Introduction to Computing and Programming

Recursion

Recap

FUNCTION WITH ARRAYS

MACRO & INLINE FUNCTIONS

SOME PRACTICE QUESTIONS ON THESE TOPICS

Content

- Command line Argument in C
- What is Recursion?
- How Recursion works?
- Types of Recursion
- Advantages & Disadvantages of Recursion

Command line argument in C

```
int main() {
    ...
}
```

- main() is mostly defined with a return type of int and without parameters.
- Command-line arguments are the values given after the name of the program in the command-line.
- To pass command-line arguments, we define main() with two arguments: the first argument is the **number of command-line arguments** and the second is a **list of command-line arguments**.

```
int main(int argc, char *argv[]) { /* ... */ }
        or
int main(int argc, char **argv) { /* ... */ }
```

- Here,
  - argc (ARGument Count) is an integer variable that stores the number of command-line arguments passed by the user including the name of the program.
  - argv (ARGument Vector) is an array of character pointers listing all the arguments.
  - If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings.

Command line argument in C: Example

```
int main(int argc, char* argv[])
{printf("Program name is: %s", argv[0]);
if (argc == 1)
printf("\nNo Extra Command Line Argument Passed");
if (argc >= 2) {
printf("\nNumber Of Arguments Passed: %d", argc);
printf("\n----Following Are CLI Arguments Passed----");
for (int i = 0; i < argc; i++)
printf("\nargv[%d]: %s", i, argv[i]);
}return 0;}
```

```
$ ./a.out
```

```
Program Name Is: ./a.out
No Extra Command Line Argument
Passed Other Than Program Name
```

```
$ ./a.out First Second Third
```

```
Program Name Is: ./a.out
Number Of Arguments Passed: 4
----Following Are CLI Arguments
Passed----
argv[0]: ./a.out
argv[1]: First
argv[2]: Second
argv[3]: Third
```

Recursion

What is Recursion?

- A process by which a function **calls itself repeatedly** is called **recursion**.
- **Either directly**
  - X calls X
- **OR cyclically in a chain**
  - X calls Y, and Y calls X

- Recursion **breaks the problem into smaller subproblems** and applies the same function to solve the smaller subproblems.

Recursion Example

Mowing the lawn can be broken down into a recursive process.

- Mow the lawn
  - Mow the frontyard
    - Mow the left front
    - Mow the right front
  - Mow the backyard
    - Mow the left back
    - Mow the right back

Write a C program to calculate the factorial of a number using function

Example: Let say number is 5

5! = 5*4*3*2*1 =120

Write a C program to calculate the factorial of a number

```
#include <stdio.h>
int calculateFactorial(int n) {
    int result = 1;  // Initialize result to 1

    // Iteratively multiply result by i (from 2 to n)
    for (int i = 2; i <= n; i++) {
        result *= i;
    }

    return result;  // Return the factorial result
}
int main() {
    int number;
    // Ask the user for input
    printf("Enter a number to calculate its factorial: ");
    scanf("%d", &number);

    // Check if the input is non-negative
    if (number < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        // Call the function and print the result
        int factorial = calculateFactorial(number);
        printf("Factorial of %d is: %d\n", number, factorial);
    }

    return 0;
}
```

What is Recursion?

- Used for repetitive computations in which each action is stated in terms of a previous result
  - **factorial(n) = n * factorial (n-1)**
- **Basic Syntax Structure of Recursive Functions:**

```
Return_type function_name (args) {
    // function statements
    // base condition
    // recursion case (recursive call)
}
```

How Recursion Works?

1. Divide the problem into smaller subproblems.
2. Solve the smallest version of the problem (Base Case).
3. Combine the solutions of the smaller problems to solve the original problem.

