) = *((matrix + i*c) + j);
        }
    }
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < c; j++)
        {
            *((matrix + i*c) + j) = *((result + i*r) + j);
        }
    }
}

void Clockwise_90(int r, int * result)
{
    int start = 0; int end = r - 1;
    while(start < end)
    {
        for(int i = 0; i < r; i++)
        {
            int temp = *((result + i*r) + start);
            *((result + i*r) + start) = *((result + i*r) + end);
            *((result + i*r + end)) = temp;
        }
        start++; end--;
    }
}

void Clockwise_180(int r, int * result)
{
    int start = 0; int end = r - 1;
    while(start <= end)
    {
        if(start == end)
        {
            for(int i = 0; i < (r/2); i++)
            {
                int temp = *((result + start*r) + i);
                *((result + start*r) + i) = *((result + start*r) + (r-1-i));
                *((result + start*r) + (r-1-i)) = temp;
            }
        }
        for(int i = 0; i < r; i++)
        {
            int temp = *((result + start*r) + i);
            *((result + start*r) + i) = *((result + end*r) + (r-1-i));
            *((result + end*r) + (r-1-i)) = temp;
        }
        start++; end--;
    }
}
```

```
void Anti_Clockwise_90(int r, int * result)
{
    int start = 0; int end = r - 1;
    while(start < end)
    {
        for(int i = 0; i < r; i++)
        {
            int temp = *((result + start*r) + i);
            *((result + start*r) + i) = *((result + end*r) + i);
            *((result + end*r) + i) = temp;
        }
        start++; end--;
    }
}

// Anti-Clockwise 180 is the same as Clockwise 180
void Anti_Clockwise_180(int r, int * result)
{
    int start = 0; int end = r - 1;
    while(start <= end)
    {
        if(start == end)
        {
            for(int i = 0; i < (r/2); i++)
            {
                int temp = *((result + start*r) + i);
                *((result + start*r) + i) = *((result + start*r) + (r-1-i));
                *((result + start*r) + (r-1-i)) = temp;
            }
        }
        for(int i = 0; i < r; i++)
        {
            int temp = *((result + start*r) + i);
            *((result + start*r) + i) = *((result + end*r) + (r-1-i));
            *((result + end*r) + (r-1-i)) = temp;
        }
        start++; end--;
    }
}

void Copy(int * arr, int * ans, int r)
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < r; j++)
        {
            *((ans + i*r) + j) = *((arr + i*r) + j);
        }
    }
}

int main()
{
    int r;
    scanf("%d",&r);
    int matrixA[r][r];
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < r; j++)
        {
            scanf("%d", &matrixA[i][j]);
        }
    }

    int matrixB[r][r];

    Copy((int *) matrixA, (int *)matrixB, r);
    // We copy matrix A to B, so that we can perform the different rotations without losing the original transposed matrix

    printf("180 Degree Clockwise Rotation: \n");
    Clockwise_180(r, (int *) matrixB);
    display(r, r, (int *) matrixB);

    Copy((int *) matrixA, (int *)matrixB, r);

    printf("180 Degree Anticlockwise Rotation: \n"); // Same as 180 Clockwise Rotation
    Anti_Clockwise_180(r, (int *) matrixB);
    display(r, r, (int *) matrixB);

    Transpose(r, r, (int *) matrixA); // We transpose the matrix A
    Copy((int *) matrixA, (int *)matrixB, r);
    //We copy transposed matrix A to B, so that we can perform the different rotations without losing the original transposed matrix

    // Rotating matrix inplace
    printf("90 Degree Clockwise Rotation: \n");
    Clockwise_90(r, (int *) matrixB);
    display(r, r, (int *) matrixB);
```

