M M VASANTH

CH.SC.U4CSE24227

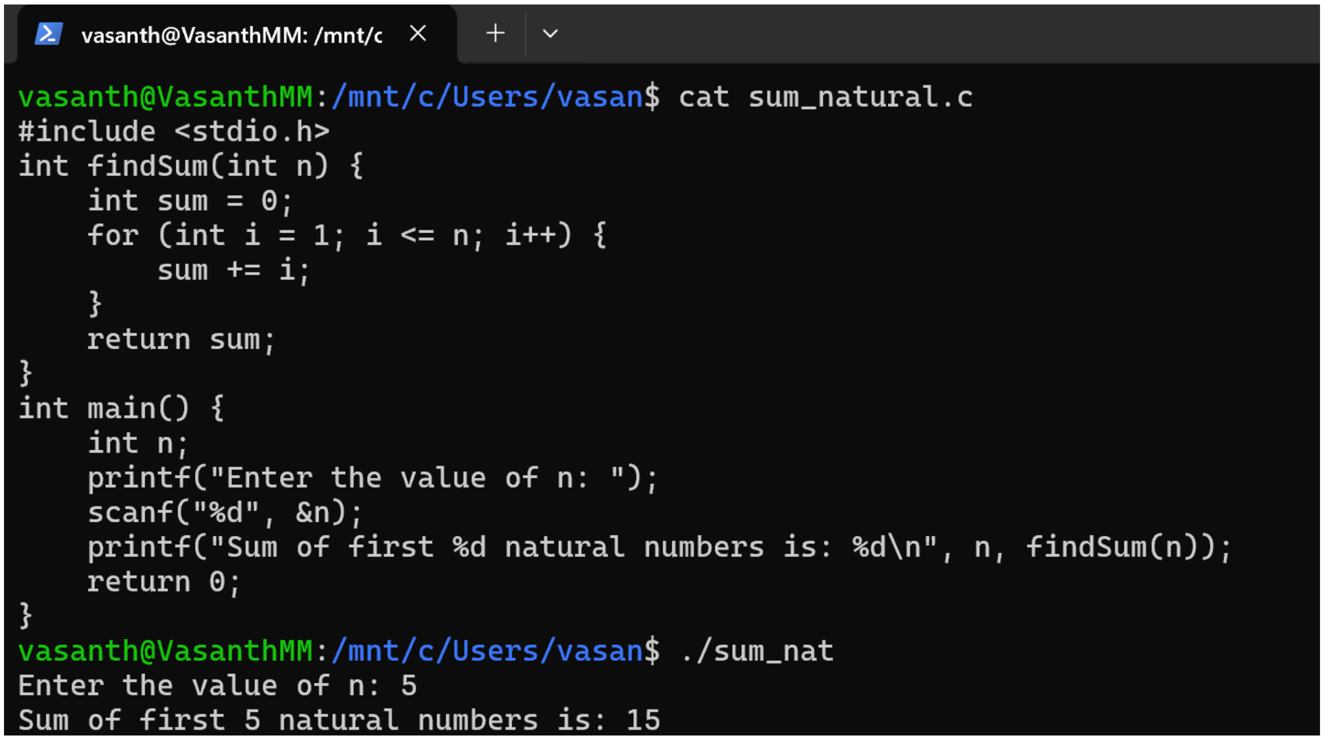
Week – 1

Date - 27/11/2025

Design and Analysis of Algorithm(23CSE211)

1. Write a program to find sum of first n natural numbers using user defined functions

Code and Output:



```
vasanth@VasanthMM:/mnt/c/Users/vasan$ cat sum_natural.c
#include <stdio.h>
int findSum(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
int main() {
    int n;
    printf("Enter the value of n: ");
    scanf("%d", &n);
    printf("Sum of first %d natural numbers is: %d\n", n, findSum(n));
    return 0;
}
vasanth@VasanthMM:/mnt/c/Users/vasan$ ./sum_nat
Enter the value of n: 5
Sum of first 5 natural numbers is: 15
```

