Recursion

Recursion is a programming technique where a method calls itself to solve the smaller version of original problem.

To solve recursion problems we always need a **base condition** which helps to stop the process.

If we don't provide base condition then it leads to `stackoverflow` error.

> *1. WAP to print numbers from 1-50.*

```
public static void printNumberFrom1to50(int count)
{
    if(count>50)
        return;
    System.out.println(count);
    printNumberFrom1to50(++count);
}
```

Method Tracing: `sumOfNumberFrom1toN(int n)`

Purpose

This method calculates the sum of all integers from 1 to `n` using recursion.

Base Case

- When `n == 0`, returns `0` (terminates recursion)

Recursive Case

- Returns `n + sumOfNumberFrom1toN(n - 1)`

Tracing Steps

Example 2: `n = 5`

Call Stack Depth	n Value	Evaluation	Return Value
1	5	5 + sumOfNumberFrom1toN(4)	5 + 10 = 15
2	4	4 + sumOfNumberFrom1toN(3)	4 + 6 = 10
3	3	3 + sumOfNumberFrom1toN(2)	3 + 3 = 6
4	2	2 + sumOfNumberFrom1toN(1)	2 + 1 = 3
5	1	1 + sumOfNumberFrom1toN(0)	1 + 0 = 1
6 (Base Case)	0	return 0	0

Final Return Value: `15`
(5 + 4 + 3 + 2 + 1 + 0 = 15)

Key Characteristics

1. **Recursive Pattern**:
    - Each call adds current `n` to sum of all previous numbers
    - Decrements `n` until reaching base case
2. **Stack Behavior**:
    - Maximum stack depth = `n + 1` (including base case)
    - Stack unwinds from base case upward
3. **Time Complexity**: O(n)
   (Performs exactly `n` recursive calls)
4. **Space Complexity**: O(n)
   (Due to call stack memory usage)

Visual Representation (n=3)

```
sumOfNumberFrom1toN(3)
│
├─ 3 + sumOfNumberFrom1toN(2)
│  │
│  ├─ 2 + sumOfNumberFrom1toN(1)
│  │  │
│  │  ├─ 1 + sumOfNumberFrom1toN(0)
│  │  │  │
│  │  │  └─ 0 (base case)
│  │  │
│  │  └─ 1 + 0 = 1
│  │
│  └─ 3 + (2 + 1) = 6
│
└─ Final result: 6
```

Limitations

1. **Stack Overflow Risk**: For large `n` (typically > 10,000 in Java)

> *2. WAP to print number from 1-n.*

```java
public static void printNumberFrom1toN(int n)
{
    if(n==0)
        return;
    printNumberFrom1toN(--n);
    System.out.println(n+1);
}
```

> *3. WAP to print number from n-1.*

```java
public static void printNumberFromNto1(int n)
{
```

```
    if(n==0)
        return;
    System.out.println(n);
    printNumberFromNto1(--n);
}
```

> *4. WAP to print sum of number from 1-n.*

```
public static int sumOfNumberFrom1toN(int n)
{
    if(n==1)
        return 1;
    return n + sumOfNumberFrom1toN(--n);
}
```

> *5. WAP to print multiplication of number from 1-n(Factorial).*

```
public static int findFactorial(int n)
{
    if(n==1)
        return 1;
    return n*findFactorial(--n);
}
```

> *6. WAP to print the sum of even number from 1-n.*

```
public static int evenSum(int n)
{
    if(n==1)
        return 0;
    if(n==2)
        return 2;
    if(n%2==0)
        return n+evenSum(--n);
    else
        return evenSum(--n);
}
```

Method Tracing: `evenSum(int n)`

Purpose

This method calculates the sum of all even numbers from 1 to `n` using recursion.

