

# Stacks

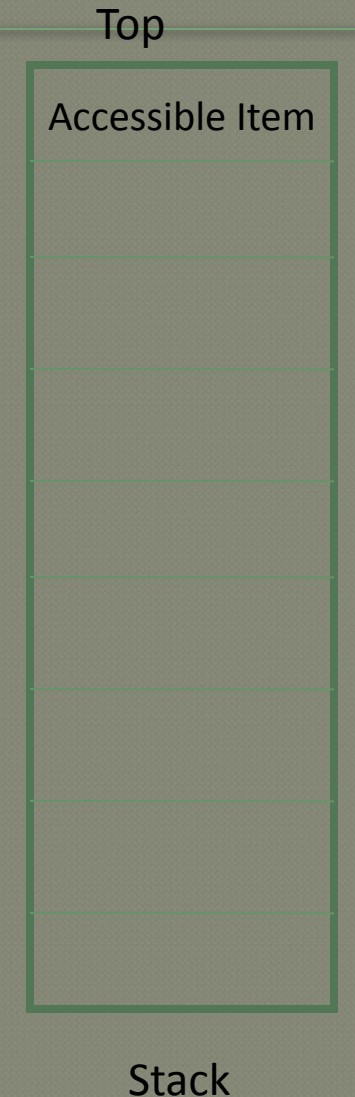
---

- Is a data structure used to organize collection of items of similar type.
- Items are added and deleted at only one end called Top.
- Most recently added item should be deleted first.
- It is Last-In-First-Out(LIFO) data structure.
- Each stack is associated with items and Top.



# Restricted Access

- Unlike arrays, stacks only allow the top most item to be accessed at any time
- The interface of a stack is designed to enforce this restriction
- This feature is useful for many day to day and programming operations





# An Abstract Data Type

---

- Stacks are defined primarily by
  - Interface
    - Only first item should be accessible
    - Last item should be popped out first
  - Operations performed on the interface
    - Push
    - Pop
    - Peek(Peep)
- Underlying mechanism of its implementation is not visible to the user.
  - Arrays, Structures or any other data types can be used for this purpose



# Uses

---

- Stack can be used in
  - Balancing brackets, paranthesis and braces in a computer source program
  - Parsing an arithmetic expression like  $3*(4+5)$
  - Traversing Binary trees (don't freak out we will discuss it later)
  - Microprocessors to keep the track of the functions call, especially recursive calls



# Real world examples

---

- Postal service and office mail
  - throw all the received letters in the inbox or in-basket
  - Process each letter starting from the top most item and working their way down
- Task performed in a day to day life
  - You are working on project A
  - While working, a coworker asked for the help on project B
  - While working on B, your boss called you for his help with travel expenses
  - After completing the task at hand you have to return to the previous task until you reach project A



# Operations on Stack

---

- Stack Size
  - Returns the number of items on the stack
- Push
  - Pushes an item on the top of the stack and increments the size and top pointer of the stack
- Pop
  - Pulls out the top most item from the stack and decrements the stack size and the top pointer of the stack
- Peek(Peep)
  - Without removing the item from the stack returns the top most item of the stack. Does not effect top of the stack pointer and the stack size



# Implementations

---

Discussed in class( 4 ways of implementations: Refer to class notes)

- Array representation with static allocation
- Array representation with dynamic allocation
- Array representation with static allocation using structure
- Array representation with dynamic allocation using structure



# Applications

---

- Simple applications like checking for paranthesized expression and palindrome(Discussed in class)
- Conversion and evaluation of expressions
- Recursion

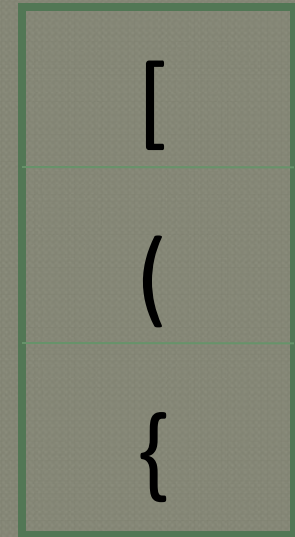



# Working Example of Delimiter Matching

- Consider the string

a{b(c[d]e)f}

a { b ( c [ d ] e ) f }



Stack



# Matching Delimiters, An Example (Pseudo Code)

---

- The program works as follows:
  1. Reads one item from the given string
  2. If character is an opening delimiter, It will be pushed on the stack
  3. Else if it's a closing delimiter then it checks
    - a. If there is no item in the stack to match with then exit with an error message
    - b. Else the top item of the stack will be popped out and compared with the read delimiter, if they match then go to step 1, else exit with the error message.
  4. After reading the whole string, if there is any delimiter left in the stack then exit with a an error message