- **Components of Recursion:**
  - Base Case: The condition that **stops recursion**.
  - Recursive Case: The condition where the function **continues calling itself.**

Recursion – How to write

- For a problem to be written in recursive form, **two conditions** are to be satisfied:
- It should be possible to express the problem in **recursive form**
  - Solution of the problem in terms of solution of the same problem on smaller sized data
- The problem statement must include a stopping condition **(base case)** and recurring condition **(recursive case)**

✧ **Example: factorial(n) defines n!**

factorial(n) = 1, if n = 0

= n * factorial(n-1), if n > 0

Stopping condition

Recursive step

Write a C program to calculate the factorial of a number using Recursion

```
#include <stdio.h>
// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0 || n == 1)  // Base case: factorial of 0 or 1 is 1
        return 1;
    else
        return n * factorial(n - 1);  // Recursive case: n * factorial of (n - 1)
}

int main() {
    int number;

    // Ask the user for input
    printf("Enter a number to calculate its factorial: ");
    scanf("%d", &number);

    // Check if the input is non-negative
    if (number < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        // Call the recursive function and print the result
        int result = factorial(number);
        printf("Factorial of %d is: %d\n", number, result);
    }

    return 0;
}
```

Recursive Function Call Stack

- Recursive calls are **pushed onto the call stack.**

- When the base case is reached, the stack **unwinds and combines** results.

Factorial Problem

✧ Let us look into an illustrative example

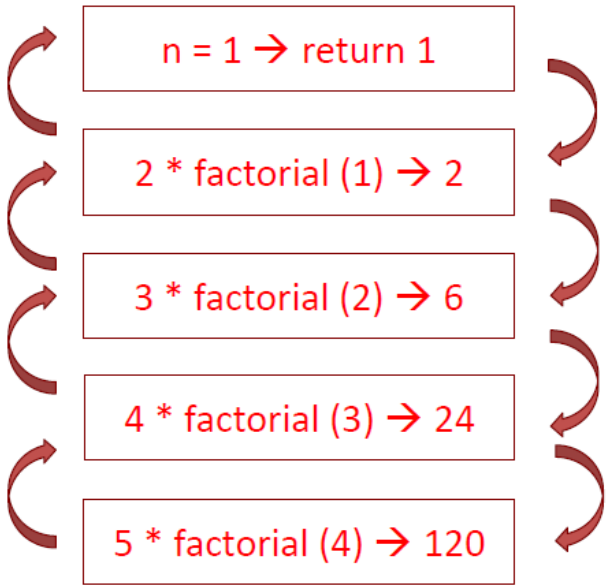
```
long int factorial (int n) {
      if (n == 1)      // Base Case
          return 1;
      else             // Recursive Case
          return n * factorial(n-1);
}
```

Consider n = 5,

Factorial (5) = 120

If n = 7 then

factorial (7) = 5040



Recursion – A few examples

- **Factorial Problem:**

  factorial(0) = 1

  factorial(n) = n * factorial(n-1), if n > 0

- **Greatest Common Divisor (GCD):**

  gcd(m, m) = m;

  gcd(m, n) = gcd(n, m%n), if m > n

  gcd(m, n) = gcd(n%m, m), if m < n

- **Fibonacci Series (1, 1, 2, 3, 5, 8, 13, 21, . . .):**

  fibo(0) = 1

  fibo(1) = 1

  fibo(n) = fibo(n-1) + fibo(n-2), if n > 1

A recursive function example

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

2.  void CountDown(int countInt) {
3.    if (countInt <= 0) {
4.      printf("Go!\n");
5.    }
6.    else {
7.      printf("%d\n", countInt);
8.      CountDown(countInt - 1);
9.    }
10. }

