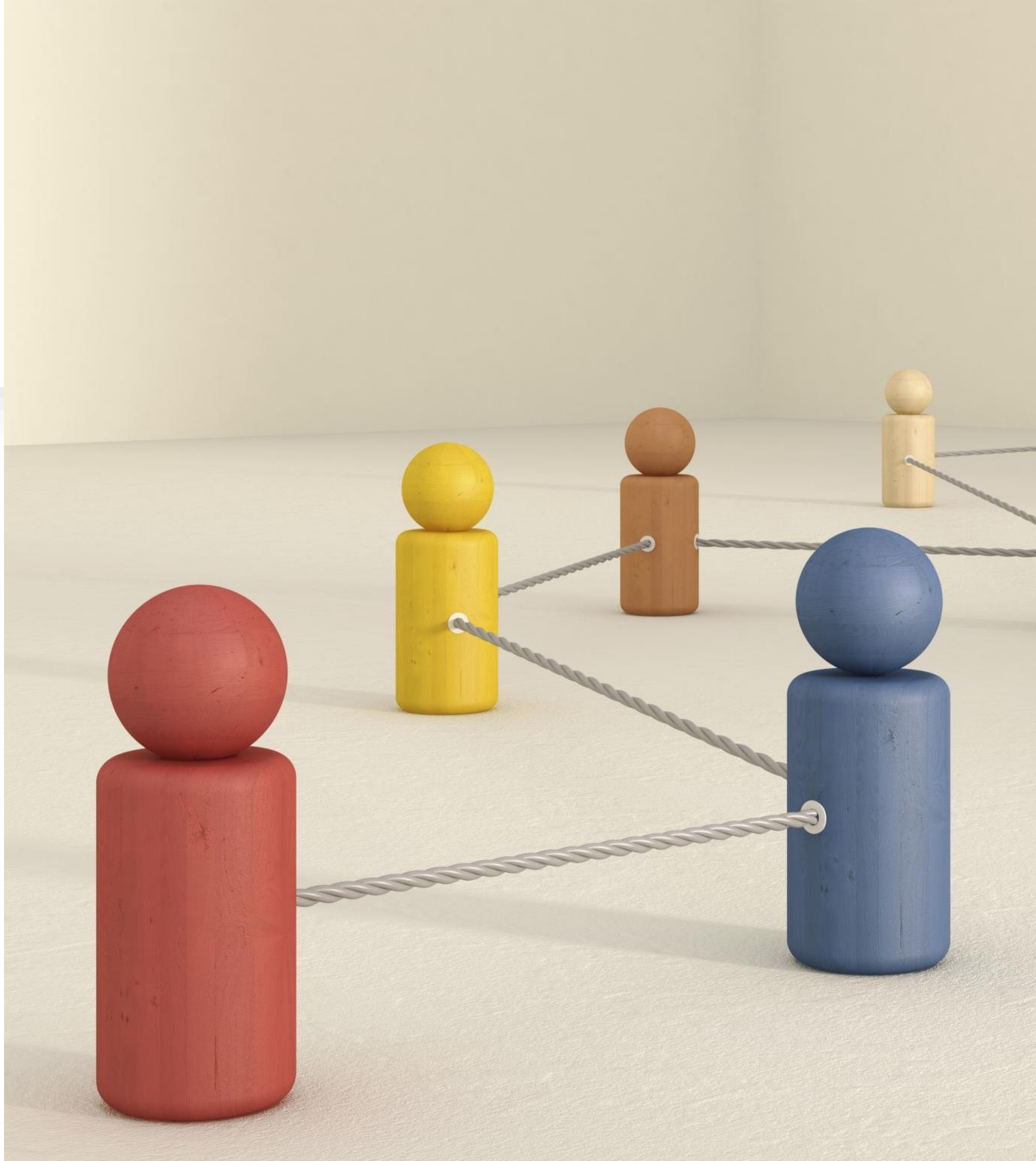
# Introduction to Computing and Programming

## Recursion in Arrays

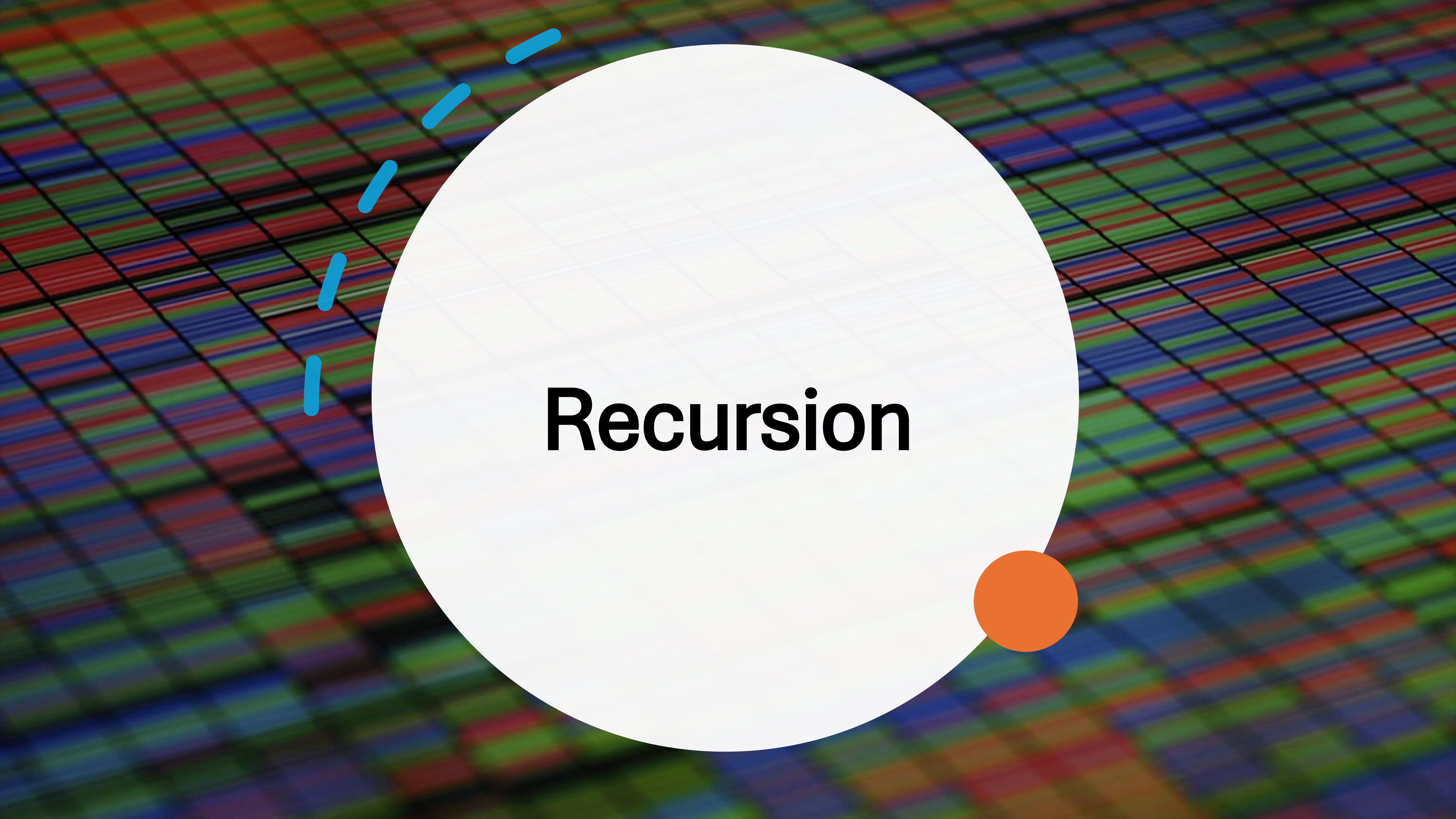
# Content

- Recap
- Types of Recursion with Example
- Recursion Vs iteration
- Advantages & Disadvantages of Recursion
- Recursion in Arrays
- Exercises



