Data Structures and Algorithms

Recursion

Recursion and Recursive methods

- Recursion occurs when a method calls itself to solve a simpler version of the problem.
- Recursive methods provide an alternative to the iterative (loop) method for implementing the repetition of activities (by invoking itself).
- With each recursive call, the problem is different from, and simpler than, the original problem.

How to write a recursive method?

- **Base case** (also called base step or anchor)
  - Values of the input variables for which we perform no recursive calls are called base cases (There should be at least one base case).
  - Every possible chain of recursive calls must reach a base case!
- **Recursive call** (also called recursive step)
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Recursion examples

- In the next few slides, you will see four examples (factorial, power, Fibonacci, and sum) of how recursion is used in programming.

- You should take out a sheet of paper and a pencil and try to work each of them out **by hand** ☺.

Example 1: Factorial

- The factorial of n is the product of the numbers from 1 to n:

  $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots (n-2) \cdot (n-1) \cdot n$

  By convention 0! = 1.

  For example, 4! = 1.2.3.4 = 24.

- A recursive definition for n! is as follows:

  Base step: $= 1$ (if $n = 0$)

  Recursive step: $\mathbf{n!} = n \cdot \mathbf{(n-1)!}$ (if $n > 0$)

Example 1: Factorial

Base step:

Recursive step:

$$n! = \begin{cases} = 1 \text{ (if } n = 0) \\ = n \cdot \mathbf{(n-1)!} \text{ (if } n > 0) \end{cases}$$

To compute the value of n! (e.g. 5!), we start calling the method recursively obtaining expressions involving lower and lower values of n, until arriving at the base case (n = 0).

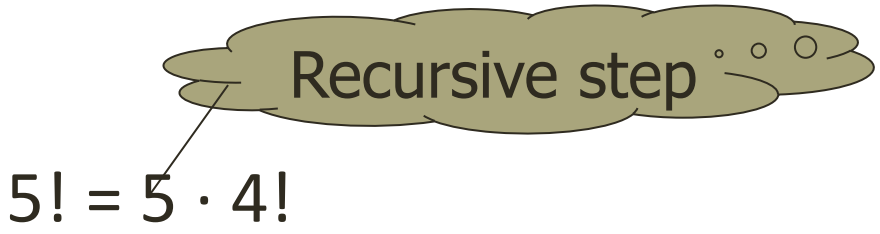
For example, 5! =

Example 1: Factorial

Base step:

Recursive step:

$$\mathbf{n!} = \begin{cases} = 1 \text{ (if } n = 0) \\ = n \cdot \mathbf{(n-1)!} \text{ (if } n > 0) \end{cases}$$

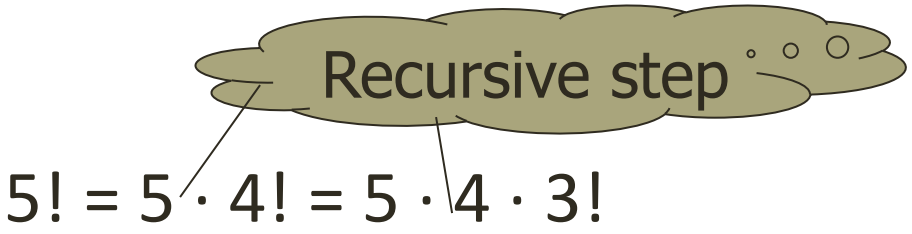


Example 1: Factorial

Base step:

Recursive step:

$$\mathbf{n!} = \begin{cases} = 1 \text{ (if } n = 0) \\ = n \cdot \mathbf{(n-1)!} \text{ (if } n > 0) \end{cases}$$