Time Complexity: O(n)

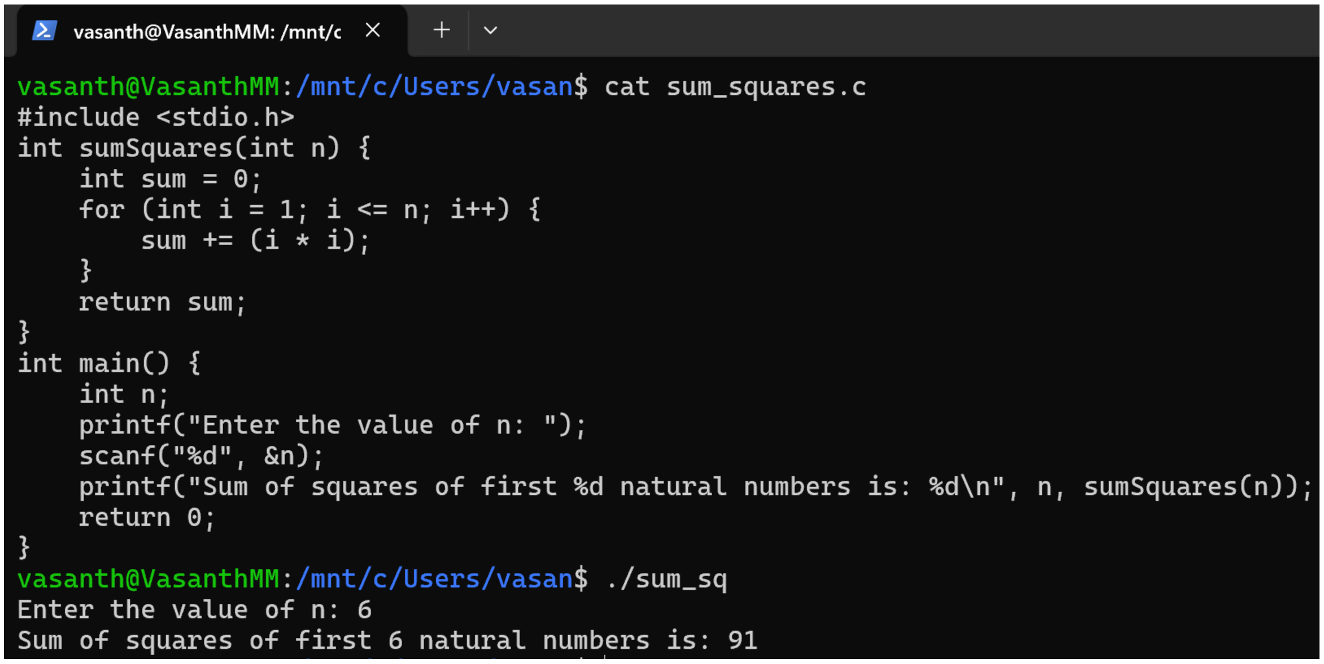
Since I used a single for loop that iterates from 1 to n, the operations inside the loop (addition) are executed exactly n times. Therefore, the time taken increases linearly with the input size.

Space Complexity: O(1)

The program uses a fixed number of variables (i, sum, n) regardless of how large the input n is. No arrays or extra data structures were used.

