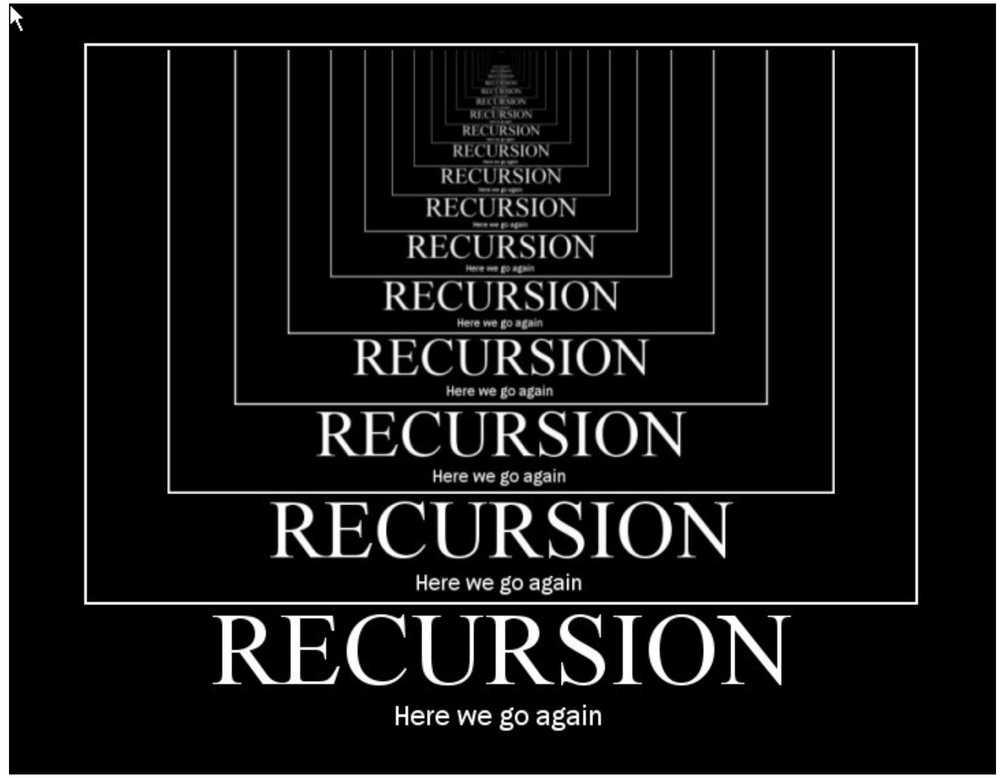
University of Science, VNU-HCM
Faculty of Information Technology