```c
    // Reseting matrix to original transposed matrix
    Copy((int *) matrixA, (int *)matrixB, r);

    printf("90 Degree Anticlockwise Rotation: \n");
    Anti_Clockwise_90(r, (int *) matrixB);
    display(r, r, (int *) matrixB);
}
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> gcc 6_Sq_Matrix_Rotation_IP.c
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
3
1 2 3
4 5 6
7 8 9
180 Degree Clockwise Rotation:
9 8 7
6 5 4
3 2 1
180 Degree Anticlockwise Rotation:
9 8 7
6 5 4
3 2 1
90 Degree Clockwise Rotation:
7 4 1
8 5 2
9 6 3
90 Degree Anticlockwise Rotation:
3 6 9
2 5 8
1 4 7
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> 
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
4
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
180 Degree Clockwise Rotation:
16 15 14 13
12 11 10 9
8 7 6 5
4 3 2 1
180 Degree Anticlockwise Rotation:
16 15 14 13
12 11 10 9
8 7 6 5
4 3 2 1
90 Degree Clockwise Rotation:
13 9 5 1
14 10 6 2
15 11 7 3
16 12 8 4
90 Degree Anticlockwise Rotation:
4 8 12 16
3 7 11 15
2 6 10 14
1 5 9 13
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2>
```

Concepts