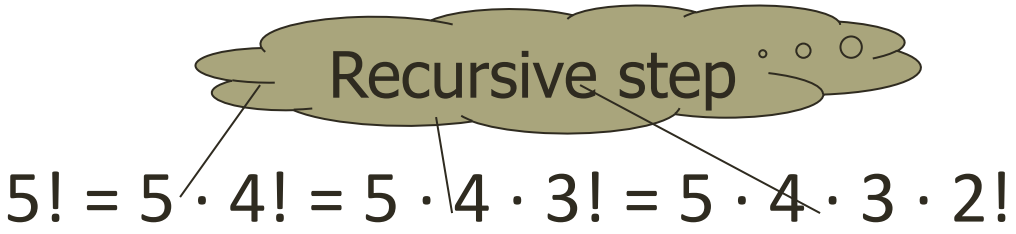


Example 1: Factorial

Base step:

Recursive step:

$$\mathbf{n!} = \begin{cases} = 1 \text{ (if } n = 0) \\ = n \cdot \mathbf{(n-1)!} \text{ (if } n > 0) \end{cases}$$

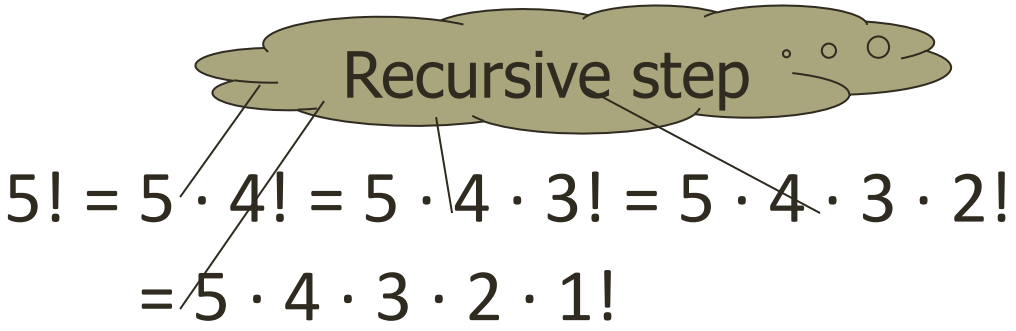


Example 1: Factorial

Base step:

Recursive step:

$$\mathbf{n!} = \begin{cases} = 1 \text{ (if } n = 0) \\ = n \cdot \mathbf{(n-1)!} \text{ (if } n > 0) \end{cases}$$

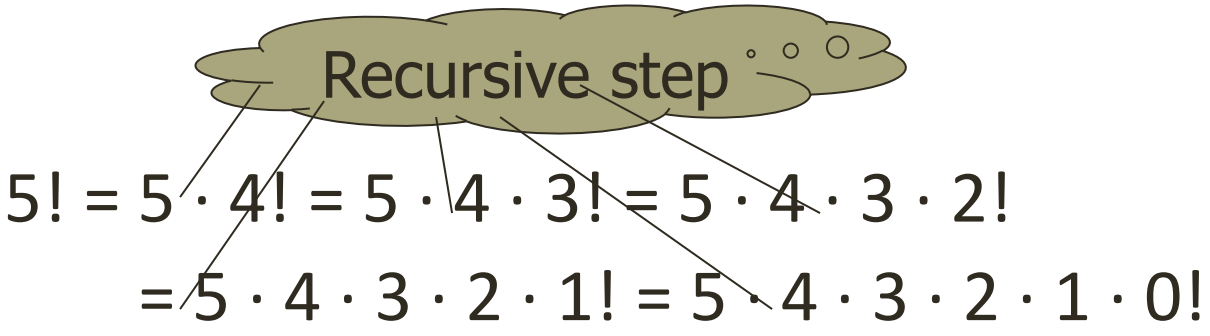


Example 1: Factorial

Base step:

Recursive step:

$$\mathbf{n!} = \begin{cases} = 1 \text{ (if } n = 0) \\ = n \cdot \mathbf{(n-1)!} \text{ (if } n > 0) \end{cases}$$