Data Structure and Algorithm

Recursion

Lecturer: Le Ngoc Thanh
Email: lnthanh@fit.hcmus.edu.vn

HCM City



Recursion

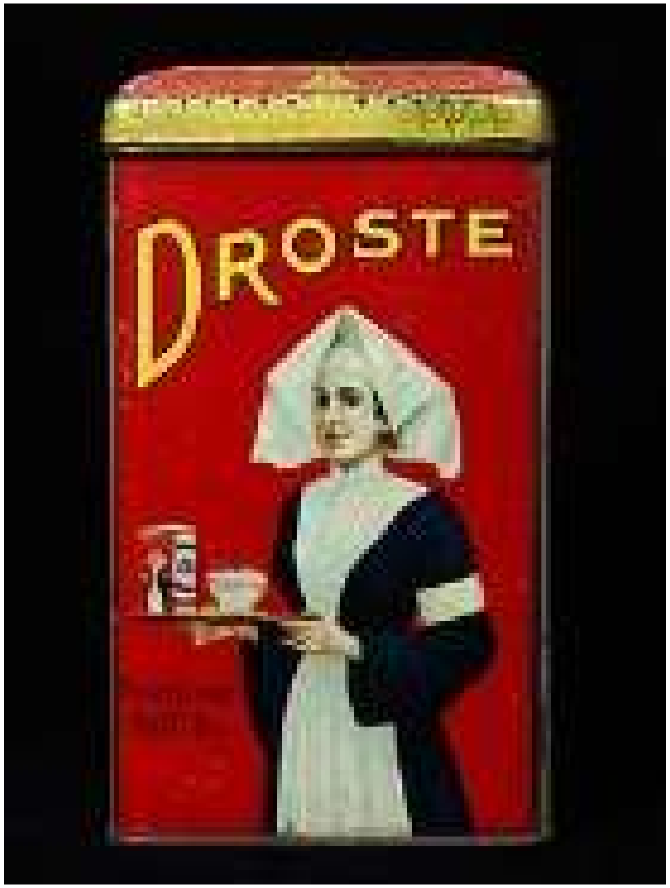
INTRODUCTION

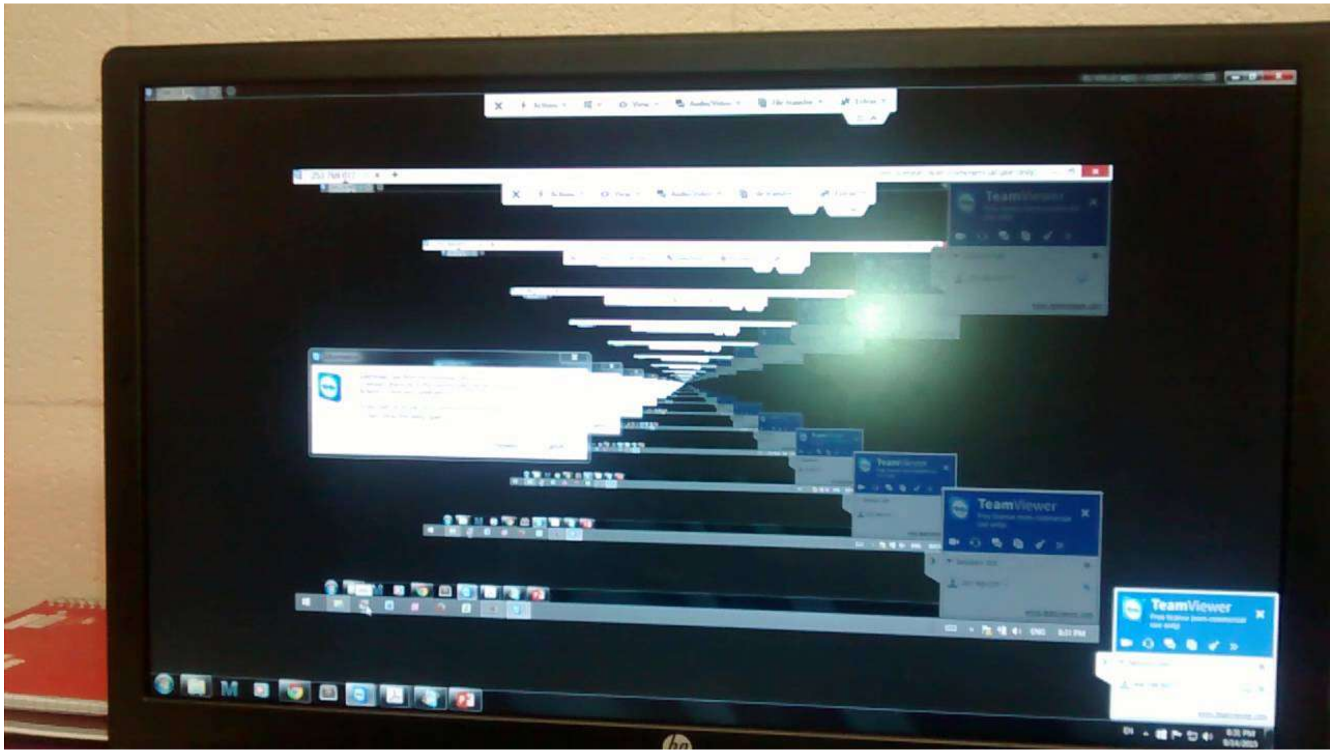
Recursion

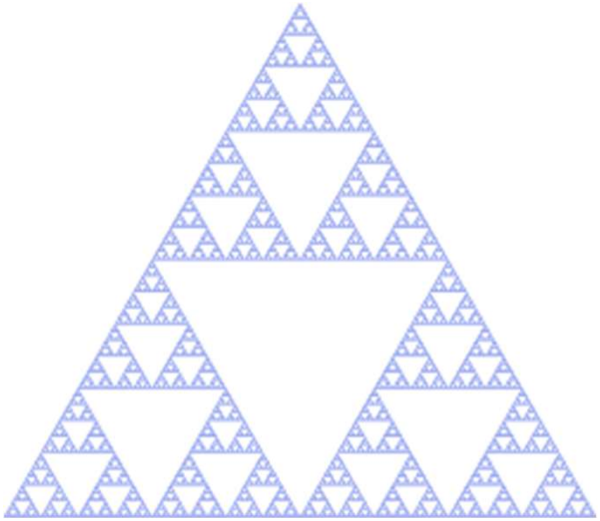
