# ALGORITHMS AND DATA STRUCTURES LECTURE 1 - RECURSION

Zhamiyev Abulkhair
zhamiyev.abulkhair@astanait.edu.kz

## CONTENT

# RECURSION OVERVIEW

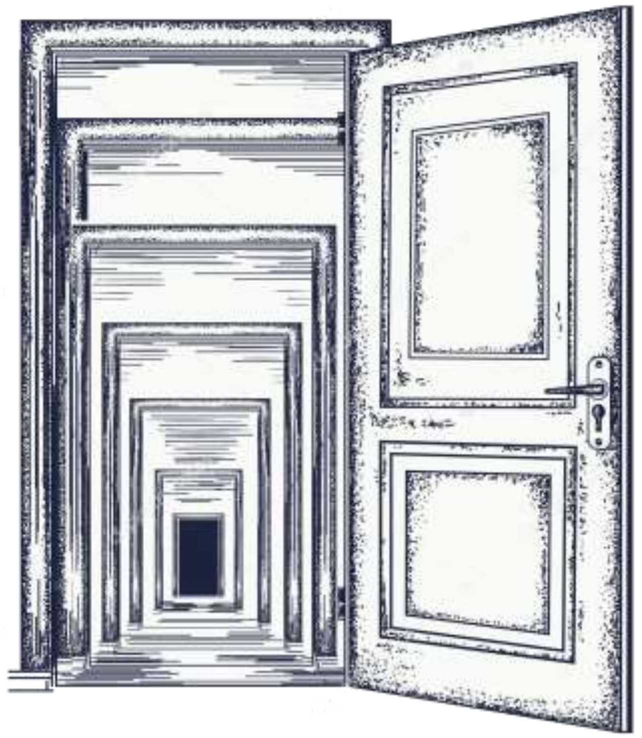
Recursion is the process of repeating items in a self-similar way

A way to design solutions by Divide-and-Conquer

- Reduce a problem to simpler versions of the same problem

A programming technique where a function calls itself

- Must have at least 1 **base case**
- **Base case** means that there exist one or more inputs for which the function produces a result trivially (without recurring)
- Must solve the same problem on some other input with the goal of simplifying the larger problem input



## SIMPLE EXAMPLE

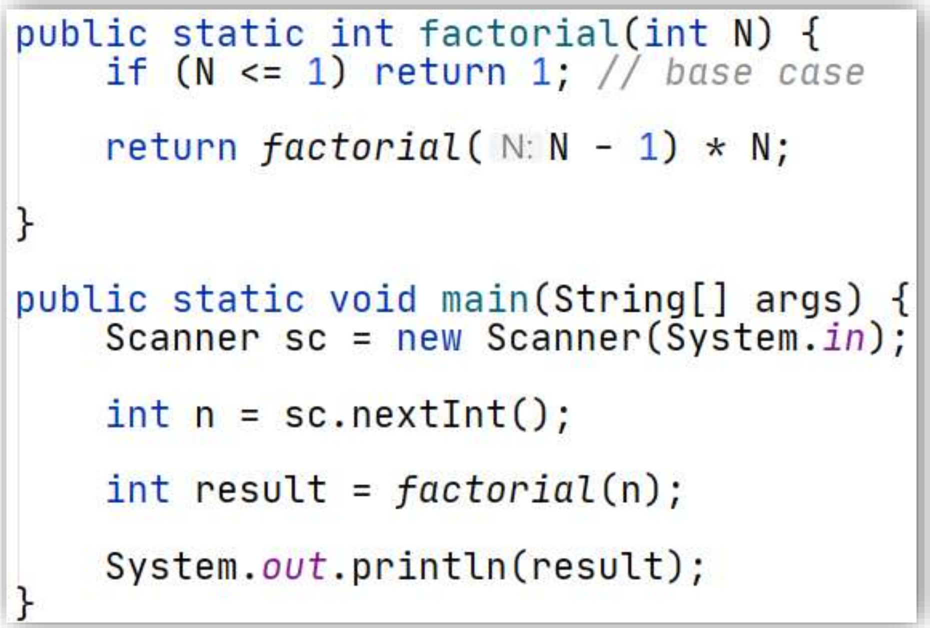
$N! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot \ldots \cdot N$ fact(N)

$N! = \underbrace{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot \ldots \cdot (N-1)} \cdot N$ fact(N-1) * N