2. Write a program to find sum of squares of first n natural numbers

Code and Output:



```
vasanth@VasanthMM:/mnt/c/Users/vasan$ cat sum_squares.c
#include <stdio.h>
int sumSquares(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += (i * i);
    }
    return sum;
}
int main() {
    int n;
    printf("Enter the value of n: ");
    scanf("%d", &n);
    printf("Sum of squares of first %d natural numbers is: %d\n", n, sumSquares(n));
    return 0;
}
vasanth@VasanthMM:/mnt/c/Users/vasan$ ./sum_sq
Enter the value of n: 6
Sum of squares of first 6 natural numbers is: 91
```

- **Time Complexity: O(n)**

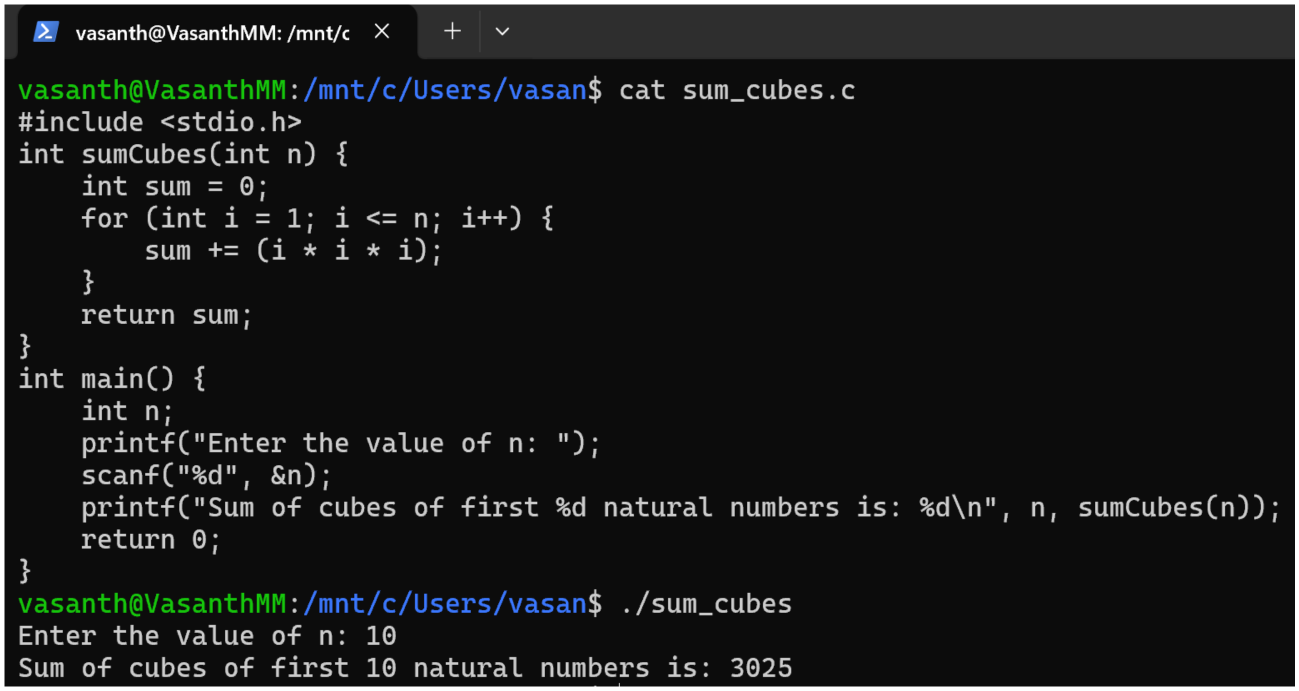
Similar to the first problem, the loop runs n times. Calculating the square (i * i) is a constant time operation inside the loop. Thus, the total time remains linear relative to n.

- **Space Complexity: O(1)**

I only used a few integer variables to store the running sum and the loop counter, so the memory usage is constant.