# Recap

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



# 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

# How Recursion Works?

1. Divide the problem into smaller subproblems.

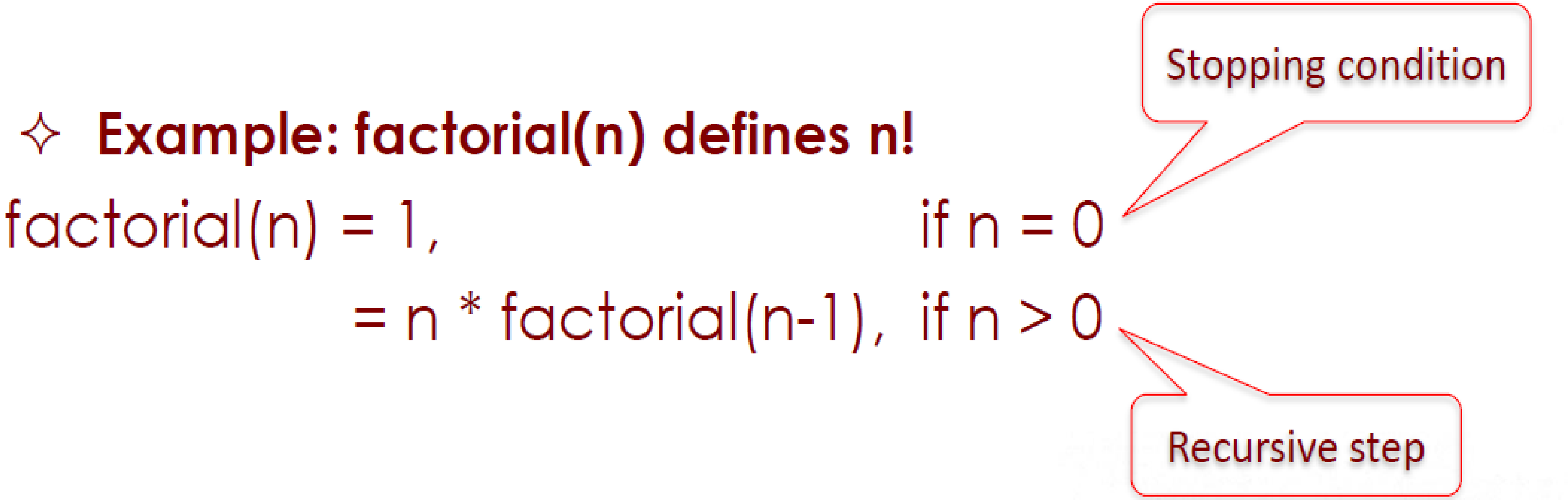
2. Solve the smallest version of the problem (Base Case).

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.**

# Few more examples of Recursion

- Factorial: **factorial(n) = n * factorial (n-1)**



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

factorial(n) = 1, if n = 0

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

Stopping condition

Recursive step

- Fibonacci
  - ***Base case:*** *F(n) = n, when n = 0 or n = 1*
  - ***Recursive case:*** *F(n) = F(n-1) + F(n-2) for n>1*

# Recursion more efficient or not?

✧ Let us consider **Fibonacci** series

**Basic case:**

n = 0 ⇒ return 1

n = 1 ⇒ return 1

**Recursive case:**

n > 1 ⇒ return fib(n -1) + fib(n -2)

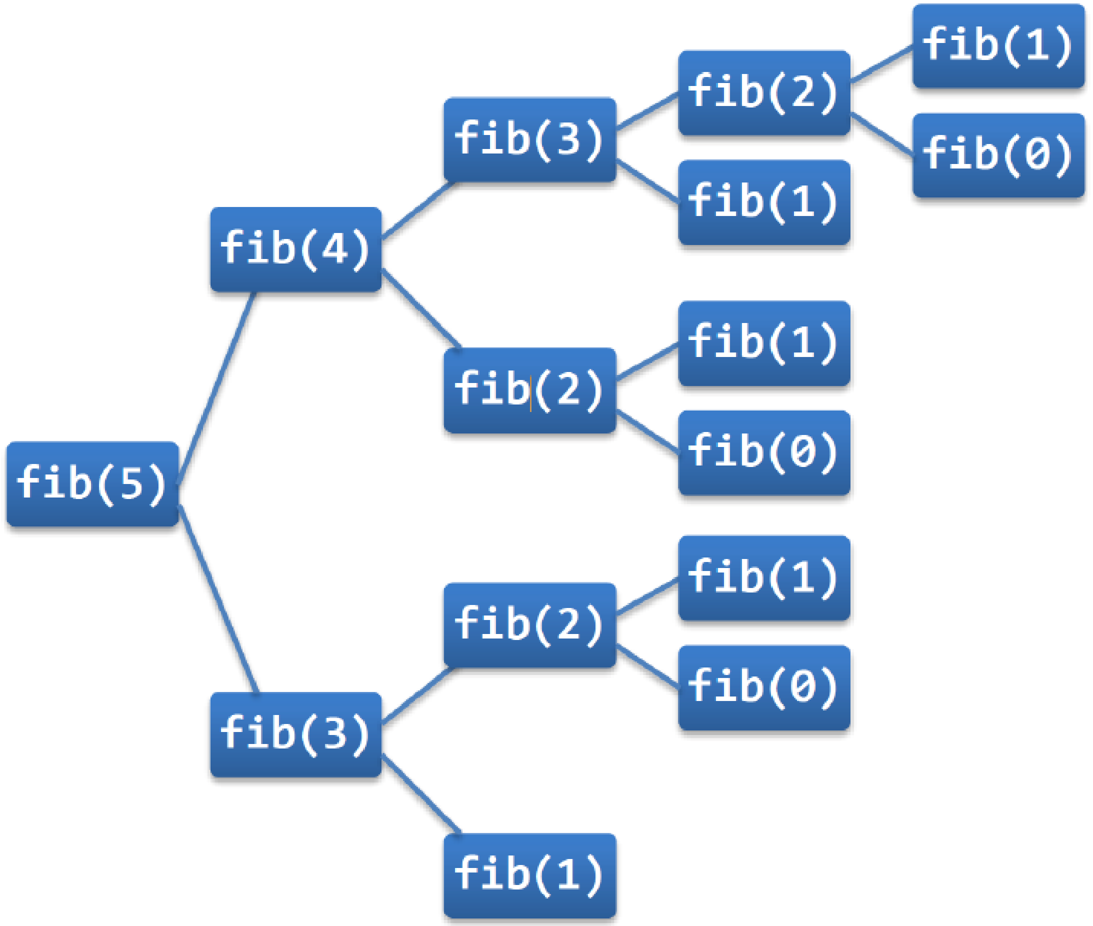
**Function:**

```
int fib(int n) {
        if(n <= 1) return1;
        return fib(n -2) + fib(n -1);
}
```

The function always terminates since the parameters of the recursive call (n-2 and n-1) are closer to 0 and 1.