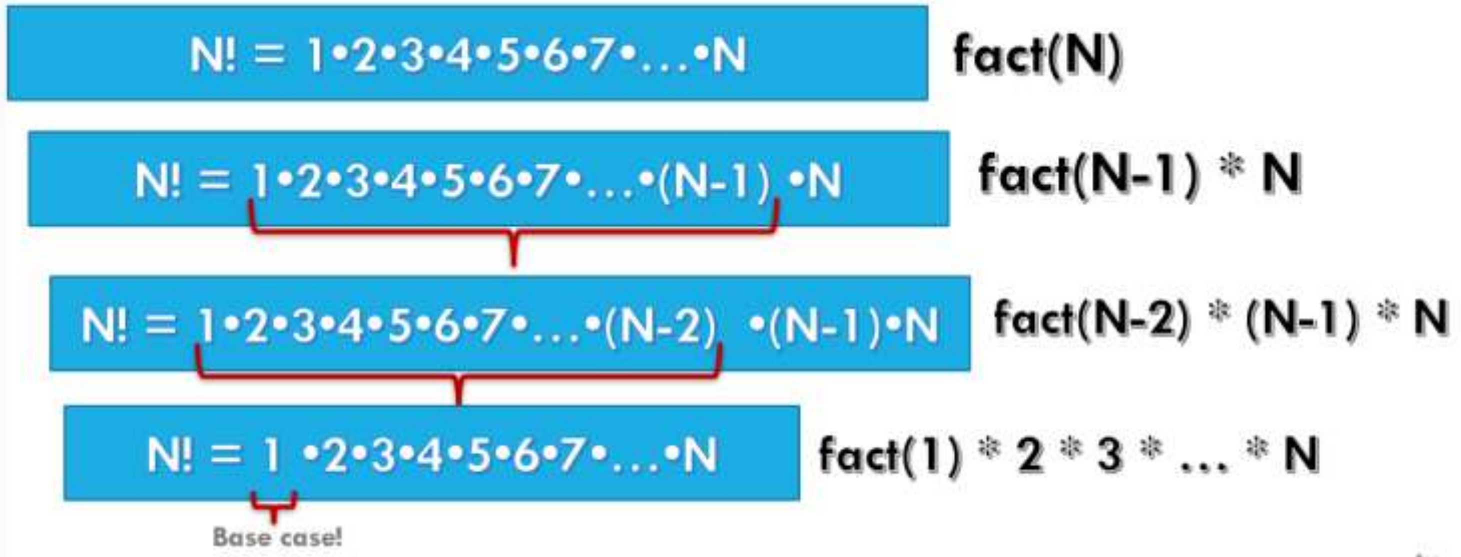
$N! = \underbrace{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot \ldots \cdot (N-2)} \cdot (N-1) \cdot N$ fact(N-2) * (N-1) * N

$N! = \underbrace{1}_{\text{Base case!}} \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot \ldots \cdot N$ fact(1) * 2 * 3 * ... * N

## SIMPLE EXAMPLE



```java
public static int factorial(int N) {
    if (N <= 1) return 1; // base case

    return factorial( N: N - 1) * N;

}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int n = sc.nextInt();

    int result = factorial(n);

    System.out.println(result);
}
```



N! = 1•2•3•4•5•6•7•...•N
fact(N)
N! = 1•2•3•4•5•6•7•...•(N-1) •N
fact(N-1) * N
N! = 1•2•3•4•5•6•7•...•(N-2) •(N-1)•N
fact(N-2) * (N-1) * N
N! = 1 •2•3•4•5•6•7•...•N
fact(1) * 2 * 3 * ... * N
Base case!

## HOW IT WORKS?

Recursion is no different than a function call

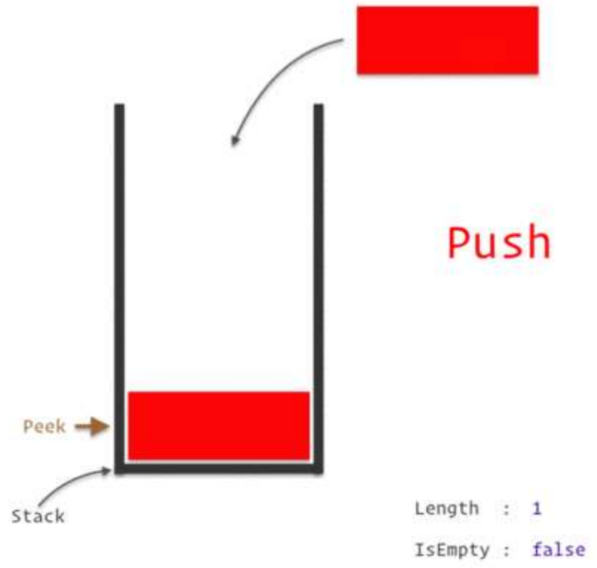
Every function call creates a new frame (block) inside the stack