# Converting Infix to Postfix

---

- An Infix notation:
  - $2*3-4/5$
- Equivalent postfix notation would be
  - $23*45/-$
- In a postfix notation operators come after the operands
- Stacks can be very effectively used to convert an infix notation to a postfix notation



# Conversion Algorithm

---

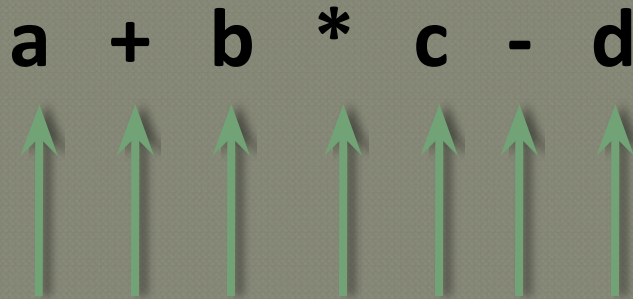
- Scan the Infix string from left to right.
- Initialize empty stack.
- If the scanned character is an operand, add it to the Postfix string.  
If the scanned character is an operator and if the stack is empty  
Push the character to stack.
  - If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack).
  - If topStack has higher precedence over the scanned character Pop the stack else Push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.
  - Repeat this step till all the characters are scanned.
- If stack is not empty add topStack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty.
- Return the Postfix string.



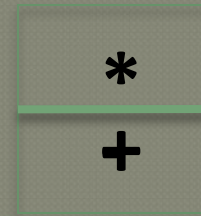
# A Working Example

- Infix String:  $a+b*c-d$

**a + b \* c - d**



The diagram shows the infix string "a + b \* c - d". Below each character, there is a green arrow pointing upwards. These arrows point towards a stack structure on the right, indicating the process of pushing operators onto the stack as they are encountered from left to right.



PostfixString

**a b c \* + d -**



# Stacks – Relevance

- Stacks appear in computer programs
  - Key to call / return in functions & procedures
  - Stack frame allows recursive calls
  - Call: **push** stack frame
  - Return: **pop** stack frame
- Stack frame
  - Function arguments
  - Return address
  - **Local variables**