11. int main(void) {
12.   CountDown(2);
13.   return 0;
14. }
```

Output

```
2
1
Go!
```

Write a C program to calculate the sum of n number using Recursion

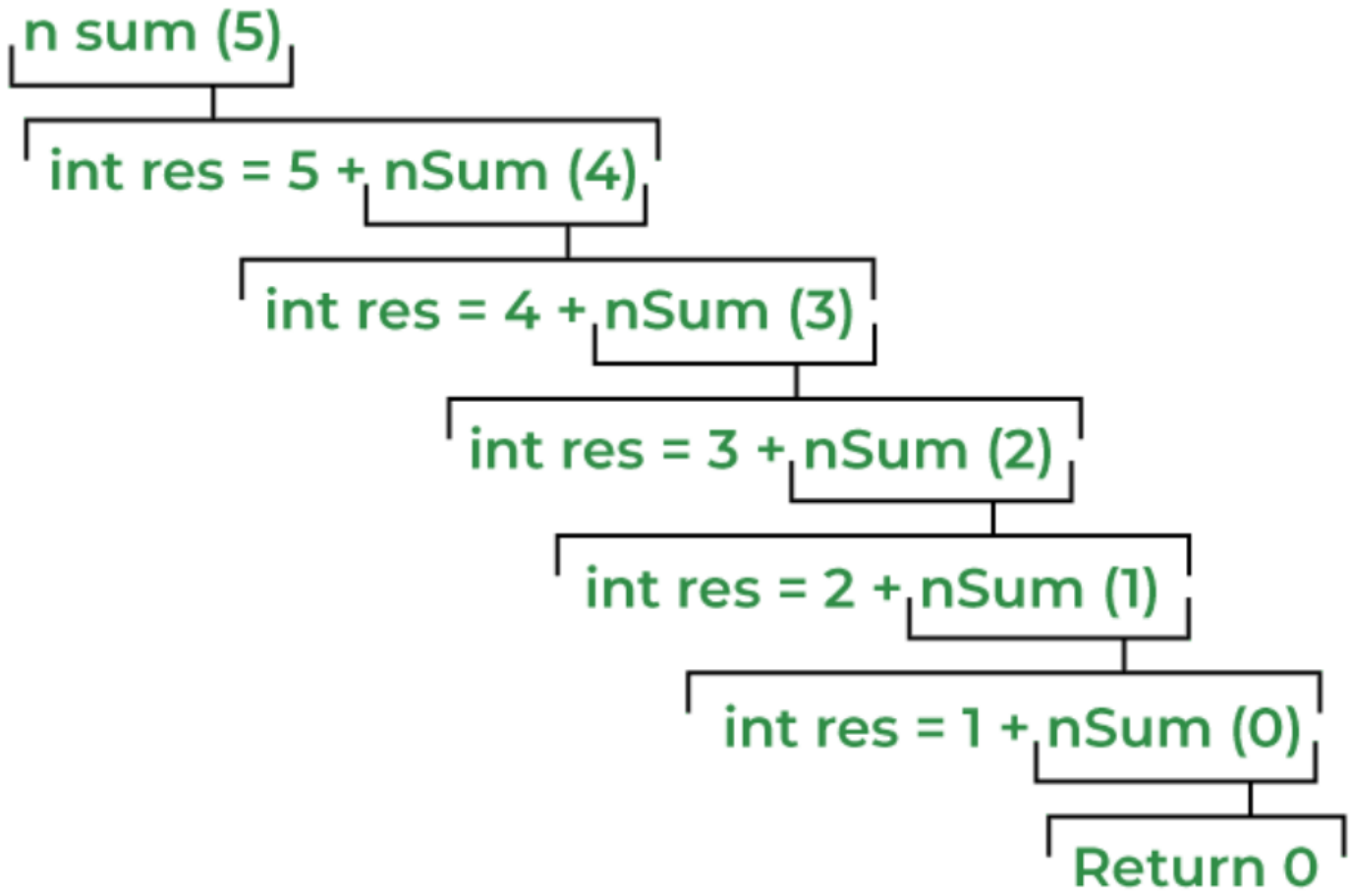
Eg: enter the number 5

5+4+3+2+1=15

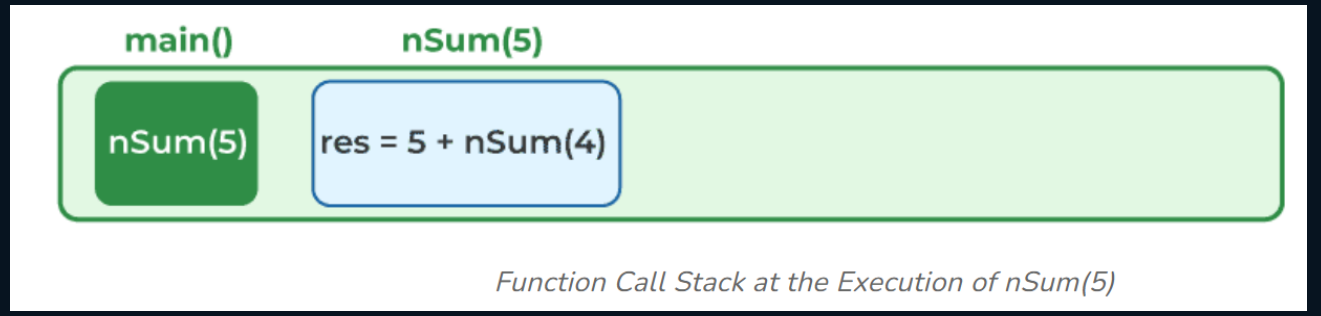
Write a C program to calculate the sum of n number using Recursion