# Fibonacci numbers

- fib(5) is illustrated below:



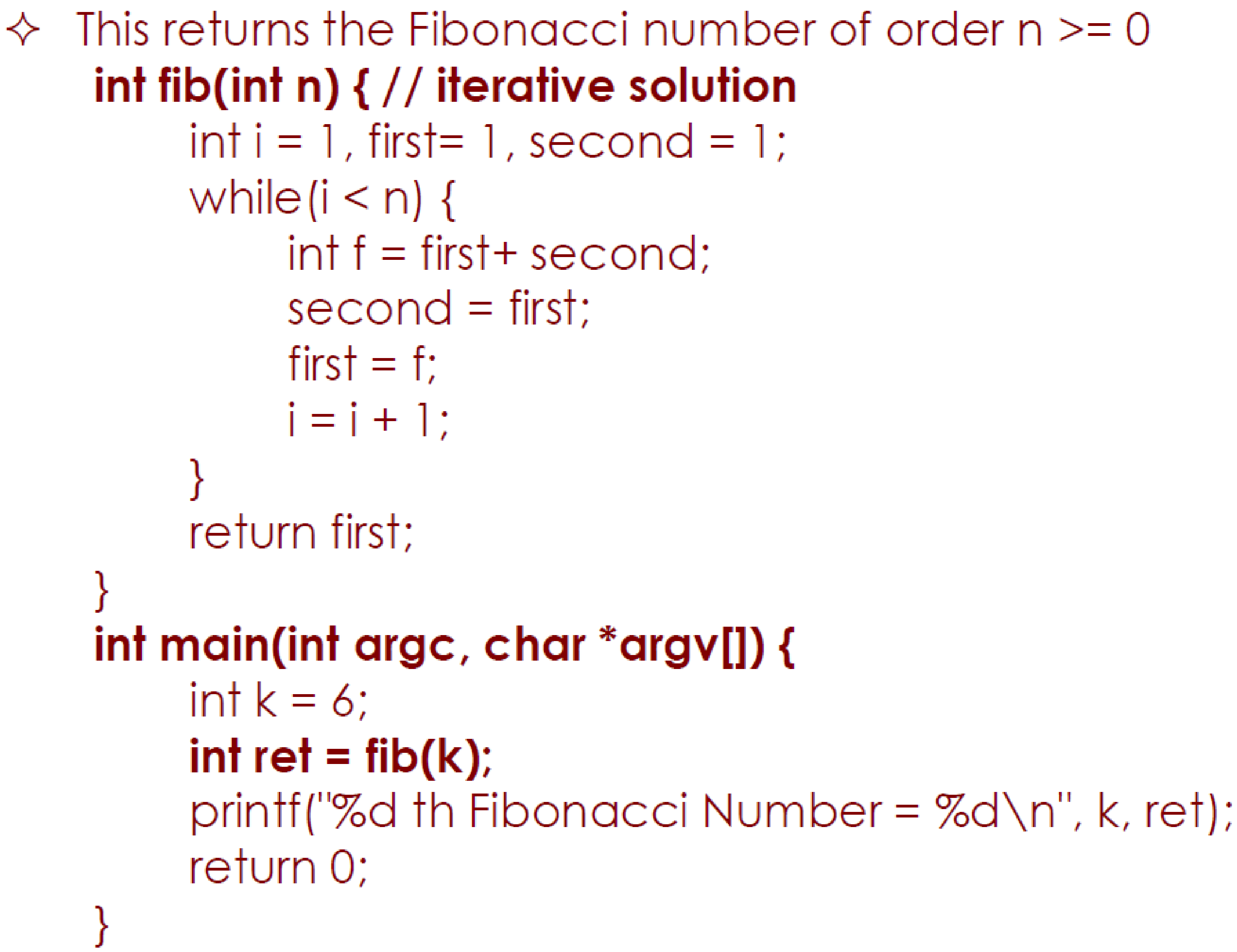
# Fibonacci numbers-Recursive method Analysis

- When fib(5) is calculated:
  - fib(5) is called once
  - fib(4) is called once
  - fib(3) is called twice
  - fib(2) is called 3 times
  - fib(1) is called 5 times
  - fib(0) is called 3 times

- When fib(n) is calculated, how many times will fib(1) and fib(0) be called?

- Example: fib(50) calls fib(1) and fib(0) about $2.4 \cdot 10^{10}$ times

# Fibonacci numbers- Iterative Method



✧ This returns the Fibonacci number of order n >= 0

```
int fib(int n) { // iterative solution
    int i = 1, first= 1, second = 1;
    while(i < n) {
        int f = first+ second;
        second = first;
        first = f;
        i = i + 1;
    }
    return first;
}
int main(int argc, char *argv[]) {
    int k = 6;
    int ret = fib(k);
    printf("%d th Fibonacci Number = %d\n", k, ret);
    return 0;
}
```

# Fibonacci numbers- Iterative Method Analysis

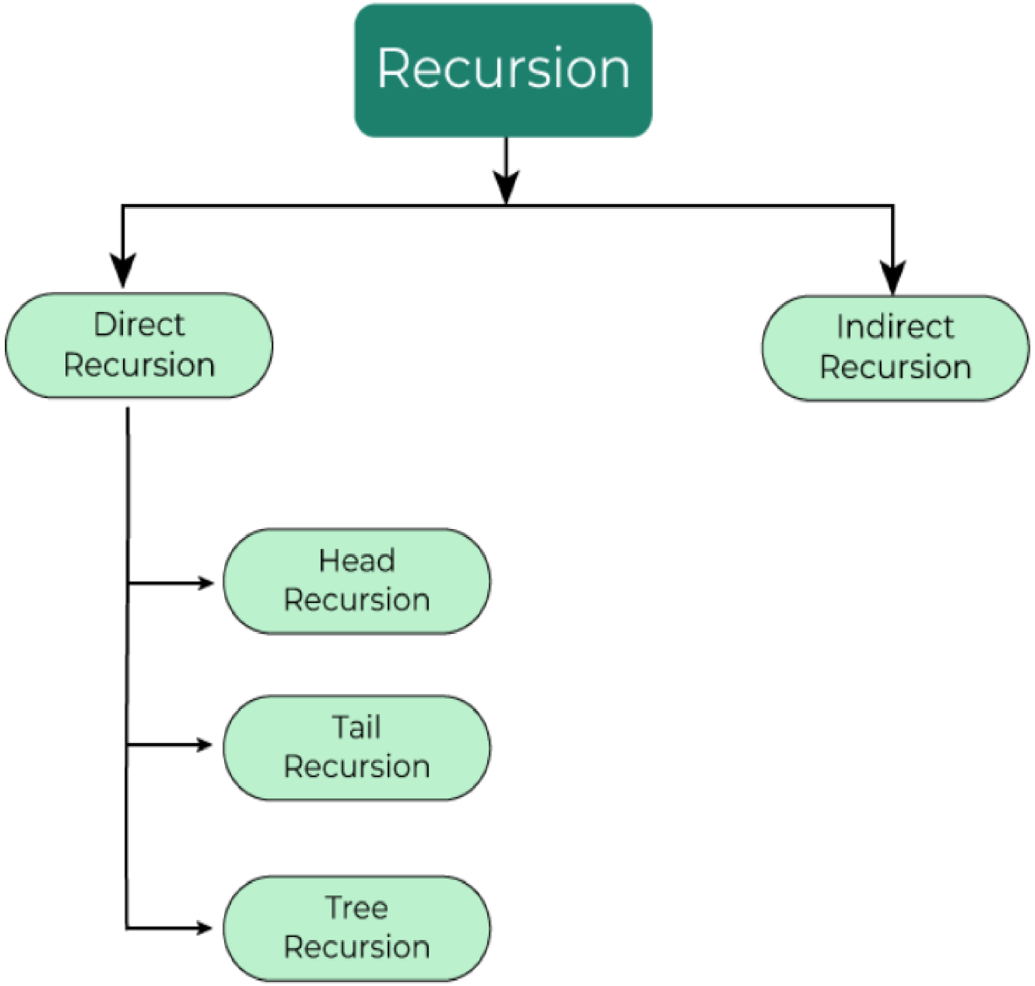
- **With the iterative solution, if we calculate fib(5), we have the following:**
  - fib(5) is calculated once
  - fib(4) is calculated once
  - fib(3) is calculated once
  - fib(2) is calculated once
  - fib(1) is calculated once
  - fib(0) is calculated once

- **Which one is efficient?**
  - Iterative or Recursive ??