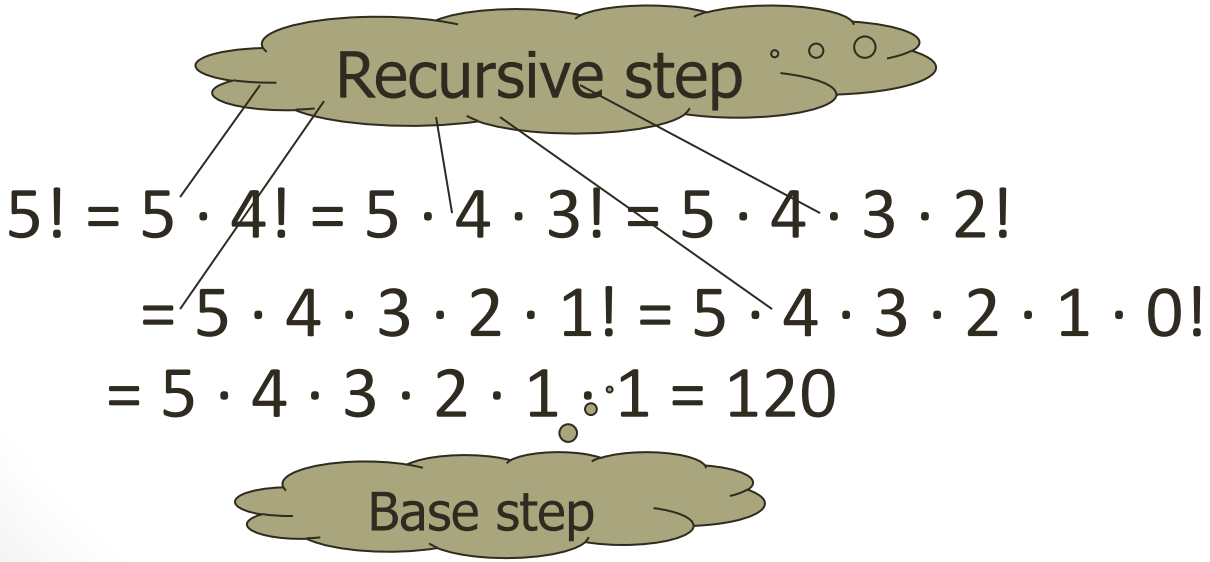


Example 1: Factorial

Base step:

Recursive step:

$$\mathbf{n!} = \begin{cases} = 1 \text{ (if } n = 0) \\ = n \cdot \mathbf{(n-1)!} \text{ (if } n > 0) \end{cases}$$



Writing recursive methods

- Our first example is writing a recursive method to compute the factorial of an integer n.
- **Base step** → when does the repetition stop?
- We know that 0! = 1 → base step / base case / anchor
- **Recursive step** → calling the method again and again, each time changing the value until it converges to the base step (n = 0).
- n! = n * (n-1)! then (n-1)! = (n-1) * (n-2)! and so on until we get to 1*0!.
- For most values of n, the value n! is larger than the maximum **int** value therefore we use the **long** type instead of the **int** type.

Non Recursive (Iterative) Factorial method

// n! = 1 * 2 * 3 * 4 * … * (n –2) * (n –1) * n

```
public static long factorial (int n) {
   int fact = 1;
   for (int i = 2; i <= n; i++)
       fact *= i; // same as fact = fact * i;
   return fact;
}

public static void main(String[] args){
   System.out.println(factorial(4));
}
```

Time Complexity of the Iterative Factorial method

```
public static long factorial (int n) {
  int fact = 1;
  for (int i = 2; i <= n; i++)
     fact *= i;
  return fact;
}
```

There are n-1 iterations of the loop in the iterative approach, so its time complexity is **O(n)**.