```
// C Program to calculate the sum of first N natural
numbers
// using recursion
#include <stdio.h>
int nSum(int n)
{
   // base condition to terminate the recursion when N
= 0
   if (n == 0) {
      return 0;
   }
   // recursive case / recursive call
   int res = n + nSum(n - 1);
   return res;}
int main()
{
   int n = 5;

   // calling the function
   int sum = nSum(n);
   printf("Sum of First %d Natural
Numbers: %d", n, sum);
   return 0;
}
```

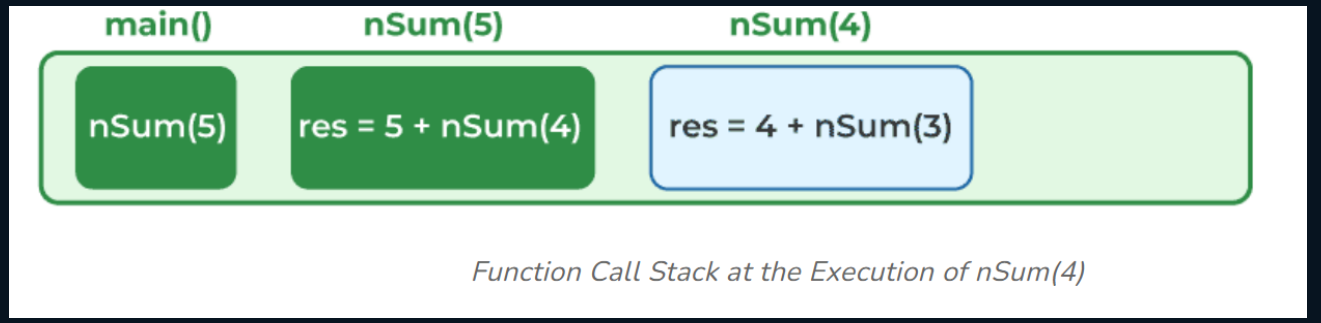
Recursion Tree Diagram of nSum(5) Function



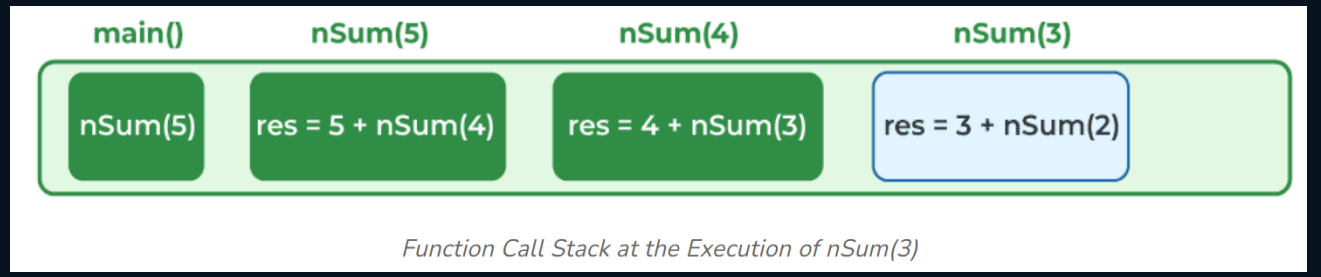
Function call Stack of nSum(5)



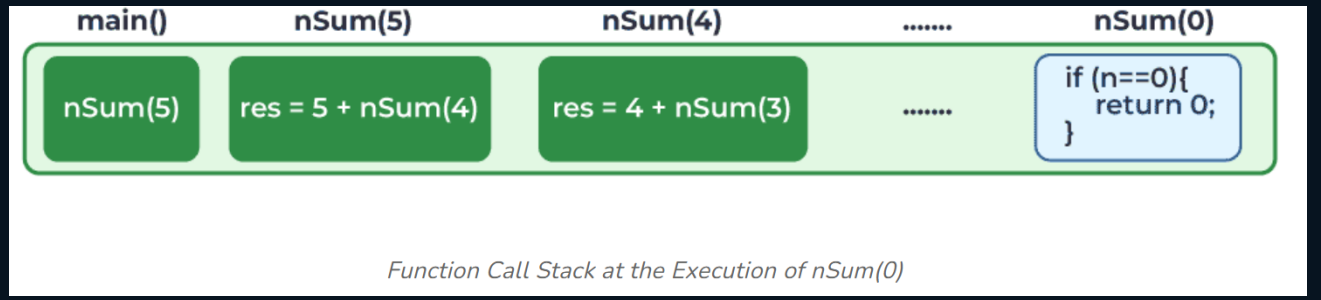
Function Call Stack at the Execution of nSum(5)



Function Call Stack at the Execution of nSum(4)

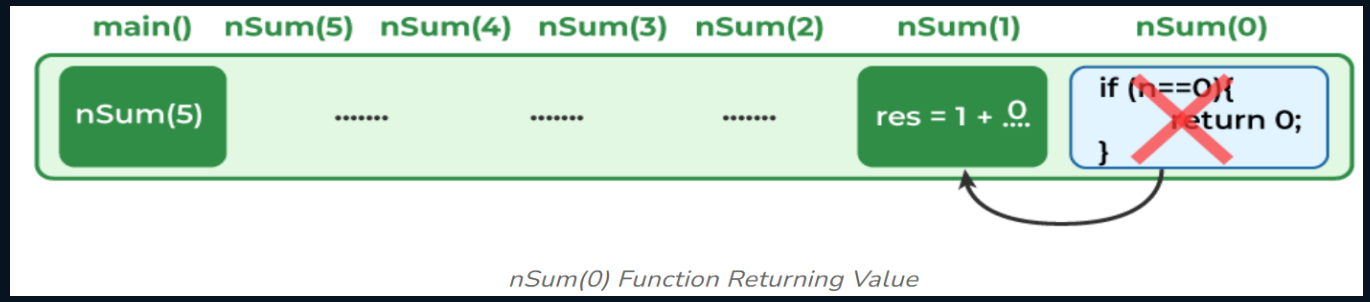


Function Call Stack at the Execution of nSum(3)

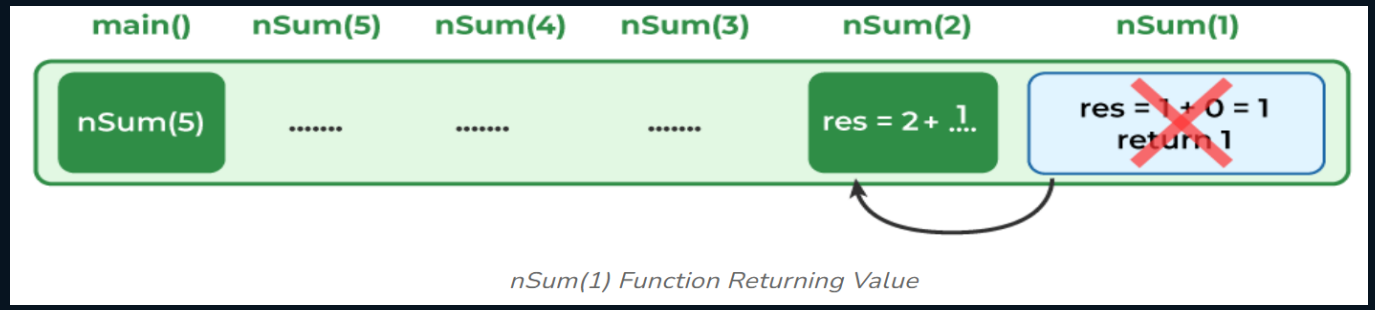