Answer: Iterative

# 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.

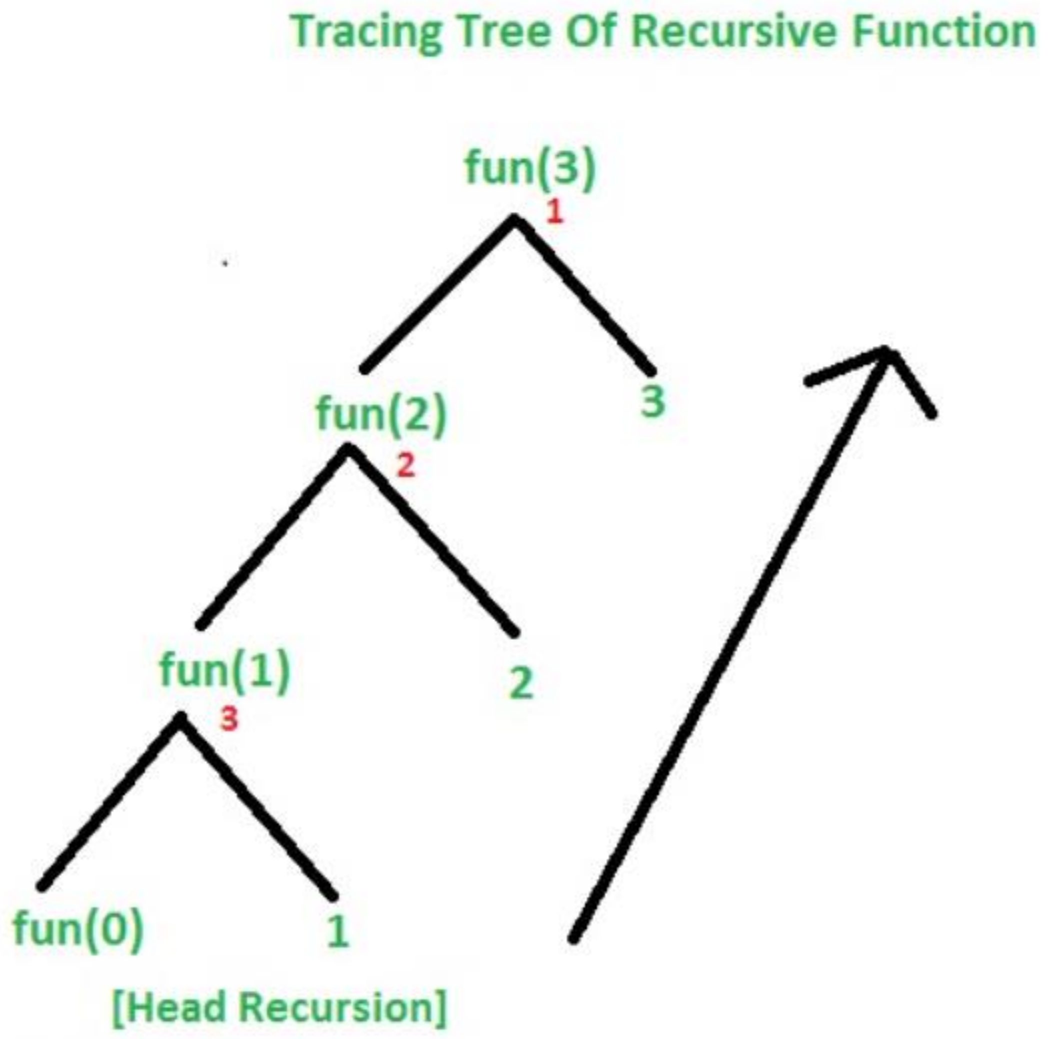


# Example of Head Recursion: **To print numbers from 1 to n**

```
// C program showing Head Recursion
#include<stdio.h>
// Recursive function
void fun(int n)
{           if (n > 0) {

            // First statement in the function
                        fun(n - 1);

                        printf(" %d”, n);
            }
}
```

```
int main()
{
            int x = 3;
            fun(x);
            return 0;
}
```

# Example of Head Recursion Cont..



**Output: 1 2 3**

# Example of Tail Recursion: **To print numbers from n to 1**

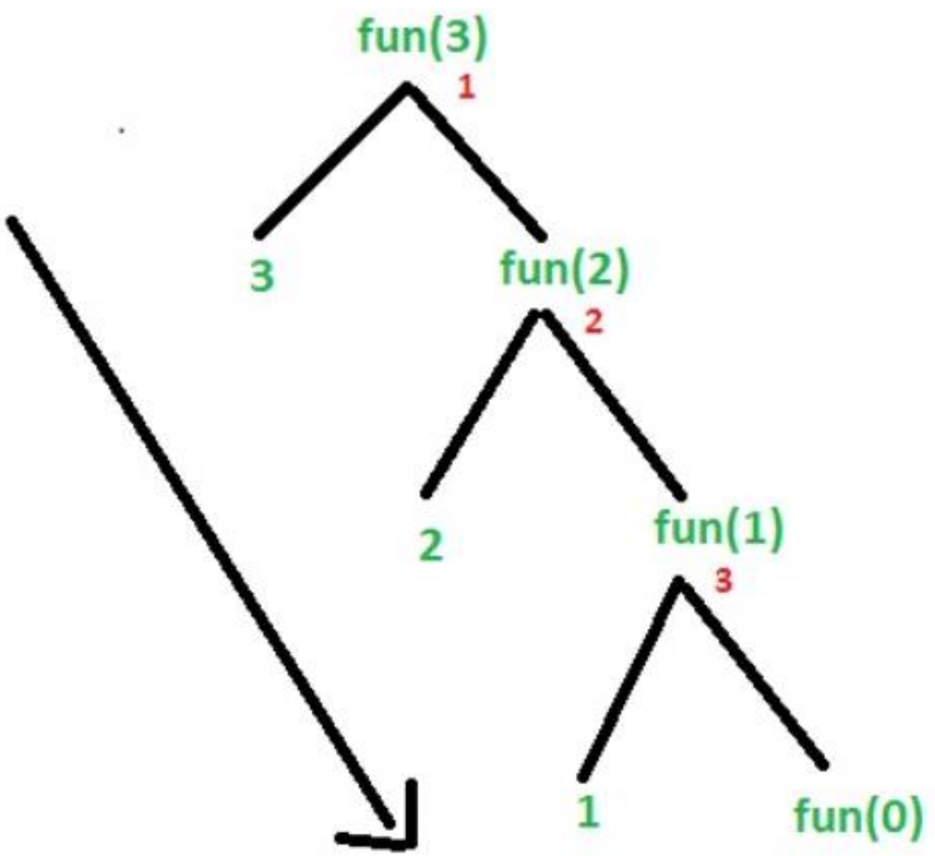
```
// Code Showing Tail Recursion
#include <stdio.h>
// Recursion function
void fun(int n)
{
        if (n > 0) {
                printf(" %d”, n);

        // Last statement in the function
                fun(n - 1);
        } }
```

```
int main()
{
        int x = 3;
        fun(x);
        return 0;
}
```

# Example of Tail Recursion Cont..

**Output: 3 2 1**

Tracing Tree Of Recursive Function



[Tail Recursion]

# Time & Space Complexity

- **Time complexity:** Quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

- **Time Complexity For Tail Recursion : O(n)**

- **Space Complexity:** The **amount of memory** an algorithm or program uses to solve a problem, based on the size of the input data

- **Space Complexity For Tail Recursion : O(n)**

# Convert Same code using Loops: **To print numbers from n to 1**

# Convert Same code using Loops

```
// Converting Tail Recursion into Loop
#include <stdio.h>

void fun(int y)
{
        while (y > 0) {
                printf(" %d”, y);

                y--;
        }
}
```

```
int main()
{
        int x = 3;
        fun(x);
        return 0;
}
```

**Output: 3 2 1**

**Time Complexity: O(n)**

**Space Complexity: O(1)**

# Recursion vs Iteration

## Recursion:

- Elegant for dividing problems.
- More readable for problems like tree traversals.