3. Write a program to find sum of cubes of first n natural numbers

Code and Output:



```
vasanth@VasanthMM:/mnt/c/Users/vasan$ cat sum_cubes.c
#include <stdio.h>
int sumCubes(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += (i * i * i);
    }
    return sum;
}
int main() {
    int n;
    printf("Enter the value of n: ");
    scanf("%d", &n);
    printf("Sum of cubes of first %d natural numbers is: %d\n", n, sumCubes(n));
    return 0;
}
vasanth@VasanthMM:/mnt/c/Users/vasan$ ./sum_cubes
Enter the value of n: 10
Sum of cubes of first 10 natural numbers is: 3025
```

- **Time Complexity: O(n)**

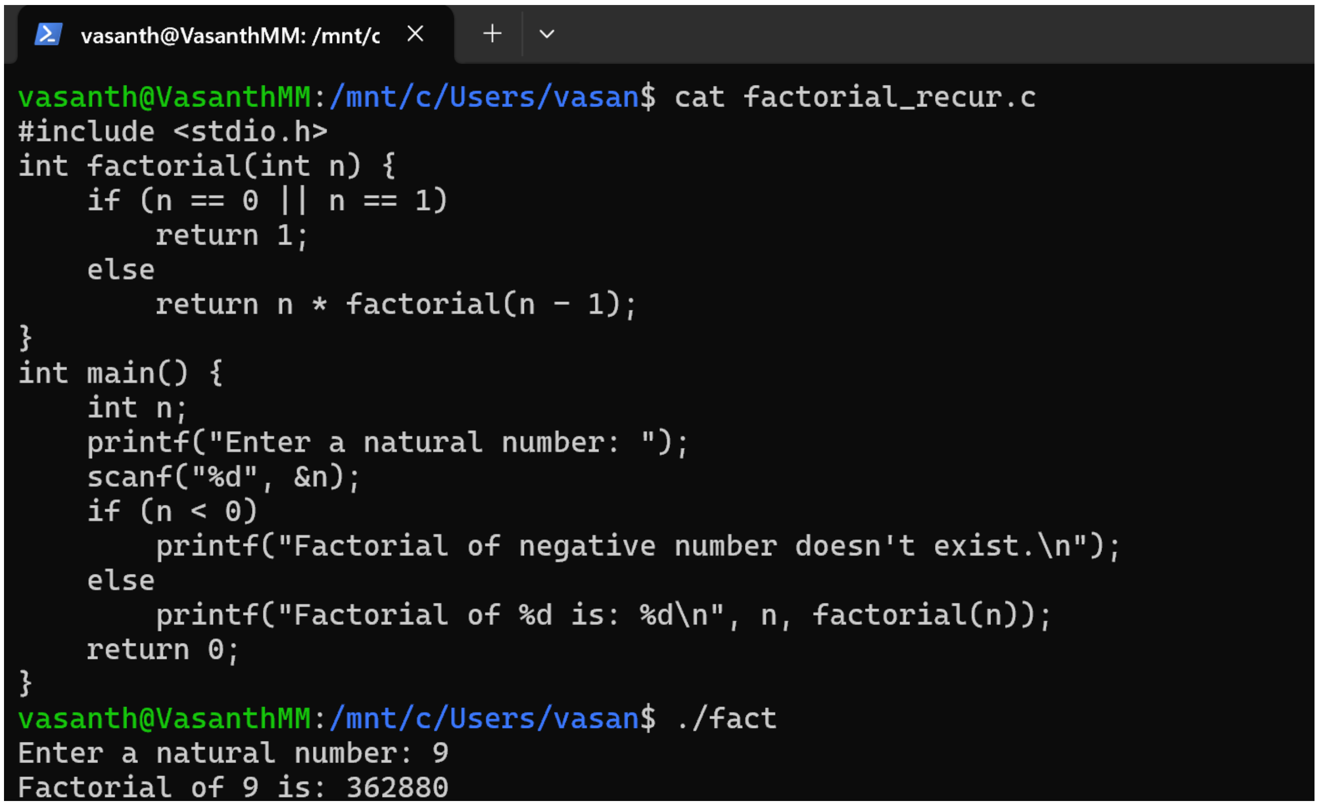
The logic relies on a single pass loop from 1 to n. Even though the arithmetic operation (i * i * i) is slightly heavier than simple addition, it is still considered a constant time operation O(1). Repeating this n times results in linear time complexity.

