

# DESIGN AND ANALYSIS OF ALGORITHM

## ASSIGNMENT 1

Saikrishna N

CH.SC.U4CSE24244

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

```
#include <stdio.h>
int sum(int n) {
    int s = 0;
    for(int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}

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

    printf("Sum = %d\n", sum(n));
    return 0;
}
```

Time Complexity : O(n)

```
raven@raven:~/daa_assignment_1$ gcc sum_of_n.c -o sum_of_n
raven@raven:~/daa_assignment_1$ ./sum_of_n
Enter n: 10
Sum = 55
```

### **My Interpretation:**

The problem asks to add all numbers from 1 up to n.

A loop is the easiest way because it lets you go through each number one by one.

A variable is used to store the running total while the loop keeps adding new values.

A user-defined function helps keep the logic separate and neat.

The loop starts at 1, ends at n, and keeps increasing by 1 each time.

Each value gets added to the total inside the function.

When the loop finishes, the total is returned back to the main program.

## **2. Write a program to find the sum of squares of first n natural numbers using user-defined function.**

```
#include <stdio.h>

int sumSquares(int n) {
    int s = 0;
    for(int i = 1; i <= n; i++) {
        s = s + (i * i);
    }
    return s;
}

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

    printf("Sum of squares = %d\n", sumSquares(n));
    return 0;
}
```

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

```
raven@raven:~/daa_assignment_1$ gcc sum_of_squares_of_n.c -o sum_of_squares_of_n
raven@raven:~/daa_assignment_1$ ./sum_of_squares_of_n
Enter n: 7
Sum of squares = 140
```

### **My Interpretation:**

This is similar to the first problem, but instead of adding the number, we add its square.

A loop from 1 to n allows easy calculation of  $i * i$  for each number.

A variable stores the total sum of all squares.  
A user-defined function organizes the calculation nicely.  
Each loop step calculates the square and adds it to the sum.  
The function finishes when the loop reaches n.  
Finally, the function returns the total square-sum.

### 3. Write a program to find the sum of cubes of first n natural numbers using user-defined function.

```
#include <stdio.h>

int sumCubes(int n) {
    int s = 0;
    for(int i = 1; i <= n; i++) {
        s = s + (i * i * i);
    }
    return s;
}

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

    printf("Sum of cubes = %d\n", sumCubes(n));
    return 0;
}
```

**Time Complexity: O(n)**

```
raven@raven:~/daa_assignment_1$ gcc sum_of_cubes_of_n.c -o sum_of_cubes_of_n
raven@raven:~/daa_assignment_1$ ./sum_of_cubes_of_n
Enter n: 5
Sum of cubes = 225
```

#### My Interpretation:

The structure matches the previous two problems.  
A loop is used to go from 1 to n in order.  
Inside the loop, the cube of each number is calculated using  $i * i * i$ .  
A running sum variable keeps track of all cubes added together.

A user-defined function makes the logic clean.  
The loop gradually builds the answer by adding each cube.  
The resulting total is returned after all cubes are processed.

#### 4. Write a program to find factorial of a natural number using recursive function.

```
#include <stdio.h>

int fact(int n) {
    if(n == 0 || n == 1)
        return 1;
    else
        return n * fact(n - 1);
}

int main() {
    int n;
    printf("Enter a number: ");
    scanf("%d", &n);

    printf("Factorial = %d\n", fact(n));
    return 0;
}
```

**Time Complexity:** O(n)

```
raven@raven:~/daa_assignment_1$ gcc factorial_of_n.c -o factorial_of_n
raven@raven:~/daa_assignment_1$ ./factorial_of_n
Enter a number: 9
Factorial = 362880
```

##### My Interpretation:

Factorial is based on the mathematical rule:  $n! = n * (n-1)!$ .  
Recursion fits this rule naturally because a function can call itself.  
A base case is important to stop the recursion when n becomes 0 or 1.  
Each recursive call reduces the value of n by 1.  
The function keeps multiplying numbers as it returns back through the calls.  
Recursive thinking helps understand how the result is built step by step.  
The final result comes after all recursive calls complete.

## 5. Write a program to transpose a 3x3 matrix.

```
#include <stdio.h>

int main() {
    int a[3][3], t[3][3];

    printf("Enter 3x3 matrix:\n");
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++) {
            scanf("%d", &a[i][j]);
        }
    }

    // Transpose
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++) {
            t[j][i] = a[i][j];
        }
    }

    printf("Transposed Matrix:\n");
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++) {
            printf("%d ", t[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

**Time Complexity:** O(1)

```
raven@raven:~/daa_assignment_1$ gcc transpose_of_3x3.c -o transpose_of_3x3
raven@raven:~/daa_assignment_1$ ./transpose_of_3x3
Enter 3x3 matrix:
5 9 2 11 6 7 3 1 10
Transposed Matrix:
5 11 3
9 6 1
2 7 10
```

### **My Interpretation:**

A  $3 \times 3$  matrix has rows and columns that need to be swapped for the transpose.

Nested loops are the simplest way to access every element in the matrix.

The element at position  $(i, j)$  becomes  $(j, i)$  in the transpose.

Storing the result in a second matrix keeps the process simple.

This swapping of row and column positions is repeated for all 9 elements.

A small fixed-size matrix makes the logic straightforward.

Finally, the transposed matrix is printed row by row.

**6. Write a program to print the Fibonacci series up to a given number using user-defined function (for first n natural numbers).**

```
#include <stdio.h>

void fibonacci(int n) {
    int a = 0, b = 1, c;

    printf("%d %d ", a, b);

    for(int i = 3; i <= n; i++) {
        c = a + b;
        printf("%d ", c);
        a = b;
        b = c;
    }
    printf("\n");
}

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

    fibonacci(n);

    return 0;
}
```

## Time Complexity: O(n)

```
raven@raven:~/daa_assignment_1$ gcc fibonacci_upto_n_terms.c -o fibonacci_upto_n_terms
raven@raven:~/daa_assignment_1$ ./fibonacci_upto_n_terms
Enter n: 8
0 1 1 2 3 5 8 13
```

### My Interpretation:

The first two Fibonacci numbers are always 0 and 1.

Each new number is found by adding the two previous numbers together.

Two variables keep track of the last two values in the sequence.

A loop starts generating new terms one by one.

Each iteration updates the variables to move forward in the sequence.

The newly generated number is printed immediately.

This continues until all n terms are produced.