In the in-place approach we try to reduce space complexity by rotating the matrices in-place, i.e, by not making additional matrices to shift elements into.

180 degree rotations can be done by flipping the rows (left to right and top to bottom) **which can be done using one loop and in-place**.

90 degree clockwise rotation is done by first taking the transpose of the matrix and then reversing the columns, (first column is the last column, second column is the second last column etc.).

90 degree anti - clockwise rotation is done by first taking the transpose of the matrix and then reversing the rows, (first row is the last row, second row is the second last row etc.).

Here, the 90 degree rotations are not truly in-place as the transpose of the matrix is not done in-place.

Alternative Methods:

1. Perform the transpose in-place while using in-place method.

5- Create Sparse Matrix as Triplet representation and perform following operation

i) Print new representation of sparse matrix

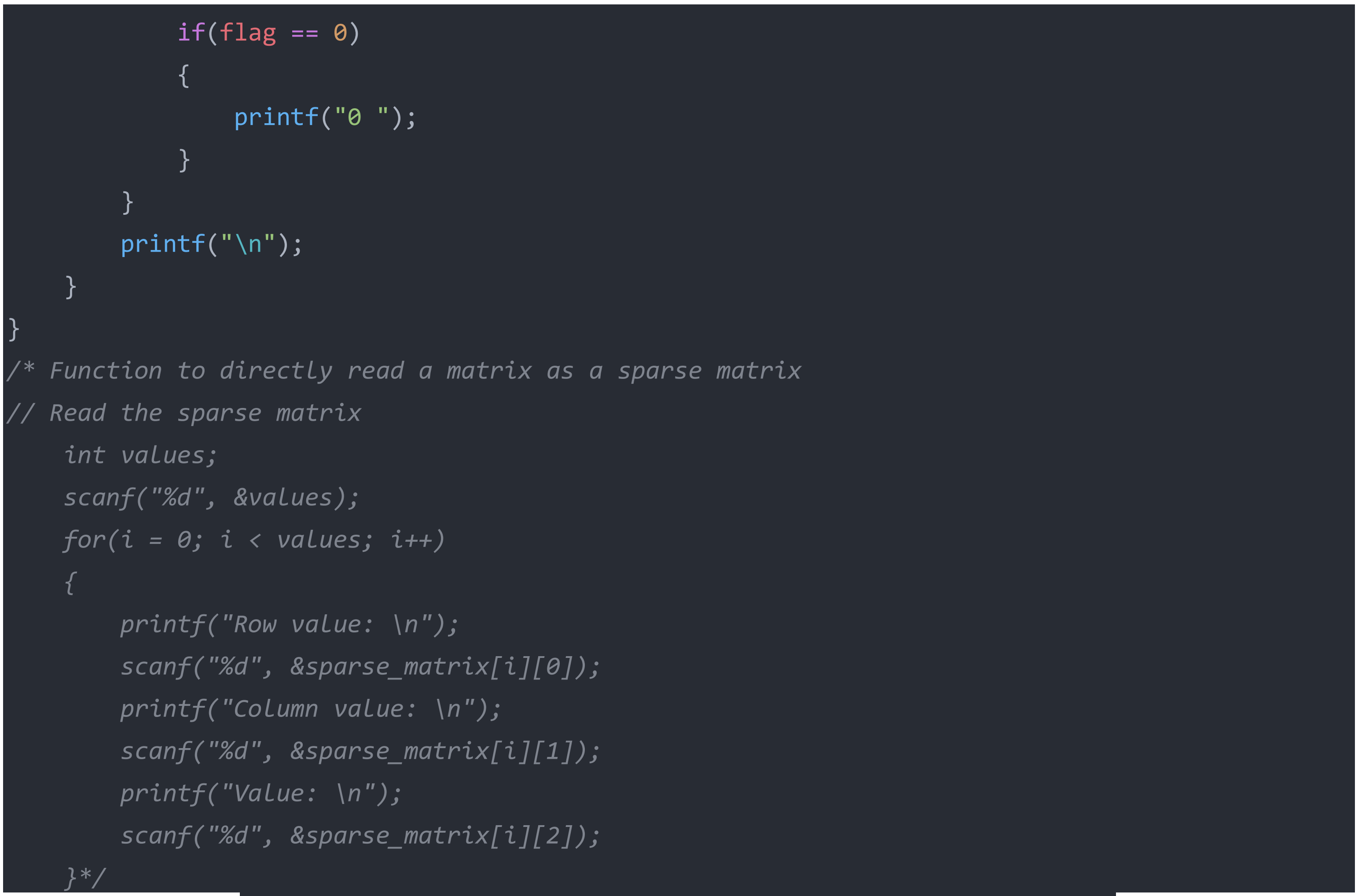
ii) Addition operation of 2 sparse matrix

iii) Transpose of sparse matrix

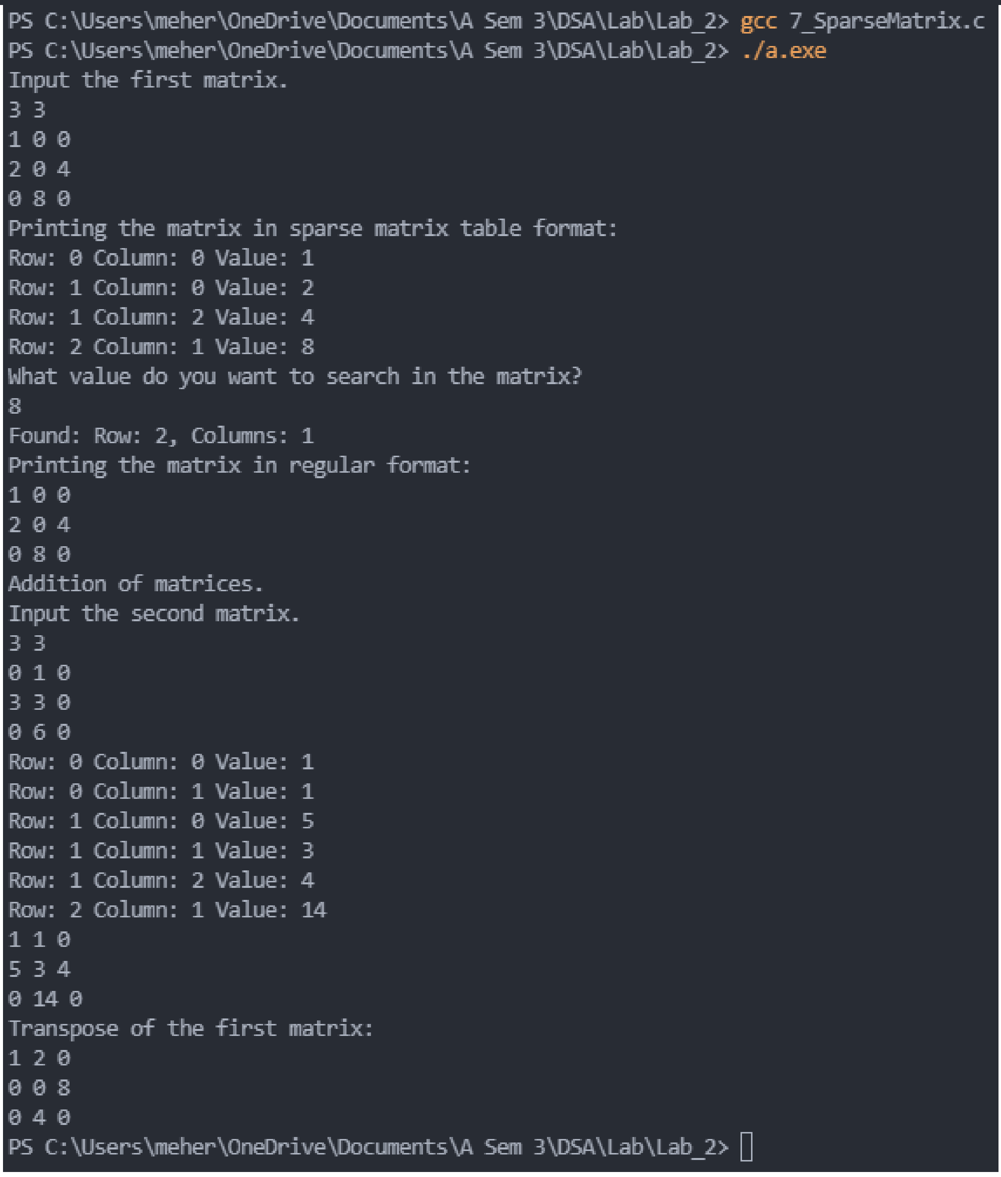
Solution in C:

```c
// 7_SparseMatrix.c
// In previous programs, I dealt with different operations as different functions.
// Although it would have been easy to do the same here, I put all the operations in the main
itself.
#include <stdio.h>
int main()
{
    int rows, columns, x, i, j, k;
    int values = 0;

    // Input the no. of its rows and columns of the sparse matrix
    printf("Input the first matrix.\n");
    scanf("%d",&rows);
    scanf("%d",&columns);
    int arr[rows][columns];
    for(i = 0; i < rows; i++)
    {
        for(j = 0; j < columns; j++)
        {
            scanf("%d", &arr[i][j]);
            if(arr[i][j] != 0)
            {
                values++;
            }
        }
    }

    int sparse_matrix[values][3];
    k = 0;
    for(i = 0; i < rows; i++)
    {
        for(j = 0; j < columns; j++)
        {
            if(arr[i][j] != 0)
            {
                sparse_matrix[k][0] = i;
                sparse_matrix[k][1] = j;
                sparse_matrix[k][2] = arr[i][j];
                k++;
            }
        }
    }
    // Output the sparse matrix in table form
    printf("Printing the matrix in sparse matrix table format:\n");
    for(i = 0; i < values; i++)
    {
        printf("Row: %d ",sparse_matrix[i][0]);
        printf("Column: %d ",sparse_matrix[i][1]);
        printf("Value: %d ",sparse_matrix[i][2]);
        printf("\n");
    }
  // Search for any integer in the sparse matrix (note: this doesn't consider the zeroes in the actual matrix)
    printf("What value do you want to search in the matrix?\n");
    scanf("%d",&x);
    int flag = 0;
    for(i = 0; i < values; i++)
    {
        if(sparse_matrix[i][2] == x)
        {
            printf("Found: Row: %d, Columns: %d \n",sparse_matrix[i][0],sparse_matrix[i][1]);
            flag = 1;
        }
    }
    if(flag == 0){printf("Not found\n");}
    // Output of the matrix in regular matrix format
    printf("Printing the matrix in regular format:\n");
    k = 0;
    for(i = 0; i < rows; i++)
    {
        for(j = 0; j < columns; j++)
        {
            if(sparse_matrix[k][0] == i && sparse_matrix[k][1] == j)
            {
                printf("%d ", sparse_matrix[k][2]);
                k++;
            }
            else
            {
                printf("0 ");
            }
        }
        printf("\n");
    }
    // Adding another matrix to the original matrix
    int rows2, columns2, values2;
    // Read the sparse matrix
    // Input the no. of its rows and columns of the sparse matrix
    printf("Addition of matrices. \nInput the second matrix.\n");
    scanf("%d",&rows2);
    scanf("%d",&columns2);
    int arr2[rows2][columns2];
    for(i = 0; i < rows2; i++)
    {
        for(j = 0; j < columns2; j++)
        {
            scanf("%d", &arr2[i][j]);
            if(arr2[i][j] != 0)
            {
                values2++;
            }
        }
    }
    int sparse_matrix2[values2][3];
    k = 0;
    for(i = 0; i < rows2; i++)
    {
        for(j = 0; j < columns2; j++)
        {
            if(arr2[i][j] != 0)
            {
                sparse_matrix2[k][0] = i;
                sparse_matrix2[k][1] = j;
                sparse_matrix2[k][2] = arr2[i][j];
                k++;
            }
        }
    }