## Iteration:

- More efficient in terms of time and memory.
- Safer, avoids stack overflow risks.

Example of Tree Recursion

# Sum of Squares – [m, n]

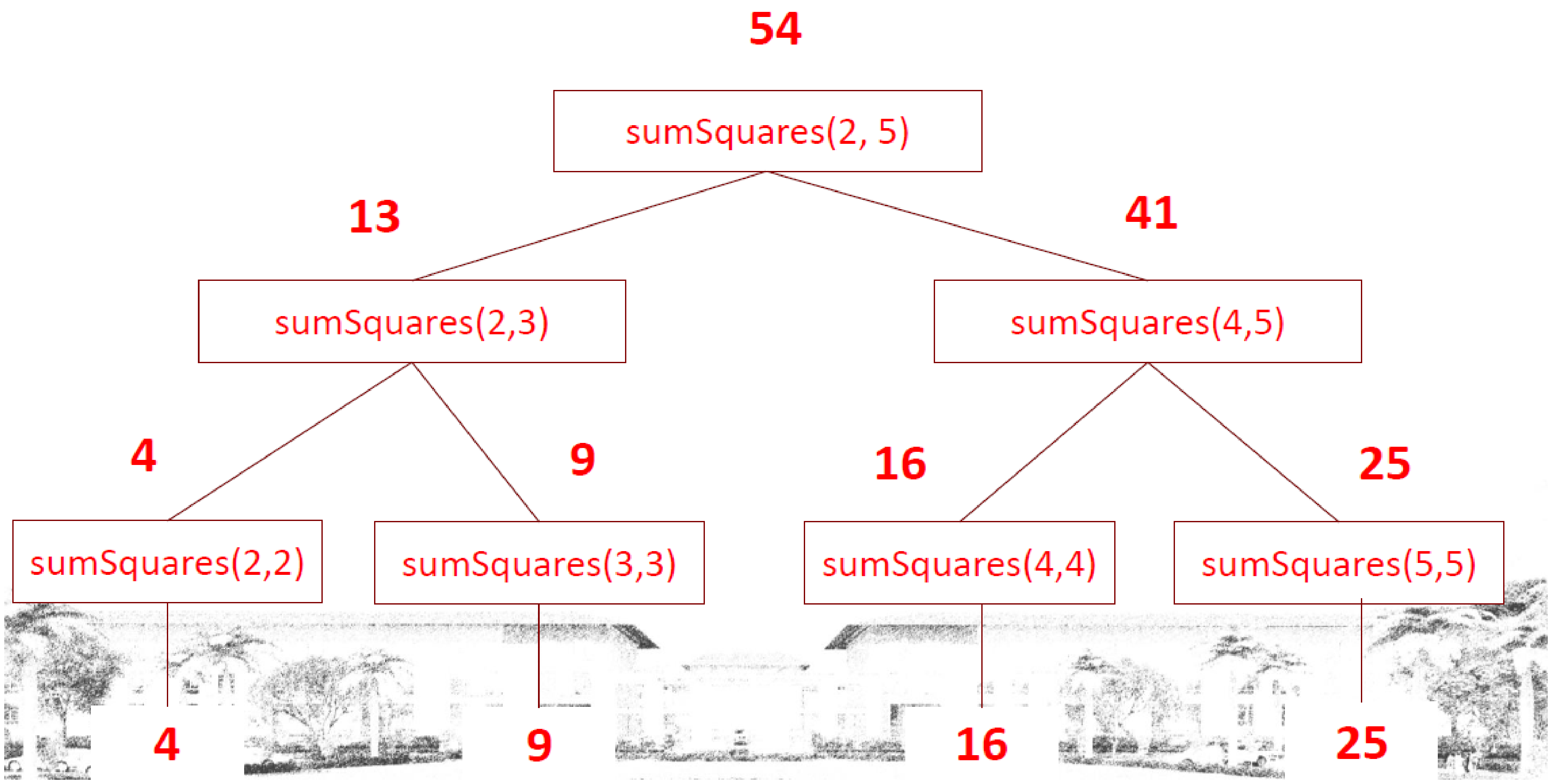
- Let us look into an illustrative example
- Given m and n are integers.
- Now write a recursive program to find the sum of squares of all integers between m and n (both inclusive)

**Example: Let m = 2, n = 5**

output = 2 * 2 + 3 * 3 + 4 * 4 + 5 * 5
= 4 + 9 + 16 + 25
= 54

# Sum of Squares of integers [m, n]

- Let us look at the illustration



54

sumSquares(2, 5)

13

41

sumSquares(2,3)

sumSquares(4,5)

4

9

16

25

sumSquares(2,2)

sumSquares(3,3)

sumSquares(4,4)

sumSquares(5,5)

4

9

16

25

# Sum of Squares

✧ Let us look into an illustrative example

```
int sumSquares (int m, int n) {
        int mid;
        if (m == n) return m*m;
        else {
                mid = (m+n)/2;
                return ( sumSquares(m, mid)
                              + sumSquares(mid+1, n) );
        }
}
```

**Test Cases:**

**a) m = 2, n = 5 →** output = 54
**b) m = 4, n = 8 →** output = 190
**c) m = 1, n = 6 →** output = 91
**d) m = 3, n = 7 →** output = 135

# C Program to Illustrate the Indirect Recursion

```
#include <stdio.h>

void functionA(int n)
{
    if (n < 1) {
        return;
    }
    printf("%d ", n);
    n = n - 1;
    // Indirect recursive call to functionB
    functionB(n);
}
void functionB(int n){
    if (n < 2) {
        return; }
 printf("%d ", n);
    n = n / 2;

    // Indirect recursive call to functionA
    functionA(n);
}

int main()
{

    // Function call
    functionB(20);

    return 0;
}
```