The system keeps track of the sequence of method calls that have been started but not finished yet (active calls)

- order matters

Recursion pitfalls

- miss base-case (infinite recursion, stack overflow)
- no convergence (solve recursively a problem that is not simpler than the original one)



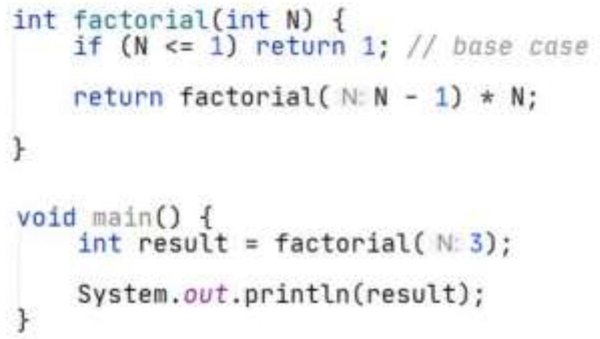
# FUNCTION CALL AND STACK

When you run a program, the computer creates a **stack** for you

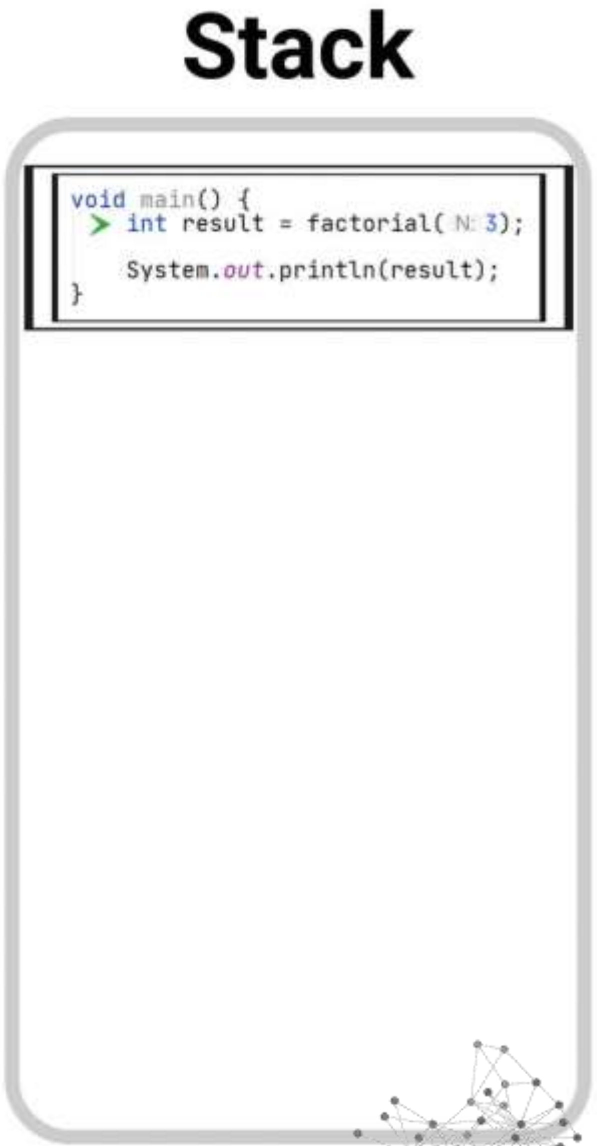
Each time you invoke a method, the method is placed to the stack

A stack is a **last-in/first-out** memory structure. The first item referenced or removed from a stack is always the last item entered into the stack

If some function call has produced an excessively long chain of recursive calls, it can lead to **stack overflow**



```
int factorial(int N) {
    if (N <= 1) return 1; // base case

    return factorial( N: N - 1) * N;
}

void main() {
    int result = factorial( N: 3);

    System.out.println(result);
}
```



Stack

```
void main() {
  > int result = factorial( N: 3);

    System.out.println(result);
}
```