- **Space Complexity: O(1)**

No dynamic memory or arrays were allocated; only basic variables were used to hold the accumulated value.

4. Write a program to find factorial of the given integer using recursion

Code and Output:



```
vasanth@VasanthMM:/mnt/c/Users/vasan$ cat factorial_recur.c
#include <stdio.h>
int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
int main() {
    int n;
    printf("Enter a natural number: ");
    scanf("%d", &n);
    if (n < 0)
        printf("Factorial of negative number doesn't exist.\n");
    else
        printf("Factorial of %d is: %d\n", n, factorial(n));
    return 0;
}
vasanth@VasanthMM:/mnt/c/Users/vasan$ ./fact
Enter a natural number: 9
Factorial of 9 is: 362880
```

- **Time Complexity: O(n)**

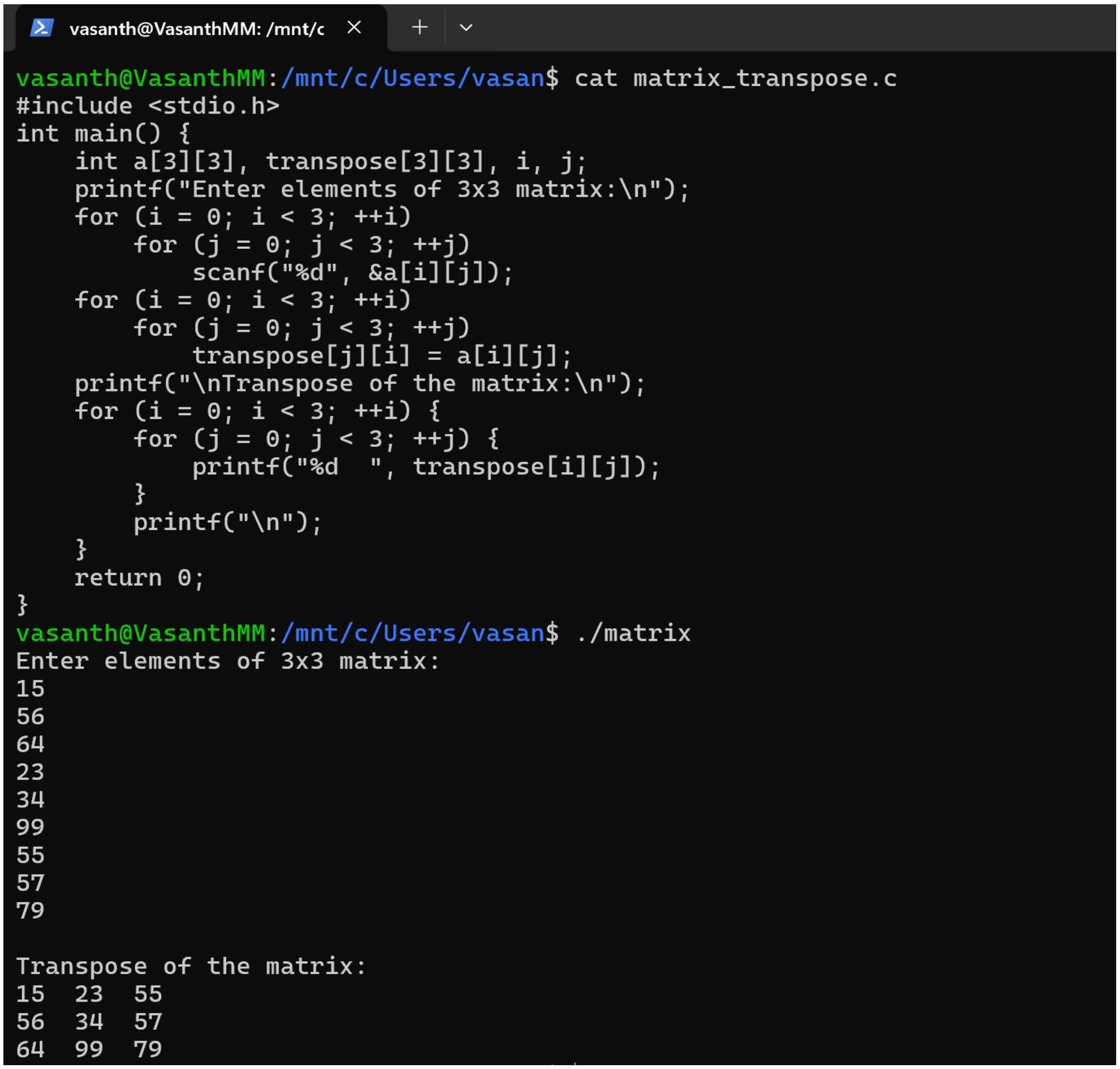
The function calls itself recursively n times (decrementing from n down to 1). Each function call performs a constant amount of work (multiplication), so the total time is proportional to n.