Recursive Factorial method

```
// n! = n * (n –1)!
// 0! = 1

public static long factorial (int n) {
   if (n == 0) // base step
      return 1;
   return n * factorial(n - 1); // recursive step
}

public static void main(String[] args){
     System.out.println(factorial(4));
}
```

Recursive Factorial method

```
public static long factorial (int n) {
   if (n == 0) // base step or anchor
     return 1;
   return n * factorial(n - 1); // recursive step
}
```

factorial(4) **returns 24**

→ is 4 = 0? No

→ 4 * **factorial(3)**

→ is 3 = 0? No

→ 3 * **factorial(2)**

→ is 2 = 0? No

→ 2 * **factorial(1)**

→ is 1 = 0 ? No

→ 1 * **factorial(0)**

→ is 0 = 0? **Yes**!

→ **return 1**

1

1*1= 1

2*1= 2

3*2= 6

4*6= 24

Time Complexity of the Recursive Factorial method

```
public static long factorial (int n) {
  if (n == 0)
    return 1;
  return n * factorial(n - 1);
}
```

There are n recursive calls in the recursive approach, and each call uses constant time operations. Thus, the time complexity of factorial using recursion is **O(n)**.

This is the same as the time complexity using the iterative approach.

Example 2: Power

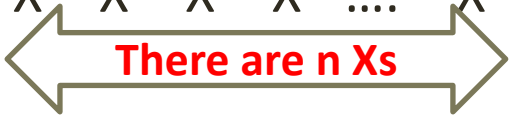
- Iterative Definition of Power:

  $X^0 = 1$

  $X^1 = X$

  $X^2 = X*X$

  $X^n = X * X * X * X* \ldots. * X$

- Recursive definition of Power:

  $X^0 = 1$ (Base step)

  $X^n = X * X^{n-1}$ (Recursive step)

Example 2: Power

- If we call power(2,3), it returns 8
  $2^3 = 2 * 2 * 2 = 8$

- To look at it recursively

  $2^3 = 2^* 2^2 = 2 * 2 * 2^1 = 2 * 2 * 2 * 2^0 = 2 * 2 * 2 * 1 = 8$

- If n = 0, the result is 1. x^0 is 1 for any x.
- Otherwise we have to repeat the multiplication with a smaller value of n as in **x* power(x, n-1)**

Non Recursive (Iterative) Power method

```
public static int power(int x, int n) {
   int pow = 1;
   for (int i = 1; i <= n; i++) // n times
      pow *= x;
   return pow;
}

public static void main(String[] args){
   System.out.println(power(2,3));
}
```

Recursive Power method

```
// Assuming n >= 0
public static int power(int x, int n) {
   if (n == 0)   // anchor
      return 1;
   return x * power(x, n-1);   // recursive step
}
public static void main(String[] args){
   System.out.println(power(2,3));
}
```

Recursive Power method

```
public static int power(int x, int n) {
   if (n == 0)   // anchor
      return 1;
   return x * power(x, n-1);   // recursive step
}
```

Power(2,3) **returns 8**

→ is 3 = 0 ? No

2 * 4 = 8 → 2 * **power (2,2)**

→ is 2 = 0 ? No

2 * 2 = 4 → 2 * **power(2,1)**

→ is 1 = 0 No

2 * 1 = 2 → 2 * **power(2,0)**

→ is 0 = 0 ? **Yes**!

1 → **return 1**

Time Complexity of the Iterative and Recursive Power method

```
public static int power(int x, int n) {
  int pow = 1;
  for (int i = 1; i <= n; i++) // n times
     pow *= x;
  return pow;
}
```

There are n iterations of the loop in the iterative approach, so its time complexity is O(n).

```
public static int power(int x, int n) {
  if (n == 0)
    return 1;
  return x * power(x, n-1);
}
```

There are n recursive calls, and each call uses constant time operations. Thus, the time complexity is O(n).
This is the same as the time complexity using the iterative approach.

Recursion Problem

- If the recursion never reaches the base case, the recursive calls will continue to run forever (similar to the infinite loop) . Therefore, when programming recursively, you need to make sure that the algorithm is moving toward the base case to ensure that recursion will stop.