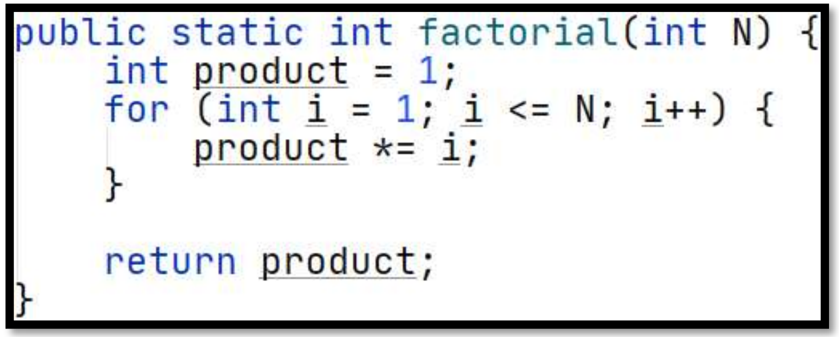
# ITERATION VS RECURSION

## Iteration

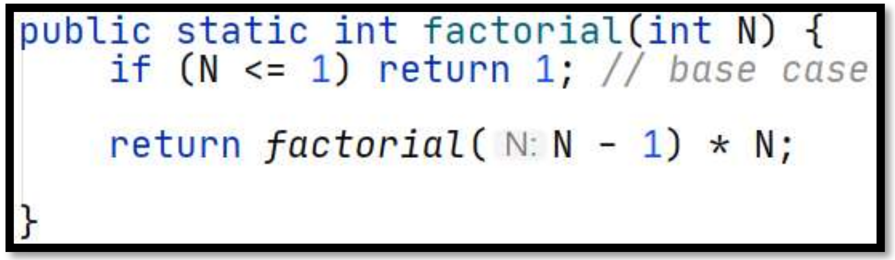
- Uses repetition structures (for, while or do…while)
- Repetition through explicitly use of repetition structure
- Terminates when loop-continuation condition fails
- Controls repetition by using a counter

## Recursion

- Uses selection structures (if, if…else or switch)
- Repetition through repeated method calls
- Terminates when base case is satisfied
- Controls repetition by dividing problem into simpler one



```
public static int factorial(int N) {
    int product = 1;
    for (int i = 1; i <= N; i++) {
        product *= i;
    }

    return product;
}
```



```
public static int factorial(int N) {
    if (N <= 1) return 1; // base case

    return factorial( N: N - 1) * N;

}
```

# ITERATION VS RECURSION

Repetition

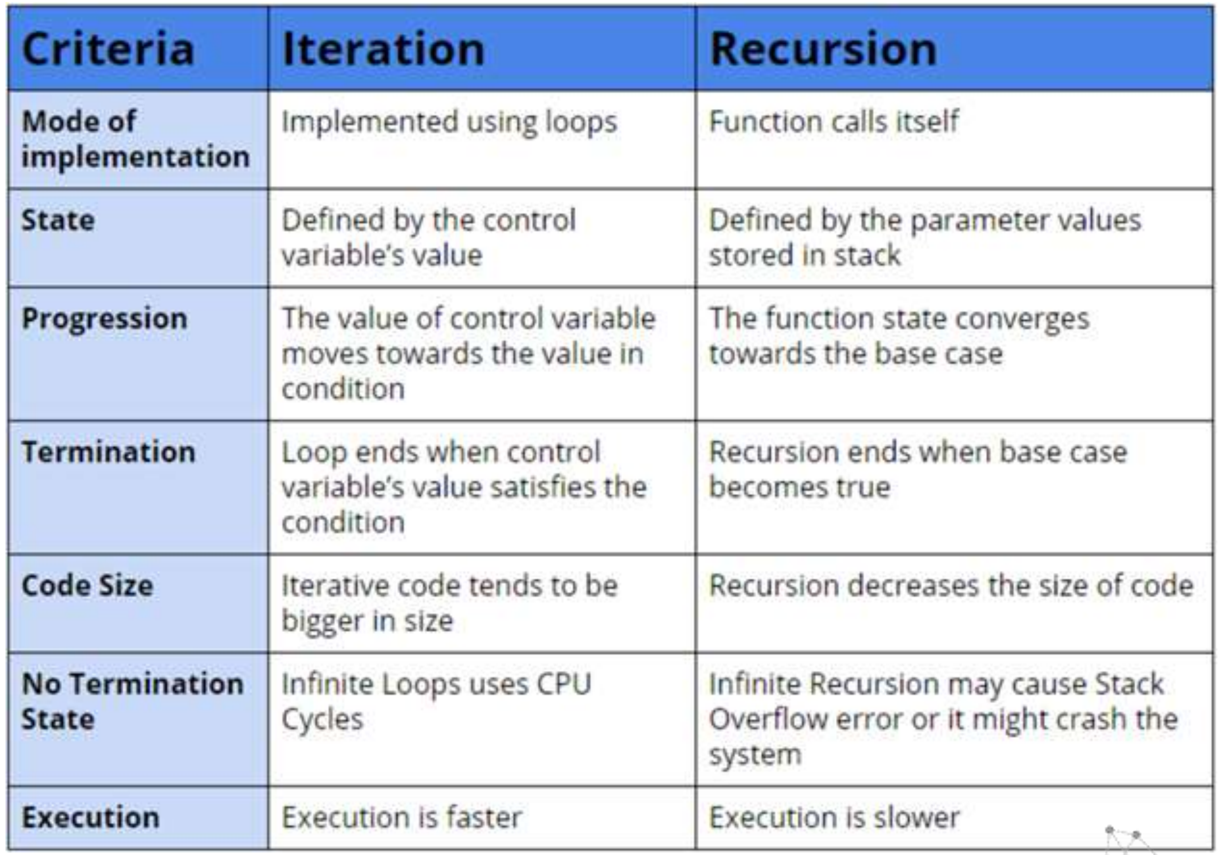
- Iteration: explicit loop
- Recursion: repeated function calls