- **Recursion** occurs when a thing is defined in terms of itself or of its type.

Recursion in Natural

Recursion by human





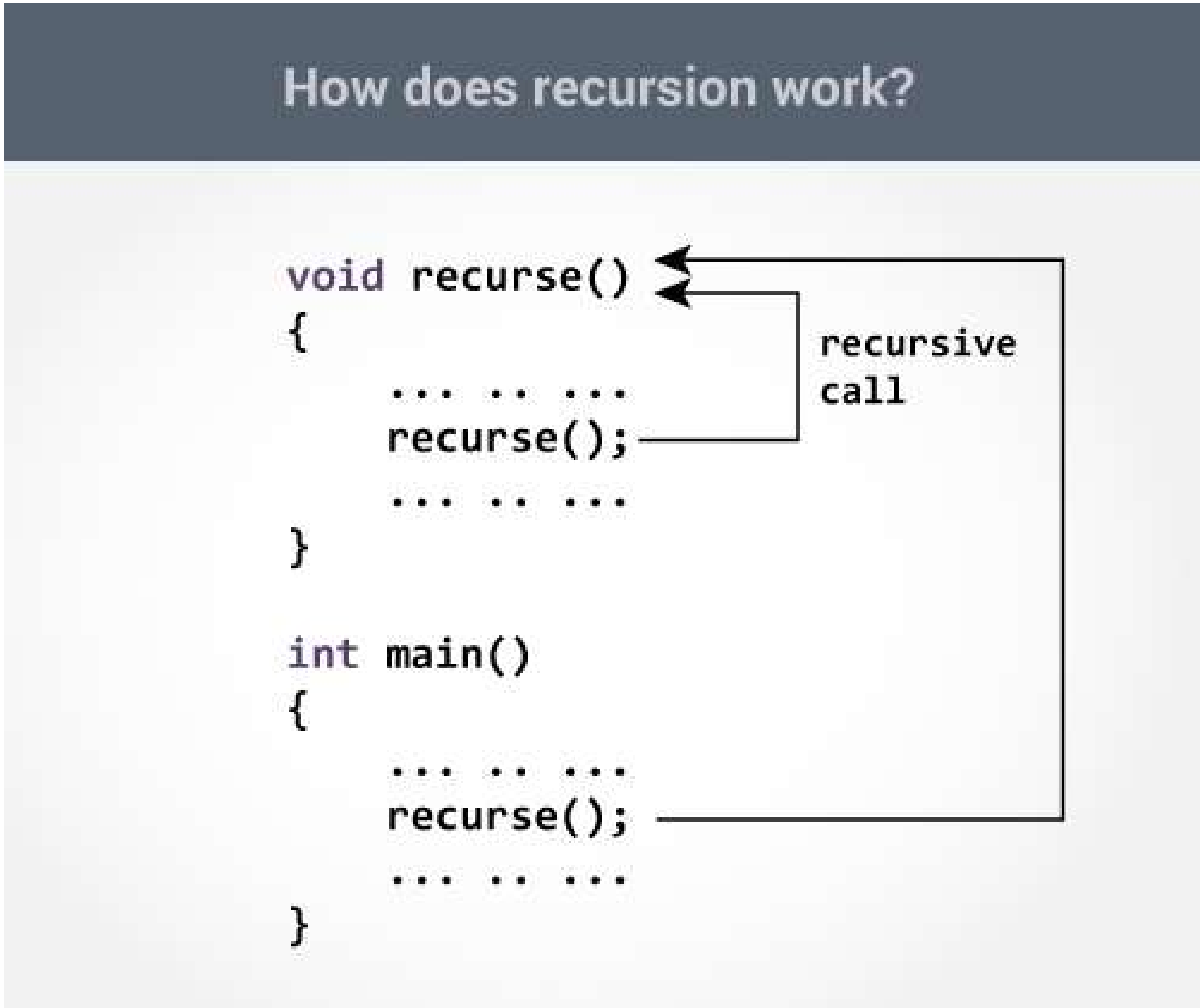


Recursion

RECURSION IN PROGRAMMING

Recursion in programming

- **Recursion** is when a function calls itself.