Example 3: The Fibonacci Sequence

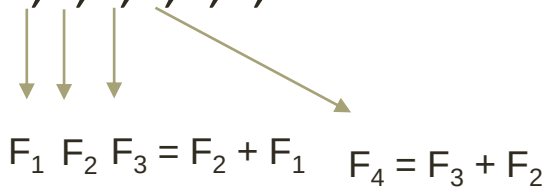
f_n = 1,1,2,3,5,8,13,21,34,55,…

Base steps:

Recursive step:

$$f_n = \begin{cases} \mathbf{1} \text{ if } n=1 \\ \mathbf{1} \text{ if } n=2 \\ \mathbf{f_{n-1} + f_{n-2}} \text{ if } n > 2 \end{cases}$$

f_n = 1,1,2,3,5,8,13



F_5= … (try to write the recursive call)

- $F_5 = F_4 + F_3 = \ldots$

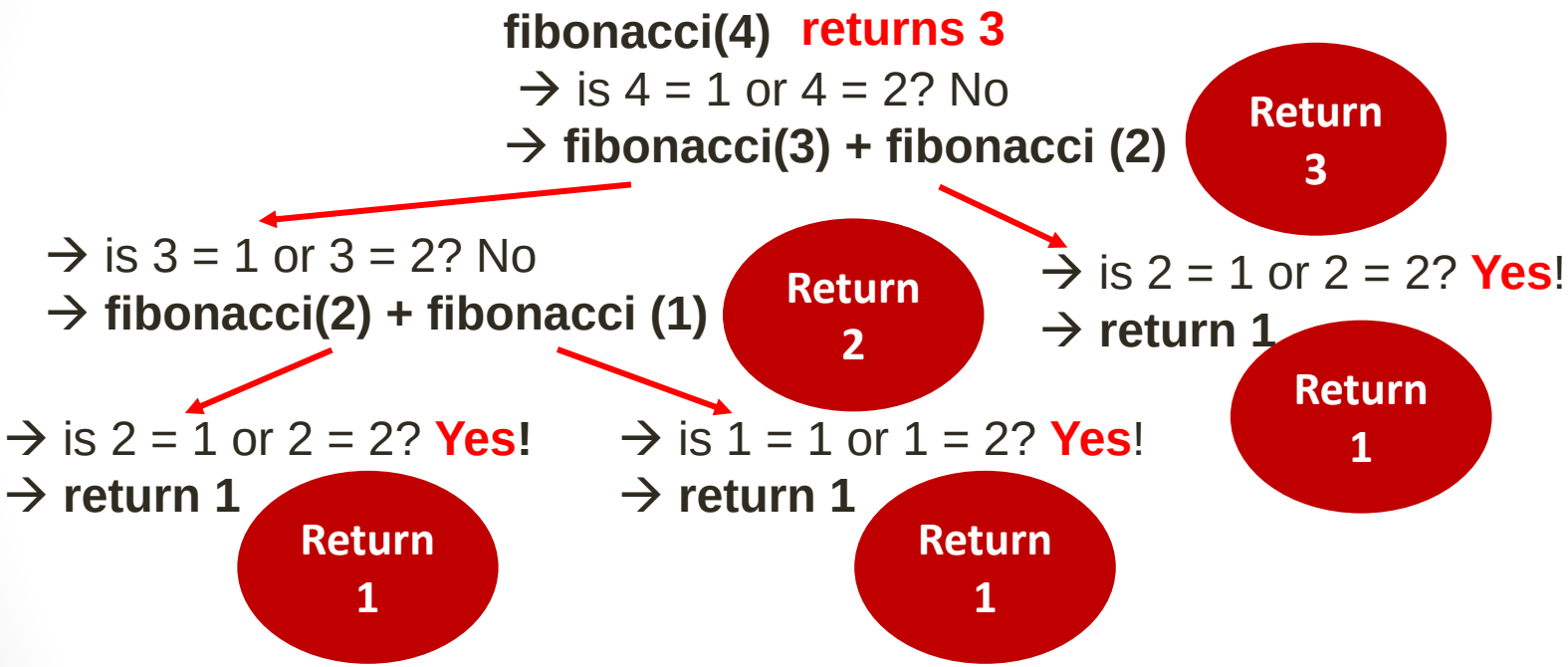
Non Recursive (Iterative) Fibonacci method

```
public static int fibonacci(int n) {
  if (n == 1 || n == 2)
      return 1;
  int fibo1 = 1, fibo2 = 1, fibo3 = 1;
  for(int i = 3; i <= n; i++) {
    fibo3 = fibo1 + fibo2; // Fibonacci number is sum of previous two Fibonacci numbers
    fibo1 = fibo2;
    fibo2 = fibo3;
  }
  return fibo3; // Fibonacci number
}
public static void main(String[] args){
  System.out.println(fibnoacci(4));
}
```