Base Cases

- `n == 1` → returns `0` (no even numbers)
- `n == 2` → returns `2` (only even number)

Recursive Cases

- If `n` is even: `n + evenSum(n-1)`
- If `n` is odd: `evenSum(n-1)`

Tracing Steps

Example 1: `n = 6`

Call Stack Depth	n Value	Condition Evaluation	Action	Return Value
1	6	n%2 == 0 (even)	6 + evenSum(5)	6 + 6 = 12
2	5	n%2 != 0 (odd)	evenSum(4)	6
3	4	n%2 == 0 (even)	4 + evenSum(3)	4 + 2 = 6
4	3	n%2 != 0 (odd)	evenSum(2)	2
5	2	n == 2 (base case)	return 2	2

Final Return Value: `12`
(6 + 4 + 2 = 12, odd numbers 5 and 3 are skipped)

Key Characteristics

1. **Recursive Pattern**:
   - Even `n`: Adds current `n` and processes `n-1`
   - Odd `n`: Skips current `n` and processes `n-1`
2. **Termination**:
   - Base cases at `n=1` and `n=2` stop recursion
   - Decrements `n` until reaching base case
3. **Behavior**:

   ```
   evenSum(5) → evenSum(4) → 4 + evenSum(3) → evenSum(2) → 2
   ```

4. **Time Complexity**: O(n)
   (Makes `n` recursive calls in worst case)
5. **Space Complexity**: O(n)
   (Call stack depth grows linearly with `n`)

Visual Representation (n=6)

```
evenSum(6)
│
├─ 6 (even) → 6 + evenSum(5)
│  │
│  ├─ 5 (odd) → evenSum(4)
│  │  │
│  │  ├─ 4 (even) → 4 + evenSum(3)
```

```
|  |  |  |
|  |  |  ├─ 3 (odd) → evenSum(2)
|  |  |  |  |
|  |  |  |  └─ 2 (base case) → 2
|  |  |  |
|  |  |  └─ 4 + 2 = 6
|  |  |
|  |  └─ Returns 6
|  |
|  └─ 6 + 6 = 12
|
└─ Final result: 12
```

Limitations

1. **Pre-decrement Operator**: Using `--n` modifies `n` before recursion
2. **Stack Overflow**: Risk for large `n` (like all recursive solutions)

> *7. WAP to print the sum of odd number from 1-n.*

```java
public static int oddSum(int n)
{
    if(n==1)
        return 1;
    if(n%2==1)
        return n+oddSum(--n);
    else
        return oddSum(--n);
}
```

> *8. WAP to find power of a number.*

```java
public static int powOfNumber(int base,int power)
{
    if(power == 0)
        return 1;
    if(power==1)
        return base;
    return base * powOfNumber(base, --power);
}
```

Method Tracing: `powOfNumber(int base, int power)`

Purpose

This method calculates `base` raised to the power of `power` using recursion.

Base Case

- When `power == 1`, returns `base` (terminates recursion)

- When `power == 0`, returns `1` (terminates recursion)

Recursive Case

- Returns `base * powOfNumber(base, --power)`

Tracing Steps

Example 1: `base = 2`, `power = 3`

Call Stack Depth	base	power	Condition	Action	Return Value
1	2	3	power != 1	2 * powOfNumber(2, 2)	2 * 4 = 8
2	2	2	power != 1	2 * powOfNumber(2, 1)	2 * 2 = 4
3 (Base Case)	2	1	power == 1	return 2	2

Final Return Value: `8`
($2^3 = 2 \times 2 \times 2 = 8$)

Key Characteristics

1. **Recursive Pattern**:
   - Multiplies `base` with result of reduced power
   - Decrements `power` until reaching base case
2. **Termination**:
   - Stops when `power` reaches 1
   - Returns `base` itself when power is 1
3. **Time Complexity**: O(power)
   (Performs exactly `power` recursive calls)
4. **Space Complexity**: O(power)
   (Call stack depth equals `power`)

Edge Cases

Input (base, power)	Output	Notes
(x, 1)	x	Any number to power 1 is itself
(1, n)	1	1 to any power is 1
(0, n)	0	0 to any positive power is 0
(x, 0)	1	1 to power is 0 always 1

Visual Representation (2^3)

```
powOfNumber(2, 3)
│
├─ 2 * powOfNumber(2, 2)
│  │
```

```
│   ├─ 2 * powOfNumber(2, 1)
│   │   │
│   │   └─ return 2 (base case)
│   │
│   └─ 2 * 2 = 4
│
└─ 2 * 4 = 8
```

Limitations

1. **Large Powers**: Risk of stack overflow and integer overflow