**Output: 20 10 9 4 3 1**

# Find GCD

**Write a Program to find the Greatest Common Divisor (GCD) of two numbers using recursion**

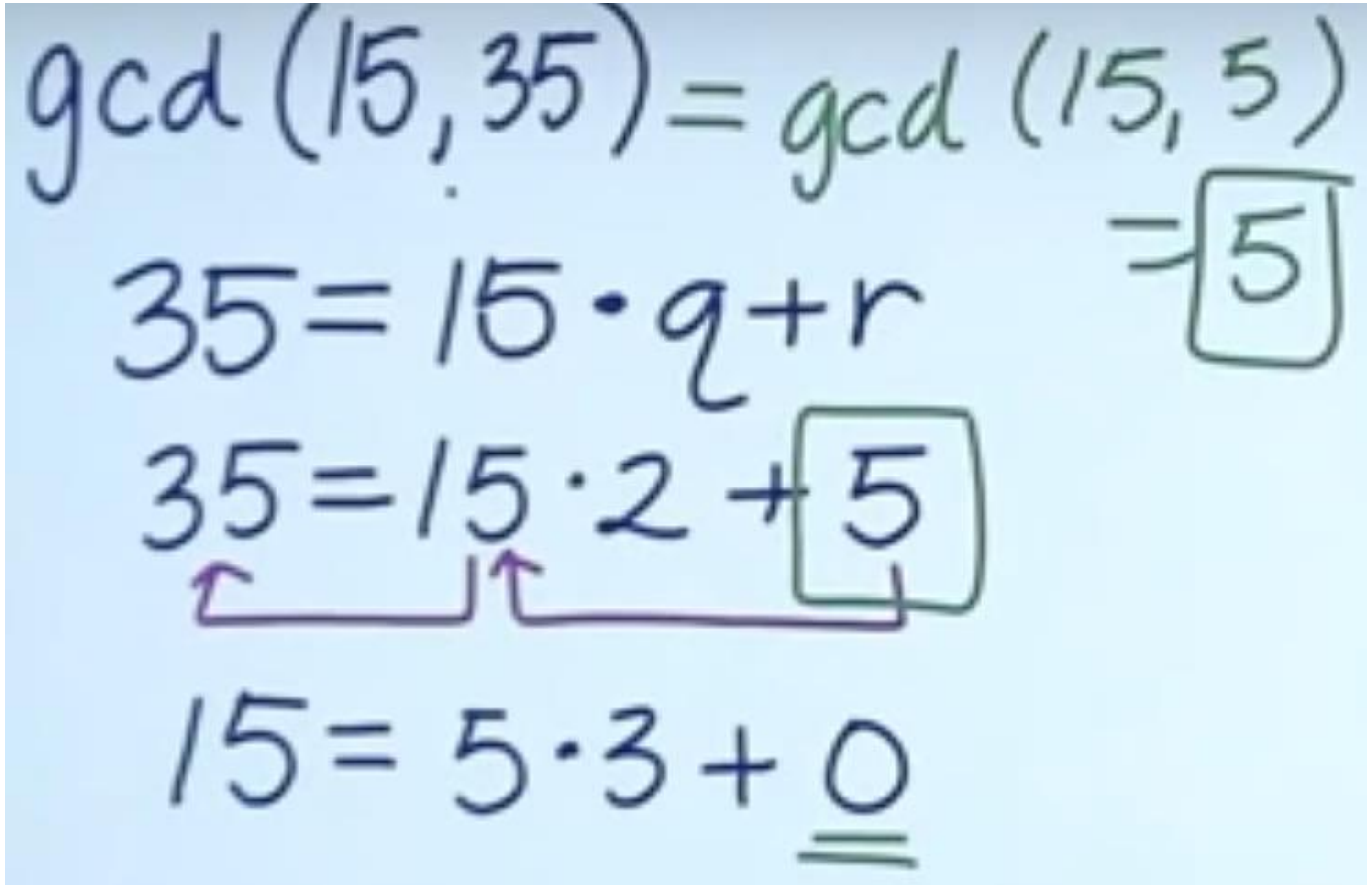
**Example:**

m = 9, n = 21

GCD(9, 21) = ???

Answer = 3

# Euclidean Algorithm



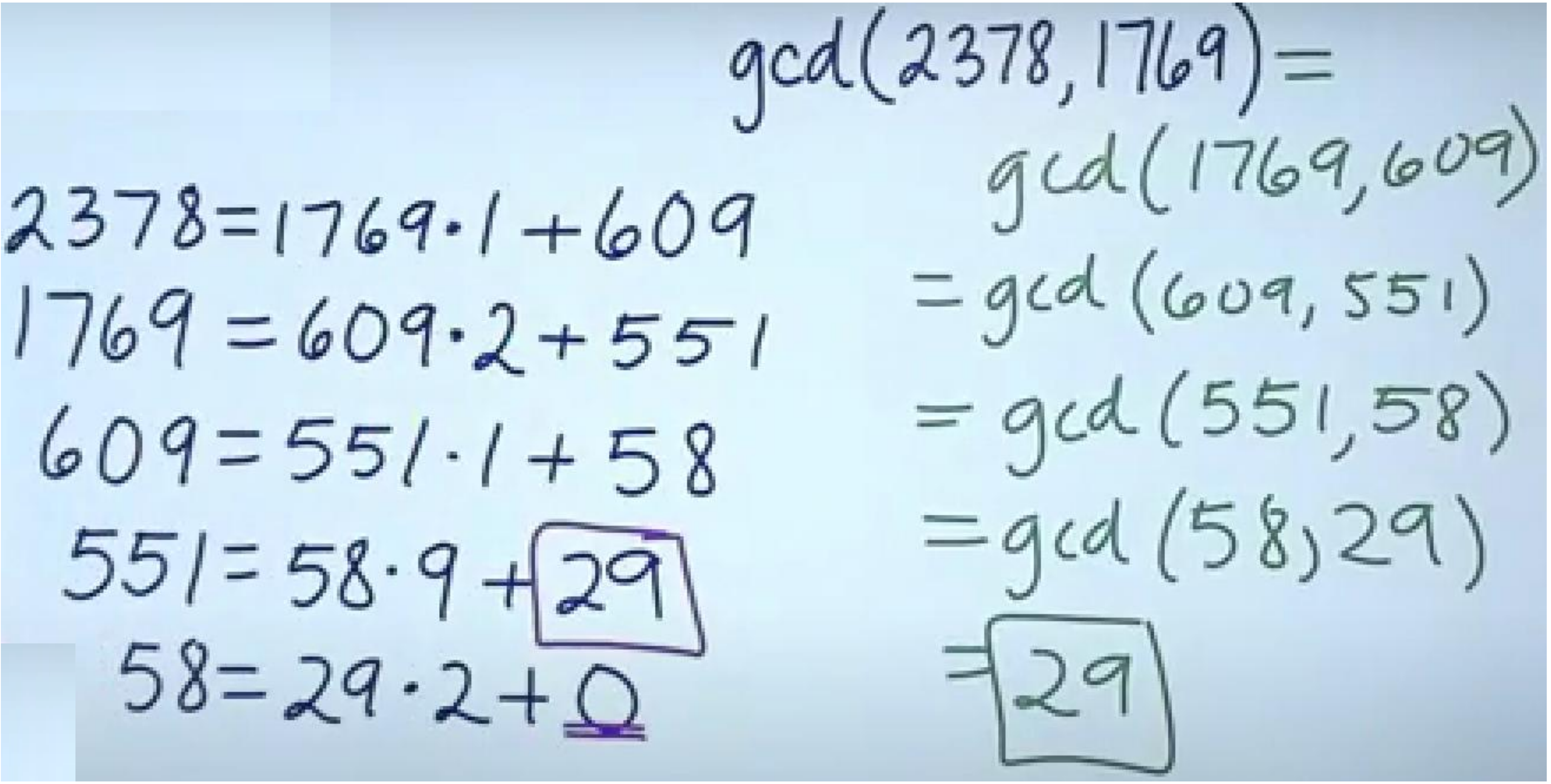
$$\gcd(15,35) = \gcd(15,5) = \boxed{5}$$

$$35 = 15 \cdot q + r$$

$$35 = 15 \cdot 2 + \boxed{5}$$

$$15 = 5 \cdot 3 + \underline{\underline{0}}$$

# Euclidean Algorithm



$$\gcd(2378,1769) =$$

$$2378 = 1769 \cdot 1 + 609$$
$$1769 = 609 \cdot 2 + 551$$
$$609 = 551 \cdot 1 + 58$$
$$551 = 58 \cdot 9 + \boxed{29}$$
$$58 = 29 \cdot 2 + \underline{\underline{0}}$$

$$\gcd(1769,609)$$
$$= \gcd(609,551)$$
$$= \gcd(551,58)$$
$$= \gcd(58,29)$$
$$= \boxed{29}$$

# Euclid Algoithm

**Steps:** Let us take two numbers ‘a’ and ‘b’

**1:** If $a < b$, exchange a and b

2: Divide a by b and get the remainder rem.
If rem = 0, report b as the GCD of a and b

# Compute GCD using Recursion

## Test Cases:

- ➔ m = 6 and n = 21

  Output: 3

- ➔ m = 30 and n = 95

  Output: 5

- ➔ m = 77 and n = 343

  Output: 7

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

int gcd(int m, int n) {
    if ( n == 0 ) return m;
    return gcd(n, m%n);
}
int main(int argc, char *argv[]) {
    int m = 18,  n = 12;
```

# Power Function

- Write a recursive function to calculate

  a^n = a * a * a * … * a (n times)

- (Do not use pow() or ^ operation)

- **Example:**
  - a = 2, n = 3
    - **Output = 8**
  - a = 4, n = 4
    - **Output = 256**
  - a = 3, n = 5
    - **Output = 243**

# Write a C program to print numbers power using Recursion

# Power Function - Solution