Recursion

- Recursion is useful for problems that can be represented by a simpler version of the same problem.

```
def fac(numb):
    if numb <= 1:
        return 1
    else:
        return numb * fac(numb - 1)
```

Example

- The factorial function

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

3! ?

Example

$$3! = (3 - 1)! \times 3$$
$$= 2! \times 3$$

We turn this problem into a smaller problem of same kind.
This is called "decomposition."

Example

$$
\begin{aligned}
3! &= (3 - 1)! \times 3 \\
&= \underline{2!} \times 3
\end{aligned}
$$

$$
\begin{aligned}
2! &= (2 - 1)! \times 2 \\
&= 1! \times 2
\end{aligned}
$$

Recursion Attributes

- Looping without a loop statement.
- A function that is part of its own definition.
- Can only work if there's a terminating condition, otherwise it goes forever (the base case).

Pros and Cons of Recursion

- Recursion makes program elegant and cleaner.
  - All algorithms can be defined recursively which makes it easier to visualize and prove.
- If the speed of the program is vital then, you should avoid using recursion.
  - Recursions use more memory and are generally slow. Instead, you can use loop.

Recursive vs Iterative

- For certain problems, a recursive solution often leads to short and elegant code.

Recursive solution

```
int fac(int numb){
  if(numb<=1)
    return 1;
  else
    return numb*fac(numb-1);
}
```

Iterative solution

```
int fac(int numb){
    int product=1;
    while(numb>1){
      product *= numb;
      numb--;
    }
    return product;
}
```

When to choose or not choose

- When to choose recursion against iteration
  - When the problem is complex and can be expressed in more simplified form as recursive case then its iterative counter part.
  - When the solution of the problem is inherently recursive. Like Structural recursion (Tree traversal) and Quick Sort.
- When to choose iterative solution against recursive solution
  - When the problem is simple.
  - When the solution of the problem is not inherently recursive. Main problem can not be expressed easily into sub problem of same type.
  - Another possible reason for choosing an iterative rather than a recursive algorithm is that in today's programming languages, the stack space available to a thread is often much less than the space available in the heap, and recursive algorithms tend to require more stack space than iterative algorithms.

Terminal (Base case)

- If we use recursion, we must be careful not to create an infinite chain of function calls:

```
int fac(int numb){
    return numb * fac(numb-1);
}
```

Oops!

No termination condition

```
int fac(int numb){
    if (numb<=1)
        return 1;
    else
        return numb * fac(numb+1);
}
```

