DSA SQUAD BY SOURCIFY IN

DAY 3:

Recursion

What is Recursion in programming?

- Recursion is a technique where a function calls itself to solve a smaller subproblem until it reaches a base case that can be solved directly.
- The code above defines a recursive function to calculate the factorial of a positive integer n, which is the product of all positive integers less than or equal to n.
- The function has two cases: a base case and a recursive case. The base case is when n is 0 or 1, in which case the function returns 1. The recursive case is when n is greater than 1, in which case the function returns n times the factorial of n-1.
- The function works by reducing the problem size by one in each recursive call, until it reaches the base case. For example, factorial(5) = 5 * factorial(4) = 5 * 4 * factorial(3) = 5 * 4 * 3 * factorial(2) = 5 * 4 * 3 * 2 * factorial(1) = 5 * 4 * 3 * 2 * 1 = 120.

A recursive function typically has two parts:

1. **Base Case:** The condition under which the function stops calling itself.
2. **Recursive Case:** The part where the function calls itself to work towards the base case.

Syntax

Recursive Function Structure

```
return_type function_name(parameters) {
    if (base_case_condition) {
        ( Base case)
        return base_case_value;
    } else {
        (Recursive case)
        return function_name(modified_parameters);
    }
}
```

How Recursion Works

1. **Function Call Stack:** Each recursive call pushes a new frame onto the call stack.
2. **Base Case:** When the base case is met, the function returns without making further calls.
3. **Unwinding the Stack:** After reaching the base case, the call stack unwinds, returning values back through the recursive calls.

Types of Recursion

Direct Recursion

A function calls itself directly.

```
void directRecursion() {
    (Some condition)
    directRecursion();
}
```

Indirect Recursion

A function calls another function, which in turn calls the first function.

```
void functionA();
void functionB();

void functionA() {
    (Some condition)
    functionB();
}

void functionB() {
    (Some condition)
    functionA();
}
```

Example:

Fibonacci Sequence

The Fibonacci sequence is a classic example of recursion, where each term is the sum of the two preceding ones.

```
#include <stdio.h>

int fibonacci(int n) {
    if (n <= 1) {
        return n; (Base case)
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2); (Recursive case)
    }
}
```

```
int main() {
    int num = 10;
    for (int i = 0; i < num; i++) {
        printf("%d ", fibonacci(i));
    }
    return 0;
}
```

Sum of Digits

Calculating the sum of the digits of a number.

```
#include <stdio.h>

int sumOfDigits(int n) {
    if (n == 0) {
        return 0; (Base case)
    } else {
        return n % 10 + sumOfDigits(n / 10); (Recursive case)
    }
}

int main() {
    int num = 12345;
    printf("Sum of digits of %d is %d\n", num, sumOfDigits(num));
    return 0;
}
```

Advantages of Recursion

1. **Simplifies Code:** Recursion can simplify the code for problems that have a natural recursive structure.
2. **Modularity:** Recursion breaks a problem into smaller subproblems, making it easier to understand and manage.
3. **Reduces Complexity:** Recursion can reduce the complexity of algorithms by eliminating the need for iterative control structures.

Disadvantages of Recursion

1. **Performance:** Recursive calls can be expensive in terms of memory and processing time due to the overhead of maintaining the call stack.
2. **Stack Overflow:** Deep recursion can lead to stack overflow if the base case is not reached or the problem size is too large.
3. **Difficult to Debug:** Recursive code can be more difficult to debug and trace compared to iterative solutions.

Optimizing Recursion

Tail Recursion

Tail recursion occurs when the recursive call is the last operation in the function. Some compilers optimize tail-recursive functions to reduce stack overhead.

```
#include <stdio.h>


int tailFactorial(int n, int result) {
    if (n == 0) {
        return result; (Base case)
    } else {
        return tailFactorial(n - 1, n * result); (Tail recursion)
    }
}
```

```
int main() {
    int num = 5;
    printf("Factorial of %d is %d\n", num, tailFactorial(num, 1));
    return 0;
}
```

Memoization

Storing the results of expensive function calls and reusing them when the same inputs occur again to avoid redundant calculations.

```
#include <stdio.h>


int memo[1000];


int fibonacciMemo(int n) {
    if (memo[n] != -1) {
        return memo[n];
    }
    if (n <= 1) {
        return n; // Base case
    }
    memo[n] = fibonacciMemo(n - 1) + fibonacciMemo(n - 2); //
Recursive case
    return memo[n];
}


int main() {
    for (int i = 0; i < 1000; i++) {
        memo[i] = -1;
    }
    int num = 10;
    for (int i = 0; i < num; i++) {
        printf("%d ", fibonacciMemo(i));
    }
    return 0;
}
```

*Advanced Topic-

Tower of Hanoi

Introduction

The Tower of Hanoi is a mathematical puzzle that involves moving a stack of disks from one rod to another, following specific rules. It is a popular example used to illustrate the power and elegance of recursion.

Problem Statement

You are given three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks neatly stacked in ascending order of size on one rod, the smallest at the top, making a conical shape.

Rules

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk may be placed on top of a smaller disk.

Objective

Move the entire stack to another rod, following the rules.

Recursive Solution

The recursive solution to the Tower of Hanoi involves breaking the problem down into smaller subproblems. The key is to recognize that moving n disks can be reduced to moving n-1 disks.

```
#include <stdio.h>


( Function to move n disks from source to destination using auxiliary)

void towerOfHanoi(int n, char source, char destination, char auxiliary) {

    if (n == 1) {

        printf("Move disk 1 from rod %c to rod %c\n", source, destination);

        return;

    }

    (Move n-1 disks from source to auxiliary, so they are out of the way')

    towerOfHanoi(n - 1, source, auxiliary, destination);


    (Move the nth disk from source to destination)

    printf("Move disk %d from rod %c to rod %c\n", n, source, destination);


    (Move the n-1 disks from auxiliary to destination)

    towerOfHanoi(n - 1, auxiliary, destination, source);

}


int main() {

    int n = 3; (Number of disks)
    towerOfHanoi(n, 'A', 'C', 'B'); (A, B, and C are names of rods)
    return 0;
}
```

Conclusion

Recursion is a powerful tool for solving complex problems by breaking them down into simpler subproblems. Understanding when and how to use recursion, along with its advantages and limitations, is essential for effective programming in C.