    int sparse_matrix3[values + values2][3];
    int values3 = 0;
    i = 0; j = 0;
    if(rows != rows2 || columns != columns2)
    {
        printf("Addition of matrices is not possible.\n");
    }
    else
    {
        while(i < values && j < values2)
        {
            if((sparse_matrix[i][0] < sparse_matrix2[j][0]) || (sparse_matrix[i][0] == sparse_
matrix2[j][0] && sparse_matrix[i][1] < sparse_matrix2[j][1]))
            {
                sparse_matrix3[values3][0] = sparse_matrix[i][0];
                sparse_matrix3[values3][1] = sparse_matrix[i][1];
                sparse_matrix3[values3][2] = sparse_matrix[i][2];
                i++;
                values3++;
            }
            else if((sparse_matrix[i][0] > sparse_matrix2[j][0]) || (sparse_matrix[i][0] == sparse_matrix2[j][0] && sparse_matrix[i][1] > sparse_matrix2[j][1]))
            {
                sparse_matrix3[values3][0] = sparse_matrix2[j][0];
                sparse_matrix3[values3][1] = sparse_matrix2[j][1];
                sparse_matrix3[values3][2] = sparse_matrix2[j][2];
                j++;
                values3++;
            }
            else
            {
                sparse_matrix3[values3][0] = sparse_matrix[i][0];
                sparse_matrix3[values3][1] = sparse_matrix[i][1];
                sparse_matrix3[values3][2] = sparse_matrix[i][2] + sparse_matrix2[j][2];
                i++; j++;