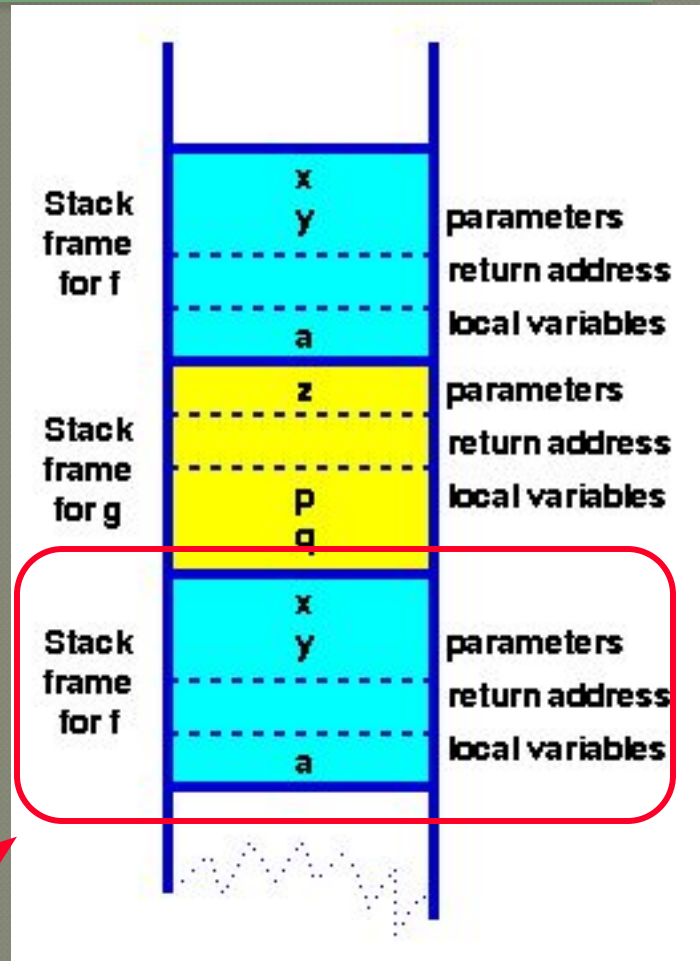
# Stack Frames - Functions in HLL

## Program

```
function f( int x, int y) {  
    int a;  
    if ( term_cond ) return ...;  
    a = ...;  
    return g( a );  
}
```

```
function g( int z ) {  
    int p, q;  
    p = ... ; q = ... ;  
    return f(p,q);  
}
```

**Context  
for execution of f**





# Recursion



# Recursive Thinking

---

- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that would be difficult to solve in other ways
- Recursion splits a problem into one or more simpler versions of itself



# Recursion

- Typically problems are solved by dividing into sub-problems
- These sub-problems may also be divided into further smaller sub-sub-problems
- When the **sub-problems are small enough to solve directly** the process stops.
- A ***recursive algorithm*** follows the same technique by solving a given problem through solution of **two or more easier to solve sub-problems**
- A **recursive call** is a function call in which the called function is the same as the one making the call.



# Recursive Definitions

---

## ● Recursion

- Process of solving a problem by reducing it to smaller versions of itself

## ● Recursive definition

- Definition in which a problem is expressed in terms of a smaller version of itself
- Has one or more base cases



# Recursive Definitions

## ● **Recursive algorithm**

---

- Algorithm that finds the solution to a given problem by reducing the problem to smaller versions of itself
- Has one or more base cases
- Implemented using recursive functions

## ● **Recursive function**

- function that calls itself



# Recursive Definitions

---

- **Base case**

- Case in recursive definition in which the solution is obtained directly
- Stops the recursion

- **General case**

- Case in recursive definition in which a smaller version of itself is called
- Must eventually be reduced to a base case



# Recursion

---

- Define an object in terms of itself
- Consists of two parts
  - Recursive step
  - Base step

$$f(n) = f(n-1) * n;$$
$$f(1) = 1;$$



# Outline of a Recursive Function

---

if (answer is known)  
    provide the answer

base  
case

else

    make a recursive call to  
    solve a **smaller** version  
    of the same problem

recursive  
case -



# Tracing a Recursive Function

---

## **Recursive function**

- Has unlimited copies of itself
- Every recursive call has
  - its own code
  - own set of parameters
  - own set of local variables



# Tracing a Recursive Function

## After completing recursive call

---

- Control goes back to calling environment
- Recursive call must execute completely before control goes back to previous call
- Execution in previous call begins from point immediately following recursive call



# How Recursion Works

---

- a recursive function call is handled like any other function call
- each recursive call has an activation record on the stack
  - stores values of parameters and local variables
- when base case is reached return is made to previous call
  - the recursion “unwinds”



# Finding a Recursive Solution

- A recursive solution to a problem must be written carefully.
- The idea is for each successive recursive call to bring you one step closer to a situation in which the problem can easily be solved.
- This easily solved situation is called the **base case**.
- Each recursive algorithm must have at least one base case, as well as the general case.



# Designing Recursive Functions

---

- Understand problem requirements
- Determine limiting conditions
- Identify base cases



# Designing Recursive Functions

---

- Provide direct solution to each base case
- Identify general case(s)
- Provide solutions to general cases in terms of smaller versions of itself



# Steps to Design a Recursive Algorithm

---

- There must be at least one case (**the base case**), for a small value of  $n$ , that can be solved directly
- A problem of a given size  $n$  can be split into one or more smaller versions of the same problem (**recursive case**)
- Recognize the base case and provide a solution to it
- Devise a strategy to split the problem into smaller versions of itself while making progress toward the base case
- Combine the solutions of the smaller problems in such a way as to solve the larger problem