- **Space Complexity: O(n)**

Unlike loops, recursion uses the Call Stack. Since there are n active function calls waiting in memory before the base case is reached, the stack depth grows linearly with n, consuming O(n) auxiliary space.

5. Write a program to transpose a 3x3 matrix

Code and Output:



vasanth@VasanthMM: /mnt/c

```
vasanth@VasanthMM:/mnt/c/Users/vasan$ cat matrix_transpose.c
#include <stdio.h>
int main() {
    int a[3][3], transpose[3][3], i, j;
    printf("Enter elements of 3x3 matrix:\n");
    for (i = 0; i < 3; ++i)
        for (j = 0; j < 3; ++j)
            scanf("%d", &a[i][j]);
    for (i = 0; i < 3; ++i)
        for (j = 0; j < 3; ++j)
            transpose[j][i] = a[i][j];
    printf("\nTranspose of the matrix:\n");
    for (i = 0; i < 3; ++i) {
        for (j = 0; j < 3; ++j) {
            printf("%d  ", transpose[i][j]);
        }
        printf("\n");
    }
    return 0;
}
vasanth@VasanthMM:/mnt/c/Users/vasan$ ./matrix
Enter elements of 3x3 matrix:
15
56
64
23
34
99
55
57
79

Transpose of the matrix:
15  23  55
56  34  57
64  99  79
```