Termination

- Iteration: loop condition fails
- Recursion: base case recognized

Both can have infinite loops

Balance between performance (iteration) and good software engineering (recursion)



| Criteria | Iteration | Recursion |
|---|---|---|
| **Mode of implementation** | Implemented using loops | Function calls itself |
| **State** | Defined by the control variable's value | Defined by the parameter values stored in stack |
| **Progression** | The value of control variable moves towards the value in condition | The function state converges towards the base case |
| **Termination** | Loop ends when control variable's value satisfies the condition | Recursion ends when base case becomes true |
| **Code Size** | Iterative code tends to be bigger in size | Recursion decreases the size of code |
| **No Termination State** | Infinite Loops uses CPU Cycles | Infinite Recursion may cause Stack Overflow error or it might crash the system |
| **Execution** | Execution is faster | Execution is slower |

## HOW TO CREATE A RECURSIVE ALGORITHM?

1. Think about a problem at a high level of abstraction
2. Figure out the **base case** for the program
3. Redefine the answer in terms of a simpler sub-problem
4. Combine the results in the formulation of the answer

## FIBONACCI SOLUTION

Fibonacci sequence

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 …
- Each element is the sum of previous two
- Starts from 0 and 1

**Task:** Find the Fibonacci number at the given position

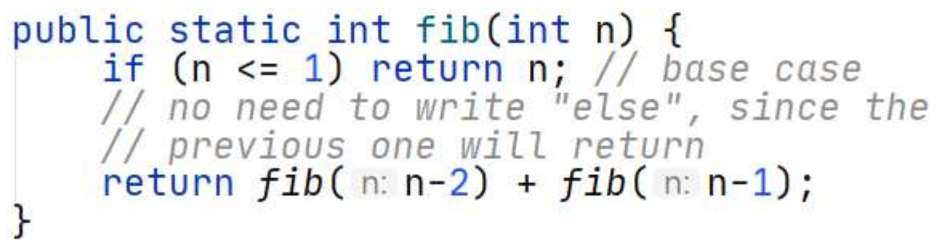
**Example:**

- 3rd element is 5
- 6th element is 8

**Solution:**

fib(n) = fib(n-2) + fib(n-1)

fib(0) = 0 and fib(1) = 1 // this is a base case



```
public static int fib(int n) {
    if (n <= 1) return n; // base case
    // no need to write "else", since the
    // previous one will return
    return fib( n: n-2) + fib( n: n-1);
}
```

## LITERATURE

Algorithms, 4th Edition, by Robert Sedgewick and Kevin Wayne, Addison-Wesley

- Chapter 1.1

Grokking Algorithms, by Aditya Y. Bhargava, Manning

- Chapter 3

# GOOD LUCK!