# Factorial

---

$$4! = 4 * 3 * 2 * 1$$

$$n! = n * (n-1) * \dots * 1$$

$$4! = 4 * 3!$$

$$n! = n * (n-1)!$$

$\text{factorial}(n) = n * \text{factorial}(n-1)$  // recursive  
step

$\text{factorial}(0) = 1$  // base step



# Recursive Factorial Function

---

```
int fact(int num)
{
    if(num == 0)
        return 1;
    else
        return num * fact(num - 1);
}
```



# Execution Model for Factorial

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1)! & \text{if } n > 0 \end{cases}$$

n 2

```
if( n == 0)
    return 1;
} else {
    return n * Fact(n - 1);
}
```

n 1

```
if( n == 0)
    return 1;
} else {
    return n * Fact(n - 1);
}
```

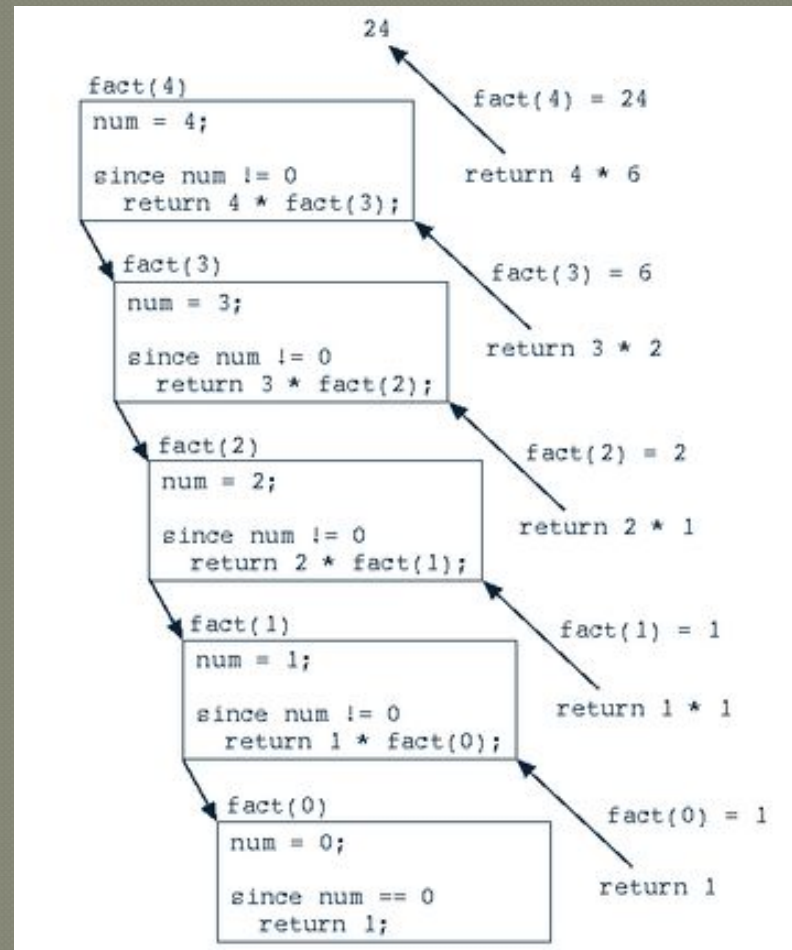
n 0

```
if( n == 0)
    return 1;
} else {
    return n * Fact(n - 1);
}
```