Recursive Fibonacci method

```
public static int fibonacci (int n) {
  if (n == 1 || n == 2) // base step
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);// recursive step
}
public static void main(String[] args){
  System.out.println(fibonacci(4));
}
```

Recursive Fibonacci method




```
public static int fibonacci (int n) {
  if (n == 1 || n == 2) // base step
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);// recursive step
}
```

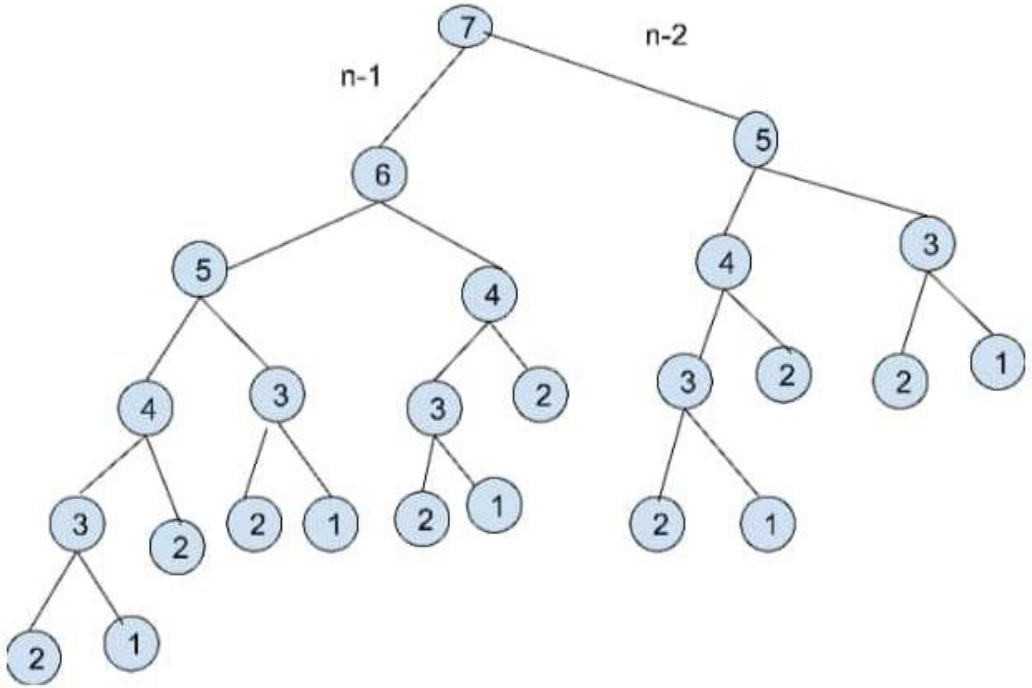
Time Complexity of the Iterative Fibonacci method

```
public static int fibonacci(int n) {
  if (n == 1 || n == 2)
      return 1;
  int fibo1 = 1, fibo2 = 1, fibo3 = 1;
  for(int i = 3; i <= n; i++) {
    fibo3 = fibo1 + fibo2;
    fibo1 = fibo2;
    fibo2 = fibo3;
  }
  return fibo3;
}
```

There are n-2 iterations of the loop in the iterative approach, so its time complexity is O(n).

