📘 Session 03: Recursion Applications

✅ What is Recursion?

Recursion is a process where a function **calls itself** to solve a smaller version of the same problem.

🔁 Real-Life Analogy:

Looking in two mirrors facing each other – reflection inside reflection – similar to function calling itself again and again.

🔸 How Does Recursion Work?

1. A recursive function calls itself with a **smaller problem**.
2. There must be a **base case** to **stop** the recursion.
3. Each call is **stored in memory** using a **function call stack**.

✅ Application 1: Factorial (n!)

📌 Definition:

Factorial of `n` is the product of all numbers from `n` to `1`.
Formula: `n! = n × (n-1) × (n-2) × ... × 1`

🔁 Iterative Version

```
int factorialIter(int n) {
    int result = 1;
    for(int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

✅ 1. Factorial Using Recursion

```
#include <iostream>
using namespace std;

// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 1) return 1; // Base case
    return n * factorial(n - 1); // Recursive case
}

int main() {
    int n;
    cout << "Enter a number to find factorial: ";
    cin >> n;

    if (n < 1) {
        cout << "Factorial not defined for numbers less than 1.";
    } else {
        cout << "Factorial of " << n << " is: " << factorial(n) << endl;
    }

    return 0;
}
```

🔁 Recursive Version

```
int factorialRec(int n) {
    if (n == 0 || n == 1)
        return 1; // base case
    else
        return n * factorialRec(n - 1); // recursive call
}
```

🧠 Dry Run: `factorialRec(4)`

Call	Value Returned
factorialRec(4) → 4 * factorialRec(3)	
factorialRec(3) → 3 * factorialRec(2)	
factorialRec(2) → 2 * factorialRec(1)	
factorialRec(1) → 1 (base case)	
Returning: 2×1=2 → 3×2=6 → 4×6=24	

🧠 Memory Stack Diagram

```
factorialRec(4)
 └── factorialRec(3)
      └── factorialRec(2)
           └── factorialRec(1) → return 1
          return 2
      return 6
 return 24
```

✅ Application 2: Fibonacci Sequence

🔢 Definition:

`F(0) = 0`, `F(1) = 1`,
`F(n) = F(n-1) + F(n-2)`

🔁 Recursive Version

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

🔁 Memoized Version (Optimized)

```
int fibMemo(int n, int memo[]) {
    if (n <= 1)
        return n;
    if (memo[n] != -1)
        return memo[n];

    memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo);
    return memo[n];
}
```

Usage:

```
int main() {
    int n = 10;
    int memo[n+1];
    fill_n(memo, n+1, -1);
    cout << "Fibonacci(" << n << ") = " << fibMemo(n, memo) << endl;
}
```

🧠 Dry Run: `fib(4)`

```
fib(4)
├── fib(3)
|    ├── fib(2)
|    |    ├── fib(1) = 1
|    |    └── fib(0) = 0
|    └── = 1
├── fib(2)
|    ├── fib(1) = 1
|    └── fib(0) = 0
Result: 3
```

⚠️ Problem with Recursive Fibonacci

- Calls same values repeatedly

- Time complexity = **O(2^n)**
  ✅ Memoization reduces to **O(n)**

✅ 2. Fibonacci Series Using Recursion

```cpp
#include <iostream>
using namespace std;

// Recursive function to get nth Fibonacci number
int fibonacci(int n) {
    if (n == 0) return 0;  // Base case
    if (n == 1) return 1;  // Base case
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive call
}

int main() {
    int n;
    cout << "Enter the number of terms in Fibonacci series: ";
    cin >> n;

    cout << "Fibonacci series: ";
    for (int i = 0; i < n; i++) {
        cout << fibonacci(i) << " ";
    }
    cout << endl;

    return 0;
}
```

✅ Application 3: GCD Using Recursion (Euclidean Algorithm)

📌 GCD: Greatest Common Divisor

The largest number that divides both numbers.

🔁 Recursive Version

```
int gcd(int a, int b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}
```

🧠 Dry Run: `gcd(48, 18)`

Call	a	b	a % b
gcd(48, 18)	48	18	12
gcd(18, 12)	18	12	6
gcd(12, 6)	12	6	0
gcd(6, 0)	6	0	—

✅ **Final GCD = 6**

🧠 Memory Stack for `gcd(48, 18)`

```
gcd(48, 18)
 └── gcd(18, 12)
      └── gcd(12, 6)
           └── gcd(6, 0) → return 6
          return 6
      return 6
 return 6
```

✅ 3. GCD (Greatest Common Divisor) Using Recursion

```
#include <iostream>
using namespace std;