**Fact(2)**



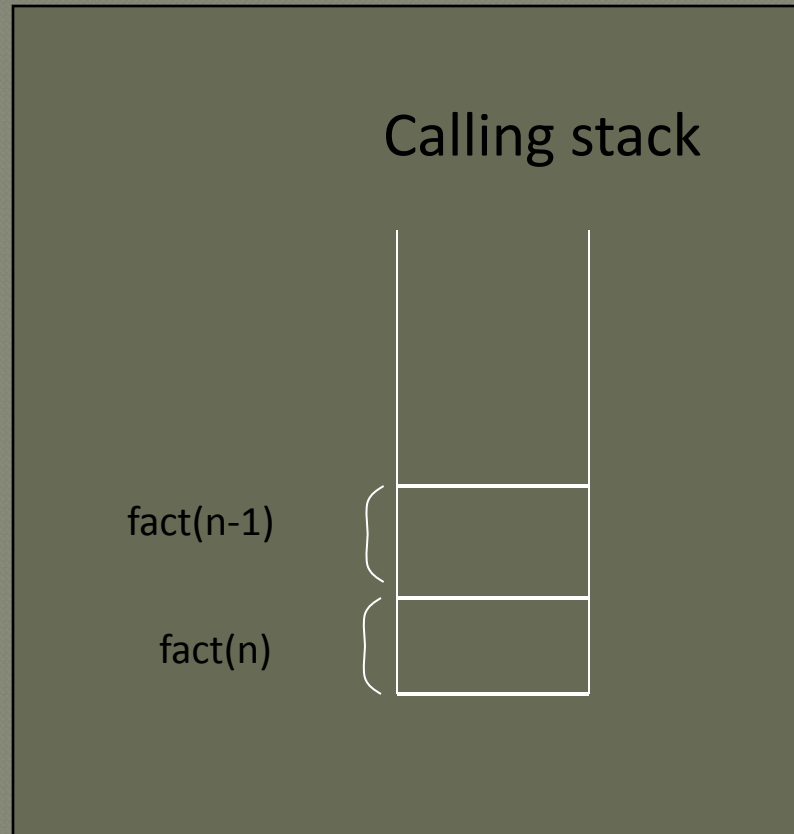
# Recursive Factorial Trace





# Calling Stack Of Recursive Call

```
int fact(int n){  
    if(n==1)  
        return 1;  
    else  
        return (fact(n-1)*n);  
}
```

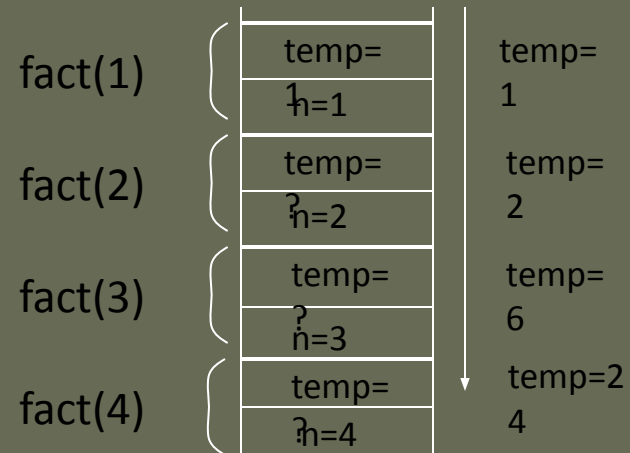




# Calling Stack

```
int fact(int n){  
    //base step  
    if (n == 1)  
        return 1;  
    //recursive step  
    int temp = n * fact(n-1);  
    return temp;  
}  
fact(4) = 4! = 24
```

## Calling stack





# Another Recursive Definition

---

Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, ....

$$\text{fib}(n) = \begin{cases} 0 & \text{for } n == 1 \\ 1 & \text{for } n == 2 \\ \text{fib}(n-2) + \text{fib}(n-1) & \text{for } n > 2 \end{cases}$$



# Fibonacci Numbers

---

0, 1, 1, 2, 3, 5, 8, 13, ...