```
#include <stdio.h>
long int power(int base, int n) {
    if (n == 0) return 1;
    return base * power(base, n-1);
}

int main(int argc, char *argv[]) {
    int a = 2, n = 3;
    long int ret = power(a, n);
    printf("\nPower(%d, %d) = %ld\n", a, n, ret);
    return 0;
}
```

# Permutations

- **Problem:**
  - Write a recursive program to generate all permutations of a given string using recursion
  - Assume that all characters of the string are distinct

- **Example:**

  Input: “xyz”

- Output = ? (How many permutations?)

# Permutations – Test Case

- **Let us look at the following example**
  - Input: “xyz“
  - Length: 3
  - Number of permutations = 1 x 2 x 3 = 6
  - There are 6 different ways to represent 3 characters
- Output:

```
x y z        y x z        z y x
x z y        y z x        z x y
```

# Permutations - Solution

✧ **Solution:**

```
/* permute (set[begin], set[end]) */
void permute(char* set, int begin, int end) {
        int i;
        int range = end - begin;
        if (range == 1) {
                printf("set: %s\n", set);
        } else {
                for(i = 0; i < range; i++) {
                        swap(&set[begin], &set[begin+i]);
                        permute(set, begin+1, end);
                        swap(&set[begin], &set[begin+i]);
                }
        }
}
```

# Permutations – Test Case

- **Let us look at the following test case:**
  - Input: “wxyz“
  - Length: 4
  - Number of permutations = 1 x 2 x 3 x 4= 24
  - There are 24 different ways to represent 4 characters

  - Output:

wxyz
wxzy
wyxz
wyzx
wzyx
wzxy

xwyz
xwzy
xywz
xyzw
xzyw
xzwy

yxwz
yxzw
ywxz
ywzx
yzwx
yzxw

zxyw
zxwy
zyxw
zywx
zwyx
zwxy

# Applications of Recursion in C

- Simple application is like printing linked lists
- Tree-Graph Algorithms
- Mathematical Problems
- Divide and Conquer
- Dynamic Programming
- In Postfix to Infix Conversion
- Searching and Sorting Algorithms

# Advantages of Recursion

| | |
|---|---|
| Simple to Solve Complex Problems: | Divides problem into smaller, manageable subproblems. |
| Useful for Certain Data Structures: | Trees and graphs. |
| Cleaner Code for Problems like: | Factorial, Fibonacci series, GCD, etc. |

# Disadvantages of Recursion

**Performance Overhead:**

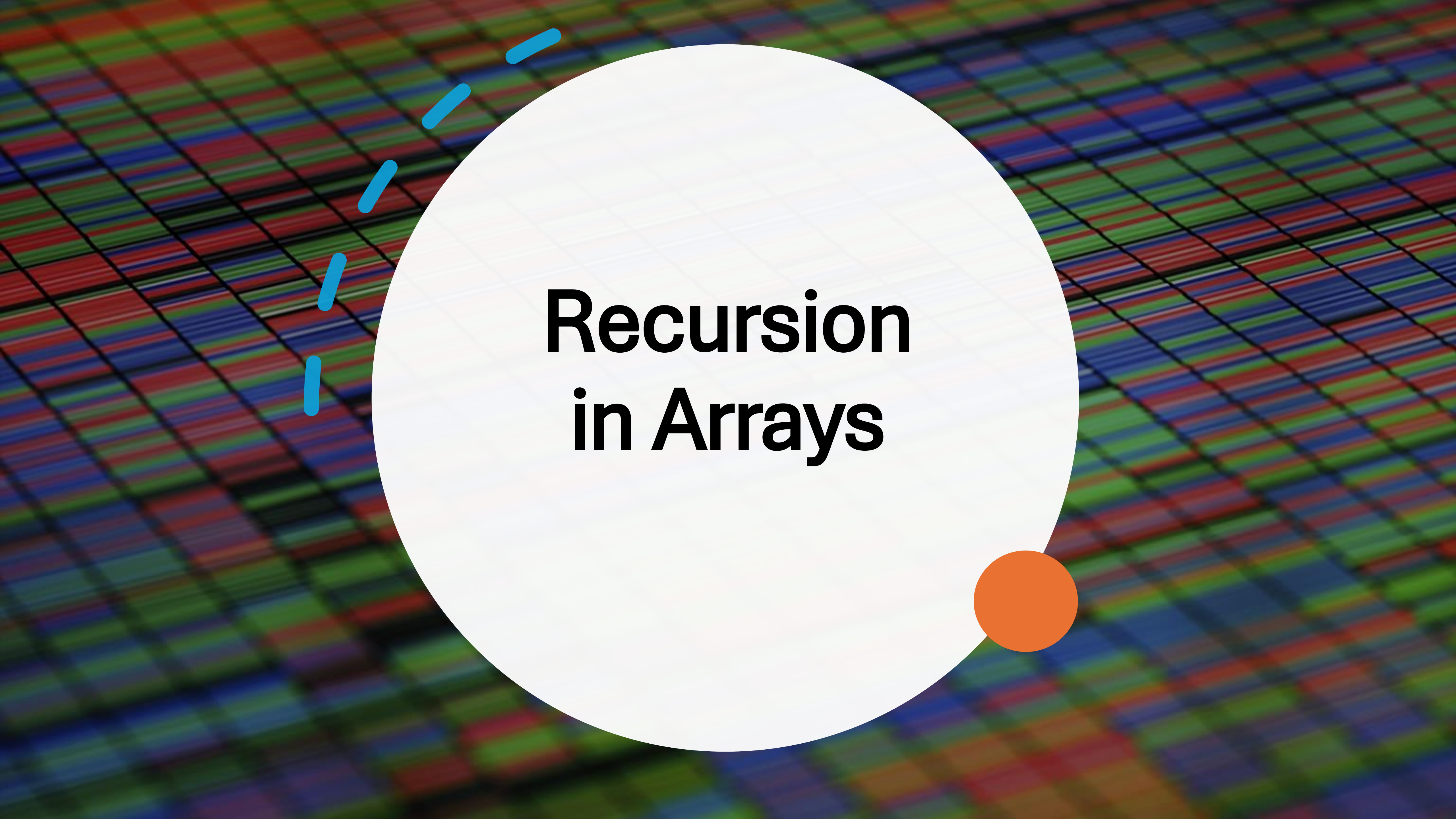
- Recursive calls add overhead to the call stack.

**Risk of Stack Overflow:**

- Deep recursion may lead to stack overflow if base case is not handled properly.
- ***Stack overflow is the error*** that occurs when the call stack of the program cannot store more data resulting in program termination.

**Slower Compared to Iteration (in some cases):**

- Recursion can be slower due to the additional function calls.



# Recursion in Arrays

Recursion in Arrays

# Sum of array elements – a[n]

- Compute the sum of the array elements

```
#include <stdio.h>
int sumArray(int a[], int n) {
	if (n == 0) return 0;
	return (a[n-1] + sumArray(a, n-1));
}
int main(int argc, char *argv[]) {
	int i, n, sum = 0;
	int a[] = {29, 27, 21, 36, 22};
	n = sizeof(a) / sizeof(a[0]);
	sum = sumArray(a, n);
	printf("\nSum = %d\n", sum);
	return 0;
}
```

Simple Recursion

# Sum of array elements – a[n]

- Compute the sum of array elements

```
#include <stdio.h>
int sumArray(int a[], int n, int sum) {
      if (n == 0) return sum;
      return sumArray(a, n-1, sum + a[n-1]);
}
int main(int argc, char *argv[]) {
      int i, n, sum = 0;
      int a[] = {14, 46, 33, 46, 44, 48, 36, 42, 27, 42};
      n = sizeof(a) / sizeof(a[0]);
      sum = sumArray(a, n, sum);
      printf("\nSum = %d\n", sum);
      return 0;
}
```

Tail Recursion

# Reverse elements of an array

**Using Recursion**