- **Time Complexity: O(n^2)**

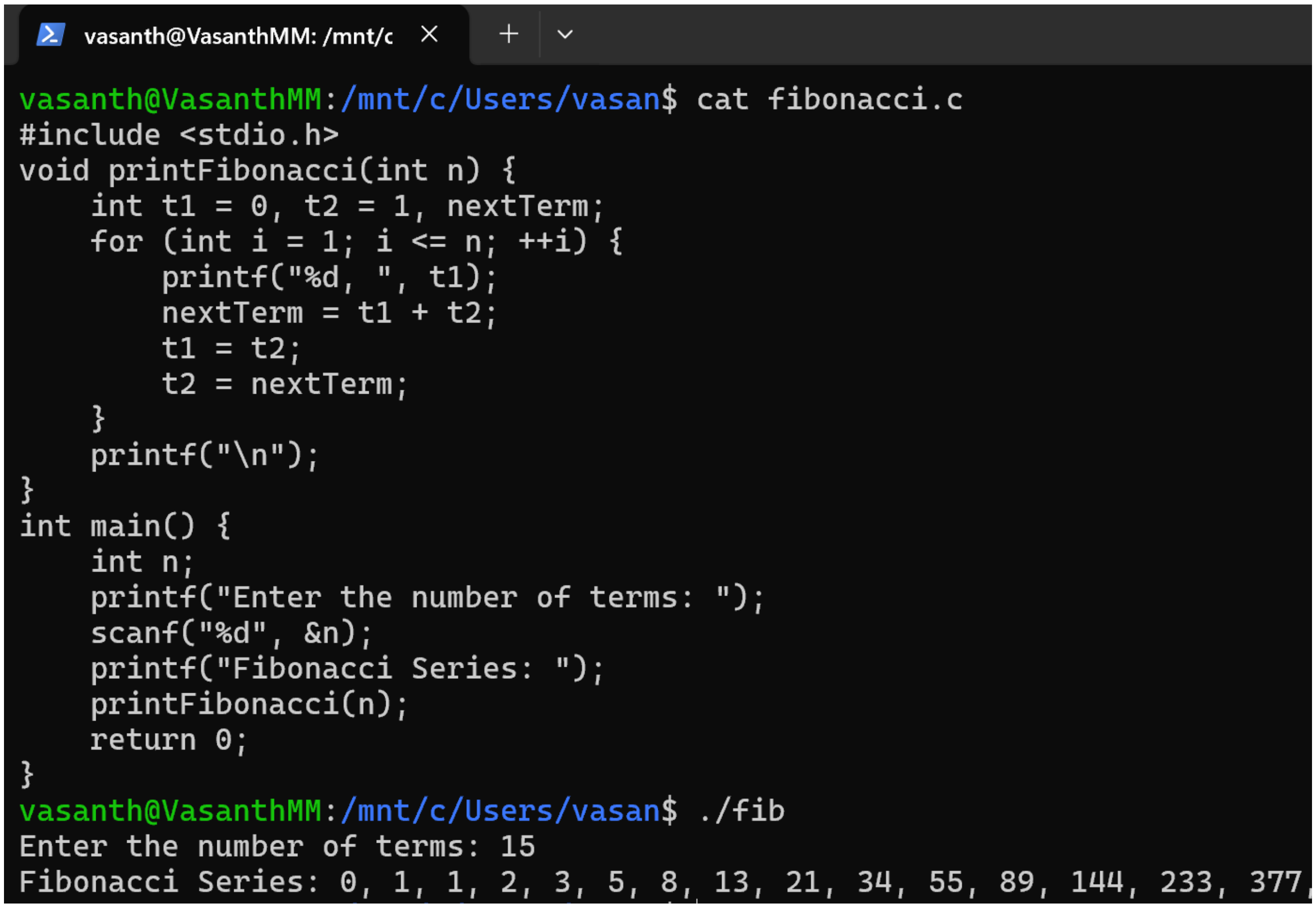
I used a nested for loop structure. The outer loop runs for the rows (n) and the inner loop runs for the columns (n). This results in n * n total iterations to visit every element in the matrix.

- **Space Complexity: O(n^2)**

To store the transposed output, I declared a second 2D array of the same size as the original. This requires auxiliary space proportional to the total number of elements in the matrix (n * n).

6. Write a program to find Fibonacci series

Code and Output:



```
vasanth@VasanthMM:/mnt/c/Users/vasan$ cat fibonacci.c
#include <stdio.h>
void printFibonacci(int n) {
    int t1 = 0, t2 = 1, nextTerm;
    for (int i = 1; i <= n; ++i) {
        printf("%d, ", t1);
        nextTerm = t1 + t2;
        t1 = t2;
        t2 = nextTerm;
    }
    printf("\n");
}
int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    printf("Fibonacci Series: ");
    printFibonacci(n);
    return 0;
}
vasanth@VasanthMM:/mnt/c/Users/vasan$ ./fib
Enter the number of terms: 15
Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
```

- **Time Complexity: O(n)**

The program requires printing n terms. To generate these, the loop must execute n times. Each iteration involves simple addition and variable swapping, which takes constant time.

- **Space Complexity: O(1)**

I optimized the space by strictly using three variables (t1, t2, nextTerm) to track the series. Since I did not use an array to store the history of all generated numbers, the memory usage remains constant regardless of the series length.