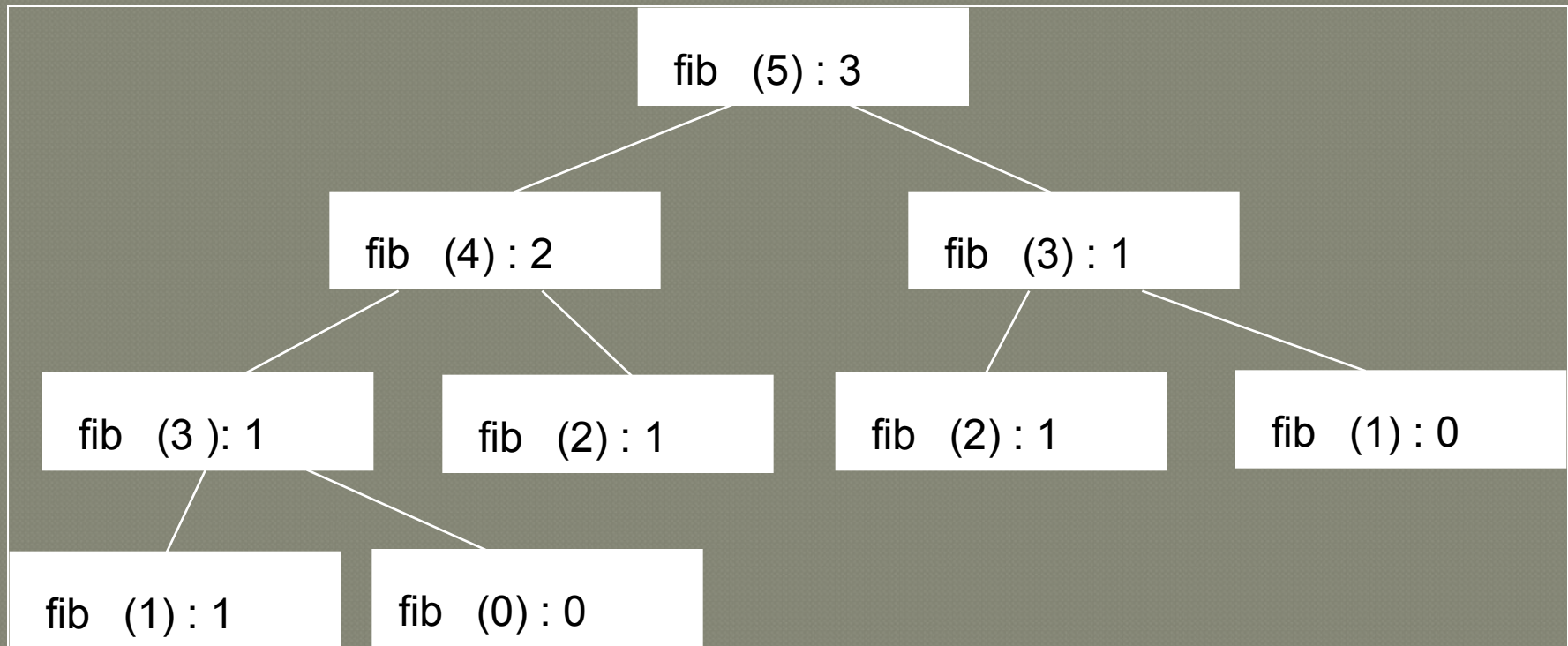
$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$  **Recursive case**

$\left. \begin{array}{l} \text{Fib}(1) = 0 \\ \text{Fib}(2) = 1 \end{array} \right\}$  **Base case**



# Recursive Fibonacci repeats computations

---





# Evaluating Exponents Recursively

---

```
int power(int k, int n) {  
    // raise k to the power n  
    if (n == 0)  
        return 1;  
    else  
        return k * power(k, n - 1);  
}
```



# Recursion or Iteration?

---

- Two ways to solve particular problem
  - Iteration
  - Recursion
- Iterative control structures: uses looping to repeat a set of statements
- Tradeoffs between two options
  - Sometimes recursive solution is easier
  - Recursive solution is often slower



# Recursion Versus Iteration

- In iteration, a loop repetition condition determines whether to repeat the loop body or exit from the loop
- In recursion, the condition usually tests for a base case
- You can always write an iterative solution to a problem that is solvable by recursion
- Recursive code may be simpler than an iterative algorithm and thus easier to write, read, and debug



# When to use recursion?

---

- if recursive and iterative algorithms are of similar efficiency --> **iteration**
- if the problem is inherently recursive and a recursive algorithm is less complex to write than an iterative algorithm --> **recursion**
- recursion very inefficient when values are recomputed



# Efficiency of Recursion

---

- Recursive methods often have slower execution times when compared to their iterative counterparts
- The overhead for loop repetition is smaller than the overhead for a method call and return
- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
  - The reduction in efficiency does not outweigh the advantage of readable code that is easy to debug



# Pitfalls of Recursion

- One pitfall of recursion is infinite regress, i.e. a chain of recursive calls that never stops. E.g. if you forget the base case. **Make sure you have enough base cases.**
- Recursion can be less efficient than an iterative implementation. This can happen because there is recalculation of intermediate results.
- There can be extra memory use because recursive calls must be stored on the execution stack.