Function Call Stack at the Execution of nSum(0)

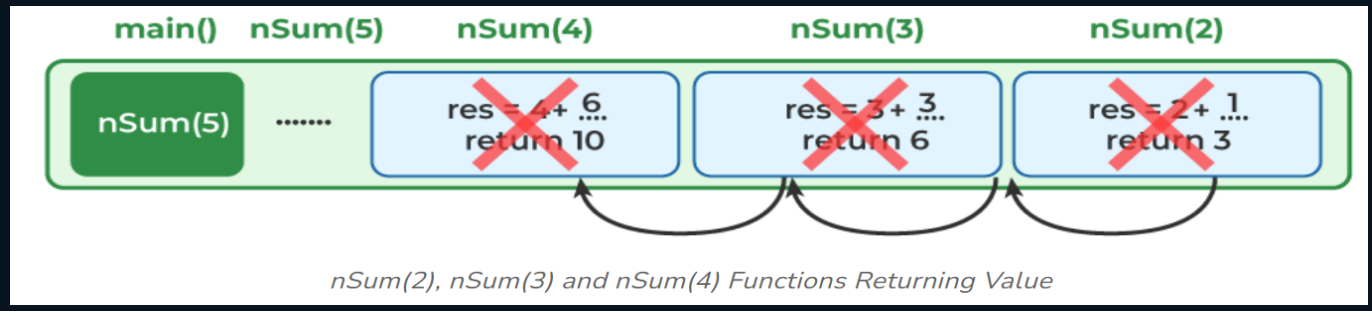
Function Returning Value



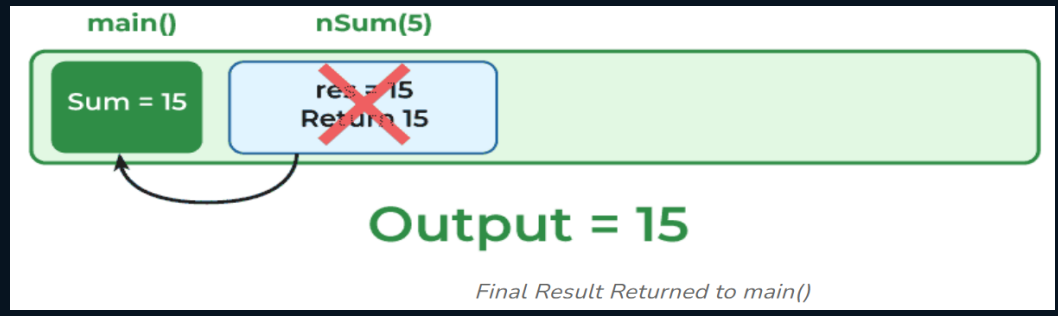
nSum(0) Function Returning Value



nSum(1) Function Returning Value



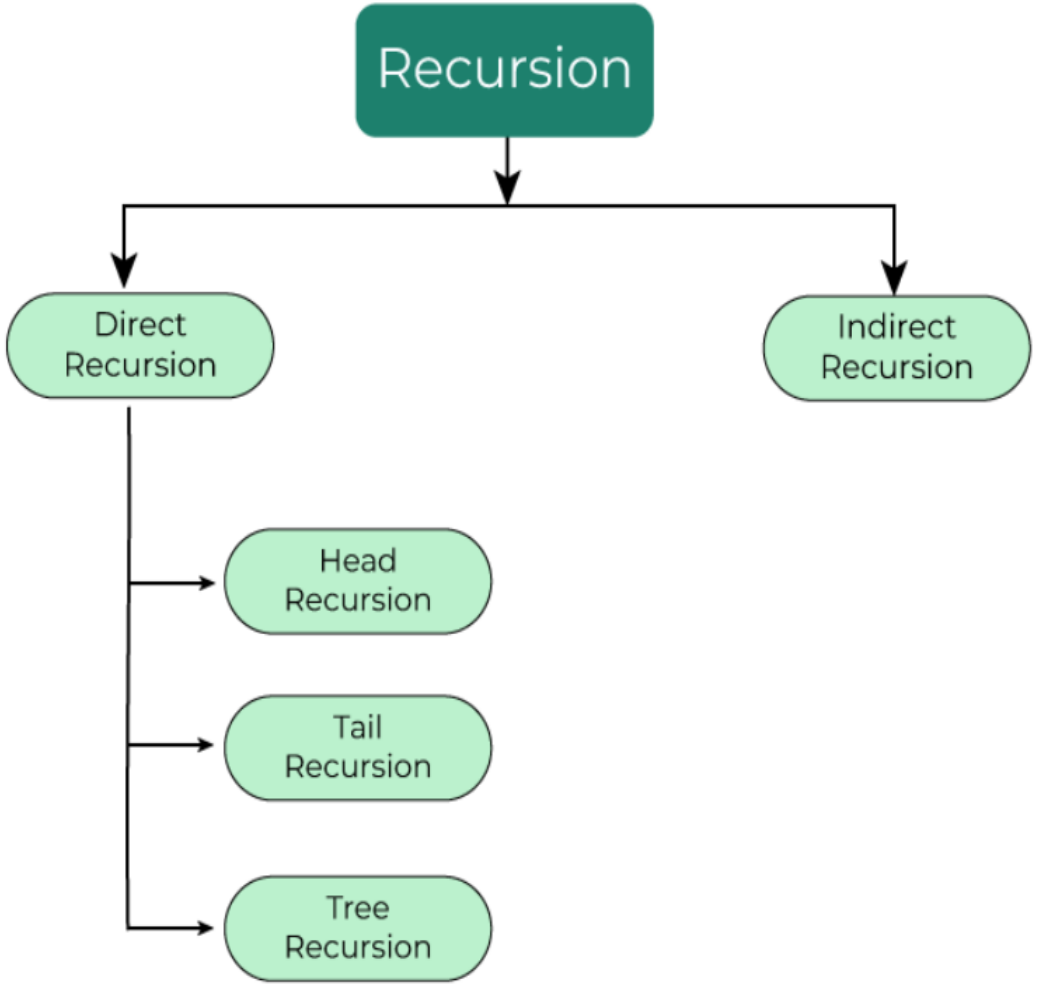
nSum(2), nSum(3) and nSum(4) Functions Returning Value



Final Result Returned to main()

Types of Recursion

- **Direct Recursion:** A function directly calls itself.
- **Head Recursion:** The position of its only recursive call is at the start of the function.
- **Tail Recursion:** the position of the recursive call is at the end of the function.
- **Tree Recursion:** multiple recursive calls present in the body of the function.
- **Indirect Recursion:** A function calls another function, which in turn calls the original function.



C Program to Find the Factorial of a Natural Number using Tail Recursion