                values3++;
            }
        }
        while(i < values)
        {
            sparse_matrix3[values3][0] = sparse_matrix[i][0];
            sparse_matrix3[values3][1] = sparse_matrix[i][1];
            sparse_matrix3[values3][2] = sparse_matrix[i][2];
            i++;
            values3++;
        }
        while(j < values2)
        {
            sparse_matrix3[values3][0] = sparse_matrix2[j][0];
            sparse_matrix3[values3][1] = sparse_matrix2[j][1];
            sparse_matrix3[values3][2] = sparse_matrix2[j][2];
            j++;
            values3++;
        }
        // Output the of the summation sparse matrix in table form
        for(i = 0; i < values3; i++)
        {
            printf("Row: %d ",sparse_matrix3[i][0]);
            printf("Column: %d ",sparse_matrix3[i][1]);
            printf("Value: %d ",sparse_matrix3[i][2]);
            printf("\n");
        }
```

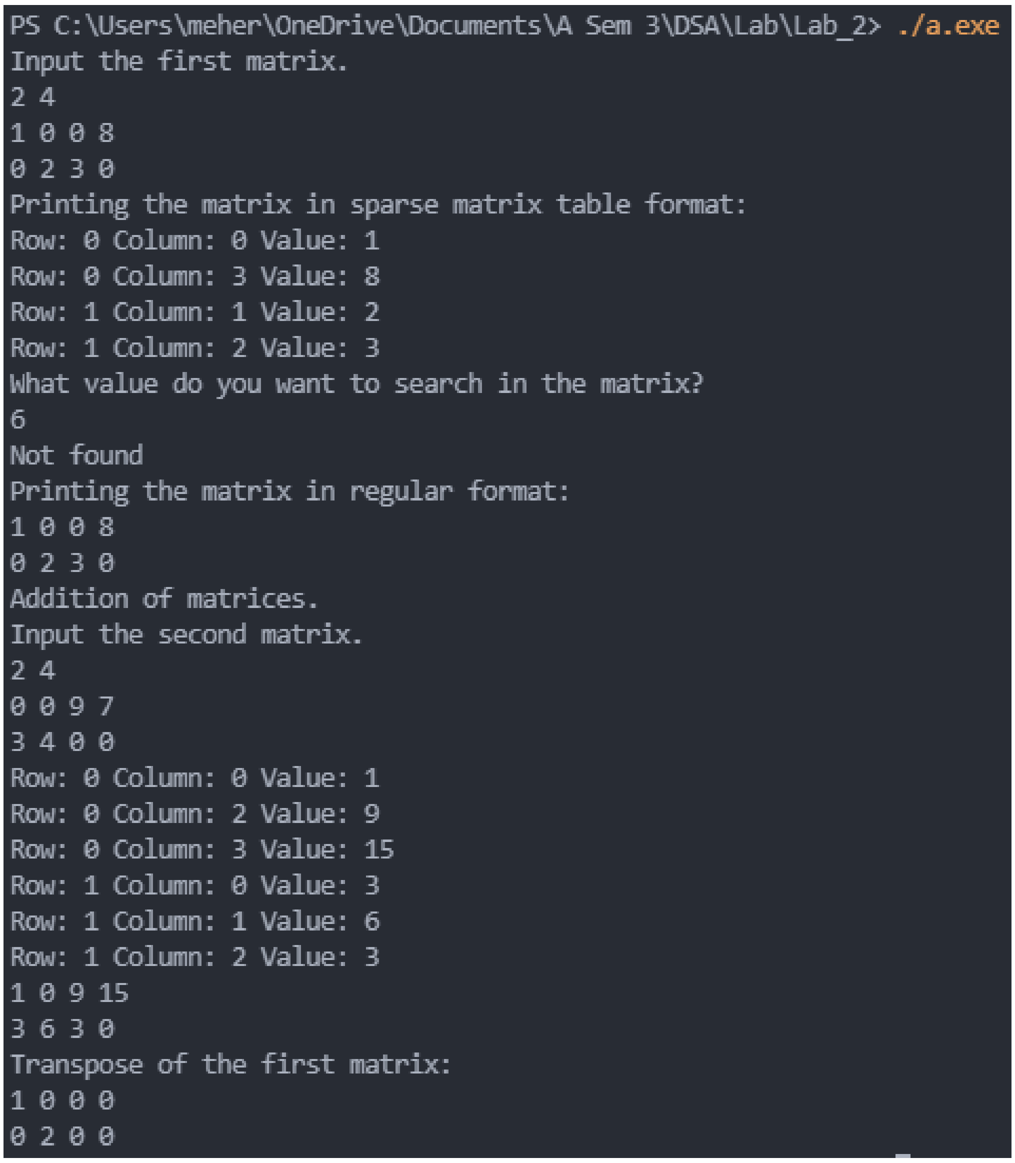
```
    // Output the of the summation sparse matrix in regular matrix format
    k = 0;
    for(i = 0; i < rows; i++)
    {
        for(j = 0; j < columns; j++)
        {
            if(sparse_matrix3[k][0] == i && sparse_matrix3[k][1] == j)
            {
                printf("%d ", sparse_matrix3[k][2]);
                k++;
            }
            else
            {
                printf("0 ");
            }
        }
        printf("\n");
    }