Time Complexity of the Recursive Fibonacci method

The below image shows the recursive calls tree of Fibonacci with n = 7. Note how many times fibonacci(1), fibonacci(2), and fibonacci(3) are called. The higher the value of n the more duplicate recursive calls we will have. Calling fibonacci with the same value several times will increase the running time of the algorithm.



Time Complexity of the Recursive Fibonacci method

```
public static int fibonacci (int n) {
  if (n == 1 || n == 2)
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

The time complexity without showing the proof is $O(2^n)$.

The brief explanation is as follows. To find the time complexity of this recursive algorithm, we should consider how many times we will be calling fibonacci() by examining the calls tree in the previous slide. In the first level of the tree, we have one call: fib 7. There are two calls in the next level (fib 6 and fib 5) and four in the level after it. Each time our node branches off, we have two additional nodes, so it's $2*2*2... = 2^n$, making it $O(2^n)$.
This shows that in the case of Fibonacci example, the iterative approach is much more efficient that the recursive approach.

Iterative(loops) VS Recursive Algorithms

- Any recursive algorithm can be implemented iteratively, but sometimes only with great difficulty.
- **Recursive algorithms** are often **shorter** and **easier** to understand than iterative algorithms (Fibonacci as an example).
- **Iterative algorithms** are usually **more efficient** in their use of space and time of recursive algorithms. So, a recursive solution will always run more slowly than an iterative one because of the **overhead** of opening and closing the recursive calls.

Example 4: Sum

- If we call sum(4), it returns 10
  sum(4) = 1 + 2 + 3 + 4 = 10

- To look at it recursively

  sum(4) = 4 + sum(3) = 4 + 3 + sum(2) =

  4 + 3 + 2 + sum(1) = 4 + 3 + 2 + 1 + sum(0) =

  4 + 3 + 2 + 1 + 0 = 10

- If n = 0, the result is 0.
- Otherwise we have to repeat the sum with a smaller value of n as in **n + sum(n-1)**

Non Recursive (Iterative) Sum method

```
public statc int sum(int n) {
   int sum = 0;
   for (int i = 1; i <= n ; i++)
      sum += i;
   return sum;
}
public static void main(String[] args){
   System.out.println(sum(4));
}
```

Recursive Sum method

```
// Assuming n >= 0
public int sum(int n) {
   if (n == 0)
      return 0;
   return n + sum (n - 1);
}
public static void main(String[] args){
   System.out.println(sum(4));

}
```

Recursive Sum method

```
public int sum(int n) {
   if (n == 0)
      return 0;
   return n + sum (n - 1);
}
```

sum(4) **returns 10**

→ is 4 = 0 ? No

4 + 6 = 10 → 4 + **sum (3)**

→ is 3 = 0 ? No

3 + 3 = 6 → 3 + **sum(2)**

→ is 2 = 0 No

2 + 1 = 3 → 2 + **sum(1)**

→ is 1 = 0 No

1 + 0 = 1 → 1 + **sum(0)**

→ is 0 = 0 ? **Yes**!

→ **return 0**

0

Time Complexity of the Iterative and Recursive Sum method

```
public statc int sum(int n) {
  int sum = 0;
  for (int i = 1; i <= n ; i++)
     sum += i;
  return sum;
}
```

There are n iterations of the loop in the iterative approach, so its time complexity is O(n).

```
public int sum(int n) {
  if (n == 0)
     return 0;
  return n + sum (n - 1);
}
```

There are n recursive calls, and each call uses constant time operations. Thus, the time complexity is O(n).

This is the same as the time complexity using the iterative approach.