// Recursive function to calculate GCD
int gcd(int a, int b) {
    if (b == 0) return a;       // Base case
    return gcd(b, a % b);       // Recursive call
}

int main() {
    int a, b;
    cout << "Enter two numbers to find GCD: ";
    cin >> a >> b;

    cout << "GCD of " << a << " and " << b << " is: " << gcd(a, b) << endl;

    return 0;
}
```

📝 Practice Worksheet

🔹 A. Factorial

1. Find `factorialRec(5)` → __________
2. Fill stack diagram for `factorialRec(3)`

🔹 B. Fibonacci

1. Find `fib(5)` (no memo) → __________

2. What's the time complexity of basic recursive Fibonacci?

🔹 **C. GCD**

1. `gcd(36, 24)` → __________

2. Trace the recursive calls for `gcd(30, 12)`

✅ 4. Tower of Hanoi Using Recursion

```
#include <iostream>
using namespace std;

// Recursive function to solve Tower of Hanoi
void towerOfHanoi(int n, char source, char helper, char destination) {
    if (n == 1) {
          cout << "Move disk 1 from " << source << " to " << destination <<
endl;
        return;
    }

     towerOfHanoi(n - 1, source, destination, helper); // Move n-1 disks to
helper
     cout << "Move disk " << n << " from " << source << " to " << destination
<< endl;
     towerOfHanoi(n - 1, helper, source, destination); // Move n-1 disks to
destination
}

int main() {
    int n;
    cout << "Enter number of disks: ";
    cin >> n;

    cout << "Steps to solve Tower of Hanoi:" << endl;
       towerOfHanoi(n, 'A', 'B', 'C'); // A = source, B = helper, C =
destination
```

```
    return 0;
}
```

📋 Quiz: Recursion Applications

🔁 Factorial | 🔢 Fibonacci | 🔗 GCD | 📚 Stack Dry Run

✅ Section A: True or False

Q1. A recursive function must always have a base case.
Q2. Recursive Fibonacci is more efficient than iterative.
Q3. The GCD of two numbers can be found using recursion.
Q4. Recursion uses a call stack to track function calls.

✅ Section B: Conceptual Questions

Q5. What is the base case in a recursive factorial function?
→ ______________________________

Q6. What will happen if a recursive function does not reach its base case?
→ ______________________________

Q7. Why is memoization used in Fibonacci?
→ ______________________________

Q8. Write the recursive formula for the Fibonacci sequence.
→ ______________________________

✅ Section C: Output Prediction

Q9. What will be the output of this code?

```
int factorial(int n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
```

```
}
cout << factorial(4);
```

→ ________

Q10. What is the output of `gcd(48, 18)` using recursion?
→ ________

Q11. How many recursive calls will be made in `factorial(3)`?

✅ Section D: Dry Run Stack Trace

Fill in the blanks to trace the recursive stack.

Q12. Dry run for `factorial(3)`

Function Call	Returned Value
factorial(3) →	
factorial(2) →	
factorial(1) →	
Return values:	

Q13. Dry run for `gcd(30, 12)`

Function Call	Returned Value
gcd(30, 12) →	
gcd(12, 6) →	
gcd(6, 0) →	
Return values:	

✅ Section E: Code Completion

Q14. Complete the recursive GCD function:

```
int gcd(int a, int b) {

    if (_________)
        return a;
    return ____________;
}
```

✅ Section F: Multiple Choice

Q15. Which of the following has time complexity $O(2^n)$?
a) Iterative factorial
b) Recursive factorial
c) Basic recursive Fibonacci
d) Memoized Fibonacci

Q16. What is the base case for Fibonacci?
a) `F(0) = 1`
b) `F(0) = 0`, `F(1) = 1`
c) `F(n) = 0`
d) None of the above

🧾 Answer Key (Teacher Only)

<details> <summary>Click to expand</summary>

A1. True
A2. False
A3. True
A4. True

Q5. `n == 0` or `n == 1`
Q6. Infinite recursion → stack overflow
Q7. To avoid recomputing same values → improves speed
Q8. `F(n) = F(n-1) + F(n-2)`

Q9. 24
Q10. 6
Q11. 3 calls

Q12.

Function Call	Returned Value

factorial(3) →	6
factorial(2) →	2
factorial(1) →	1

Q13.

Function Call	Returned Value
gcd(30, 12) →	6
gcd(12, 6) →	6
gcd(6, 0) →	6

Q14.

cpp
CopyEdit
```
if (b == 0)
return gcd(b, a % b);
```

Q15. c) Basic recursive Fibonacci
Q16. b) `F(0) = 0`, `F(1) = 1`