}
// Finding the transpose of the matrix
int sparse_matrix_transpose[values][3];
for(i = 0; i < values; i++)
{
    sparse_matrix_transpose[i][0] = sparse_matrix[i][1];
    sparse_matrix_transpose[i][1] = sparse_matrix[i][0];
    sparse_matrix_transpose[i][2] = sparse_matrix[i][2];
}
// Output of the transpose matrix in regular matrix format
flag = 0;
printf("Transpose of the first matrix: \n");
for(i = 0; i < rows; i++)
{
    for(j = 0; j < columns; j++)
    {
        flag = 0;
        for(k = 0; k < values; k++)
        {
            if(sparse_matrix_transpose[k][0] == i && sparse_matrix_transpose[k][1] == j)
            {
                printf("%d ", sparse_matrix_transpose[k][2]);
                flag = 1;
                break;
            }
        }
```



```c
            if(flag == 0)
            {
                printf("0 ");
            }
        }
        printf("\n");
    }
}
/* Function to directly read a matrix as a sparse matrix
// Read the sparse matrix
    int values;
    scanf("%d", &values);
    for(i = 0; i < values; i++)
    {
        printf("Row value: \n");
        scanf("%d", &sparse_matrix[i][0]);
        printf("Column value: \n");
        scanf("%d", &sparse_matrix[i][1]);
        printf("Value: \n");
        scanf("%d", &sparse_matrix[i][2]);
    }*/
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> gcc 7_SparseMatrix.c
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
Input the first matrix.
3 3
1 0 0
2 0 4
0 8 0
Printing the matrix in sparse matrix table format:
Row: 0 Column: 0 Value: 1
Row: 1 Column: 0 Value: 2
Row: 1 Column: 2 Value: 4
Row: 2 Column: 1 Value: 8
What value do you want to search in the matrix?
8
Found: Row: 2, Columns: 1
Printing the matrix in regular format:
1 0 0
2 0 4
0 8 0
Addition of matrices.
Input the second matrix.
3 3
0 1 0
3 3 0
0 6 0
Row: 0 Column: 0 Value: 1
Row: 0 Column: 1 Value: 1
Row: 1 Column: 0 Value: 5
Row: 1 Column: 1 Value: 3
Row: 1 Column: 2 Value: 4
Row: 2 Column: 1 Value: 14
1 1 0
5 3 4
0 14 0
Transpose of the first matrix:
1 2 0
0 0 8
0 4 0
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2>
```



```
PS C:\Users\meher\OneDrive\Documents\A Sem 3\DSA\Lab\Lab_2> ./a.exe
Input the first matrix.
2 4
1 0 0 8
0 2 3 0
Printing the matrix in sparse matrix table format:
Row: 0 Column: 0 Value: 1
Row: 0 Column: 3 Value: 8
Row: 1 Column: 1 Value: 2
Row: 1 Column: 2 Value: 3
What value do you want to search in the matrix?
6
Not found
Printing the matrix in regular format:
1 0 0 8
0 2 3 0
Addition of matrices.
Input the second matrix.
2 4
0 0 9 7
3 4 0 0
Row: 0 Column: 0 Value: 1
Row: 0 Column: 2 Value: 9
Row: 0 Column: 3 Value: 15
Row: 1 Column: 0 Value: 3
Row: 1 Column: 1 Value: 6
Row: 1 Column: 2 Value: 3
1 0 9 15
3 6 3 0
Transpose of the first matrix:
1 0 0 0
0 2 0 0
```

Concepts

The code takes in a matrix **as regular matrix input** (rows, columns, 2D-array). Then it counts the number of values in the matrix that are not 0, and then **converts the 2D-array into a sparse matrix** of size non-zero values * 3.

The code then prints out the sparse matrix in the sparse matric table format.

The code asks the user the number they want to **search in the sparse matrix**. Sparse matrix format makes it much easier to search for values as compared to 2D-arrays. It prints out the rows and columns of all the places a given value occurs at in the matrix. If it is not present it tells the user the value was not found.

The code then **prints out the matrix in the regular format**. Here, it uses advantage of fact that the sparse matrix is in order*****.

The code then asks the user for a second matrix as regular matrix input (rows, columns, 2D-array), which it then converts into a sparse matrix. It then performs the **addition operation** on the matrix elements. Here again, it takes advantage of the fact that the matrix elements are in order*****. It uses the logic which is somewhat similar to union. We traverse through both matrices and insert the element with the smaller row value (and if row value is same, the one eith the smaller column value), into the resultant sparse matrix. If they have the same row and column value, we add their values and insert this into the resultant matrix. The code then prints out the resultant matrix in sparse table format and regular matrix format.

The code next finds the **transpose** of the first matrix and prints in the regular matrix format.

At the end of the in comments in a function that can directly read a sparse matrix (rows, columns, value)

*****The sparse matrix has the elements stored in the order in which they appear on the matrix.