```
// C program to find the factorail using
tail recursion
#include <stdio.h>

int factorialTail(int n)
{
    // Base case
    if (n == 1 || n == 0) {
        return 1;
    }
    else {
        // Tail recursive call
        return n * factorialTail(n - 1);
    }
}

int main()
{

    int n = 5;

    int fact1 = factorialTail(n);
    printf("Resursive Factorial of %d: %d \n",
n, fact1);

    return 0;
}
```

Example of Tree Recursion

Fibonacci – Recurrence:

Problem: Find n^{th} Fibonacci number using Recursion

Example: 6^{th} Fibonacci number

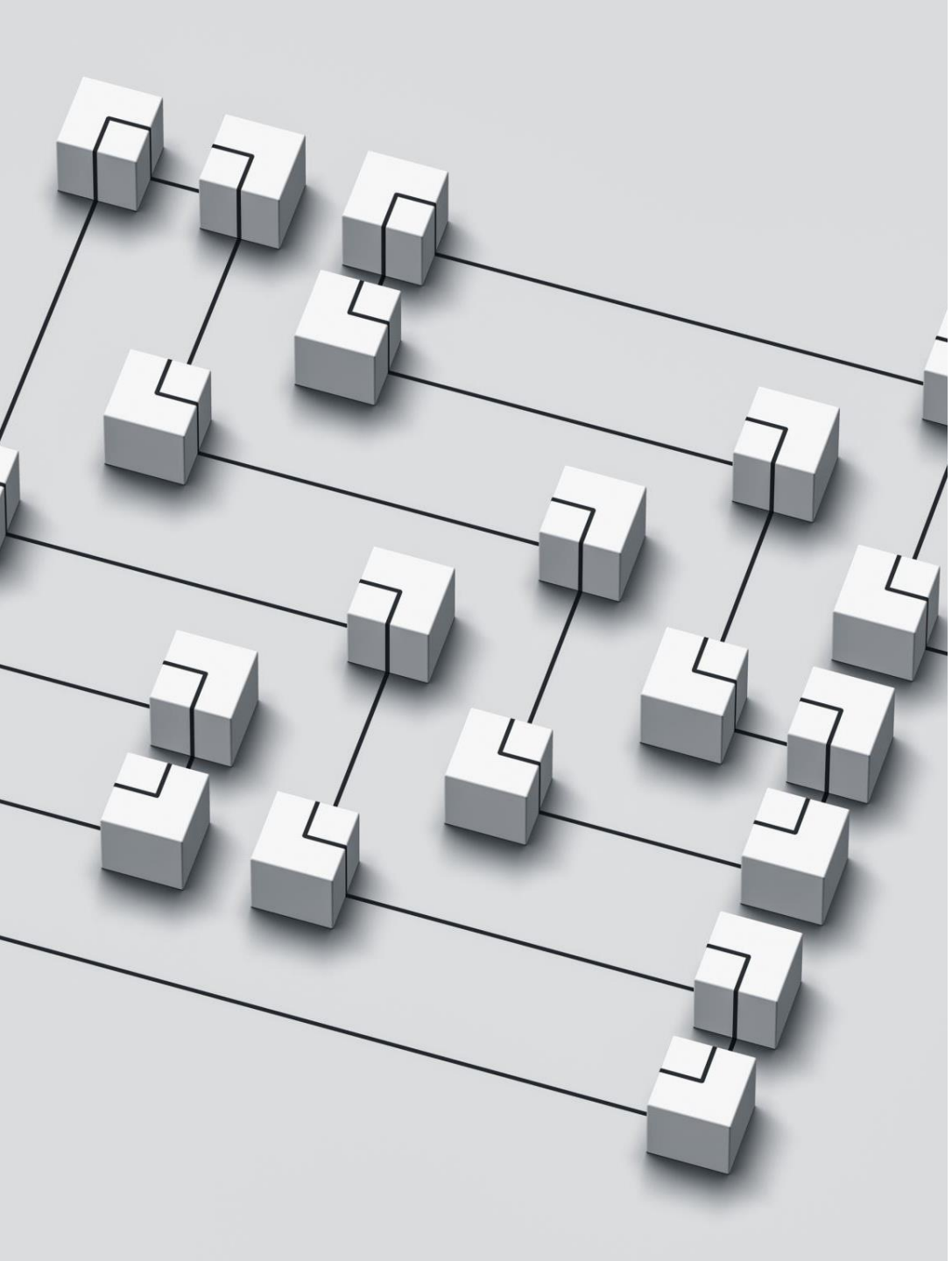
1 1 2 3 5 **8** 13 21 34 55
89 144 233 377 610 987 1597 . . .

Hint: fibo(n) = fibo(n-1) + fibo(n-2)

```
int fibo(int n) {
    if ( n == 0  || n == 1 ) return 1;
    return fibo(n-2) + fibo(n-1);
}
```

C Program to find the Fibonacci Number using Tree Recursion

```
// C Program to find the fibonacci number using tree
// recursion
#include <stdio.h>

int fibonacci(int n)
{
   // Base case
   // Fibonacci of 0 and 1 is the number itself
   if (n <= 1) {
      return n;
   }
   else {
      // Tree recursive calls
      return fibonacci(n - 1) + fibonacci(n - 2);
   }}

int main()
{

   // function call
   int n = fibonacci(3);

   // print 5th fibonacci number
   printf("%d", n);

   return 0;
}
```



Upcoming Slides

- Recursion Vs iteration
- Advantages & Disadvantages of Recursion
- Recursion in Arrays
- Exercises on Recursion