```
#include <stdio.h>
void printVals(int a[], int n) {
    for (int i = 0; i < n; i++ )
        printf(" %d ", a[i]);
    printf("\n");
}
void reverse(int a[], int left, int right) {
    if (left < right) {
        a[left] = a[left] + a[right];
        a[right] = a[left] - a[right];
        a[left] = a[left] - a[right];
        reverse(a, ++left, --right);
    }
}
```

```
int main(int argc, char *argv[]) {
    int i, n, sum = 0;
    int a[] = {14, 46, 33, 46, 44, 48};
    n = sizeof(a) / sizeof(a[0]);

    printVals(a, n);
    reverse(a, 0, n-1);
    printVals(a, n);

    return 0;
}
```

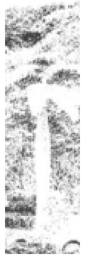
# Linear Search using recursion

✧ **Check whether a given element is present in the array or not?**

```
int search(int a[], int start, int end, int k) {
    if (a[start-1] ==  k ) return 1;
    if (start == end ) return 0;
    return search(a, ++start, end, k);
}
```

```
void printVals(int a[], int n) {
    for (int i = 0; i < n; i++ )
        printf(" %d ", a[i]);
    printf("\n");
}
```

```
int main(int argc, char *argv[]) {
    int i, n = 10, sum = 0, k = (argc > 1) ? atoi(argv[1]) : 11;
    int a[] = {14, 46, 33, 46, 44, 48, 36, 42, 27, 42};
    printVals(a, n);
    int ret = search(a, 1, n, k);
    if (ret == 1) printf("%d is found!\n", k);
    else printf("%d is NOT found!!\n", k);
    return 0;
}
```



# Linear Search – Another version

- Check whether a given element is present in the array or not?

```
int search(int a[], int n, int k) {
    if (a[0] ==  k ) return 1;
    if ( n == 1 ) return 0;
    return search(&a[1], n-1, k);
}
```

```
void printVals(int a[], int n) {
    for (int i = 0; i < n; i++ )
        printf(" %d ", a[i]);
    printf("\n");
}
```

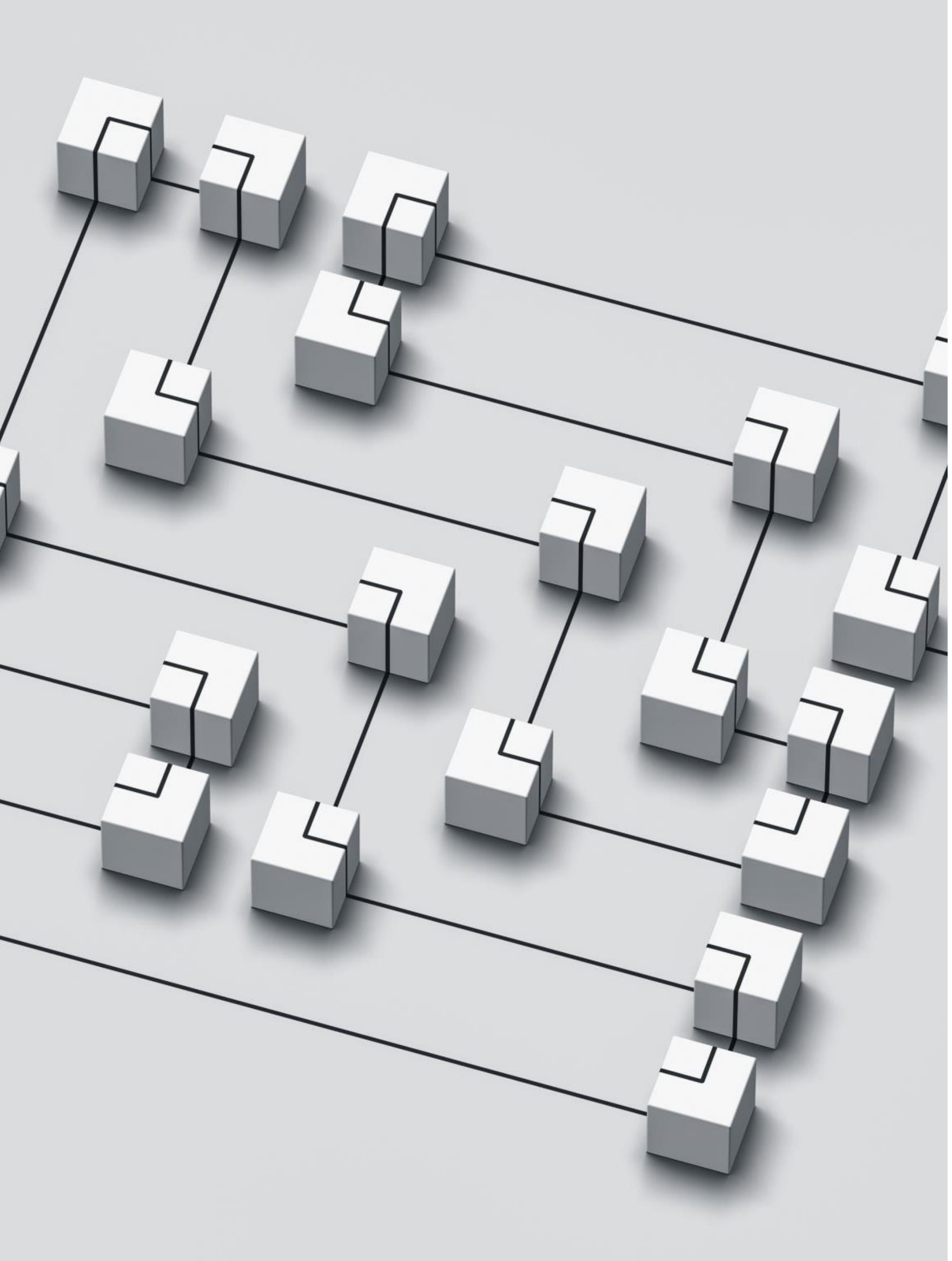
```
int main(int argc, char *argv[]) {
    int i, n = 10, sum = 0, k = (argc > 1) ? atoi(argv[1]) : 11;
    int a[] = {14, 46, 33, 41, 44, 48, 36, 42, 27, 43};
    printVals(a, n);
    int ret = search(a, n, k);
    if (ret == 1) printf("%d is found!\n", k);
    else printf("%d is NOT found!!\n", k);
    return 0;
}
```

# Exercises on Recursion

- ✧ Write a recursive function to print the values of a 1D-array of integers
- ✧ Power Set: Write a recursive program to generate all subsets of a given string (Given a set represented as a string of characters)
- ✧ Write a recursive program to check whether a given number is palindrome or not?
- ✧ Write a recursive program to check whether a given number of prime or not?
- ✧ Write a recursive program to return the sum of the first N natural numbers
- ✧ Given a binary number as string, write a recursive program to find its decimal equivalent.
- ✧ Assume sufficiently a long integer. Write a recursive program to sum up all prime digits of that number.
- ✧ Write a Recursive function that prints all numbers less than N which consist of digits, each can be of 1 or 3 or multiples of 3 (if N = 20, then output is 19, 16, 13, 11, 9, 6, 3, 1)

# Exercises on Recursion in Arrays

- Write Recursive functions for Linear Search in 2-D arrays using Recursion
- Write a C Program to Print Binary Equivalent of an Integer using Recursion
- Write a function to find the Biggest Number in an Array of Numbers using Recursion
- Write a function to perform Matrix Multiplication using Recursion
- Write a function to everse the String using Recursion
- Write a function to find reverse of a number using Recursion
- Write a function to copy one string to another using Recursion
- Write a function to find Least Common Multiple (LCM) of a given number using Recursion
- Write a function to convert a Number Decimal System to Binary System using Recursion
- Write a function to find the first capital Letter in a string using Recursion



# Upcoming Slides

- Pointers