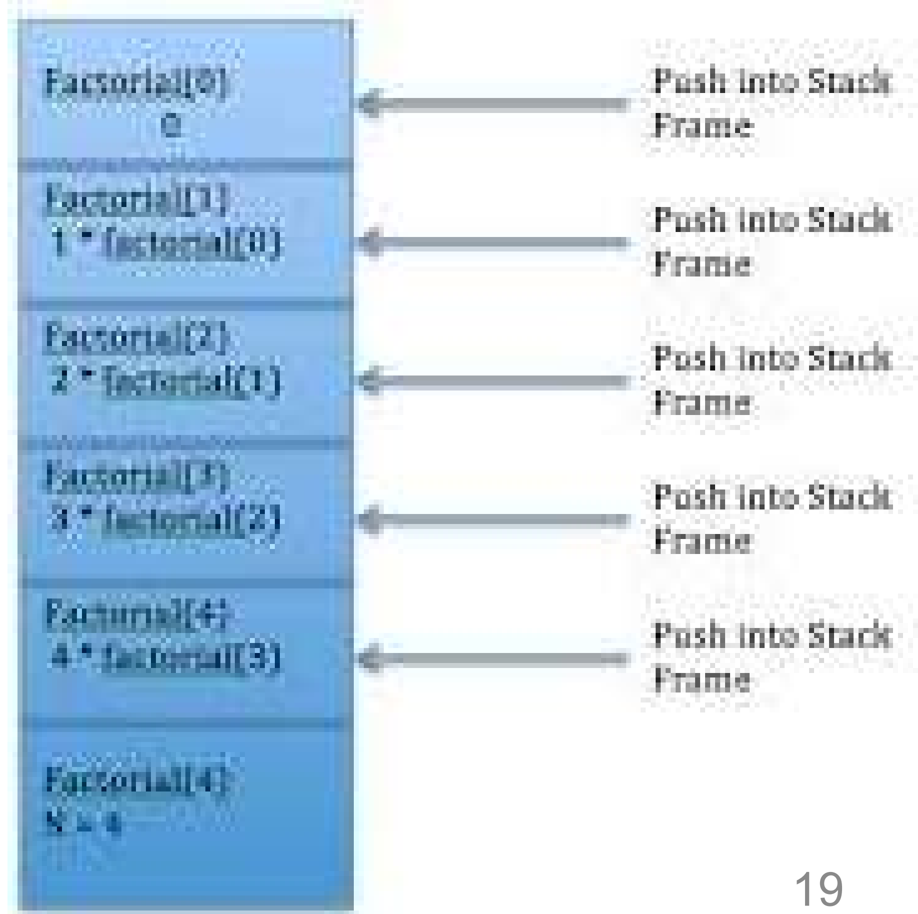
Oops!

Terminal (Base case)

- We must always make sure that the recursion *bottoms out*:
  - A recursive function must contain at least one non-recursive branch.
  - The recursive calls must eventually lead to a non-recursive branch.

How recursion is handled

- Every time a function is called, the function values, local variables, parameters and return addresses are pushed onto the stack.
- Over and Over again
- You might run out!



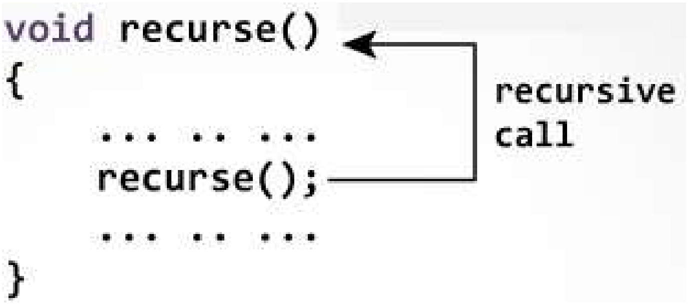
stackoverflow

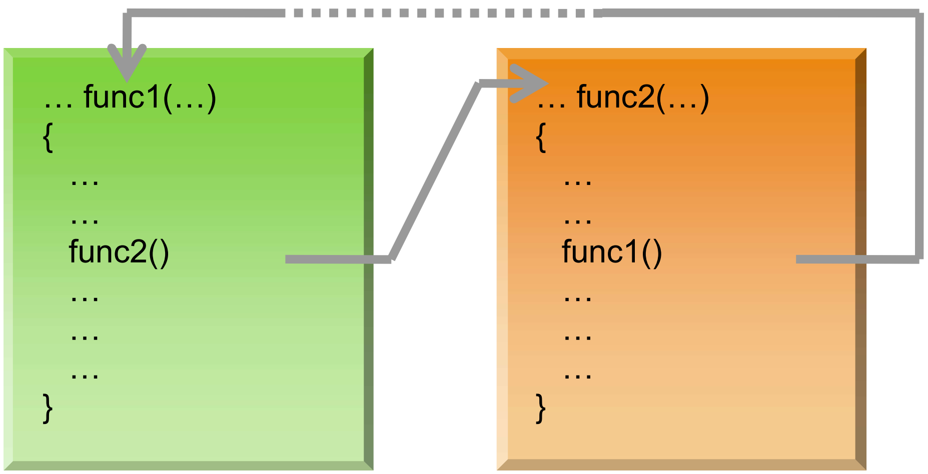
Recursion

TYPES OF RECURSION

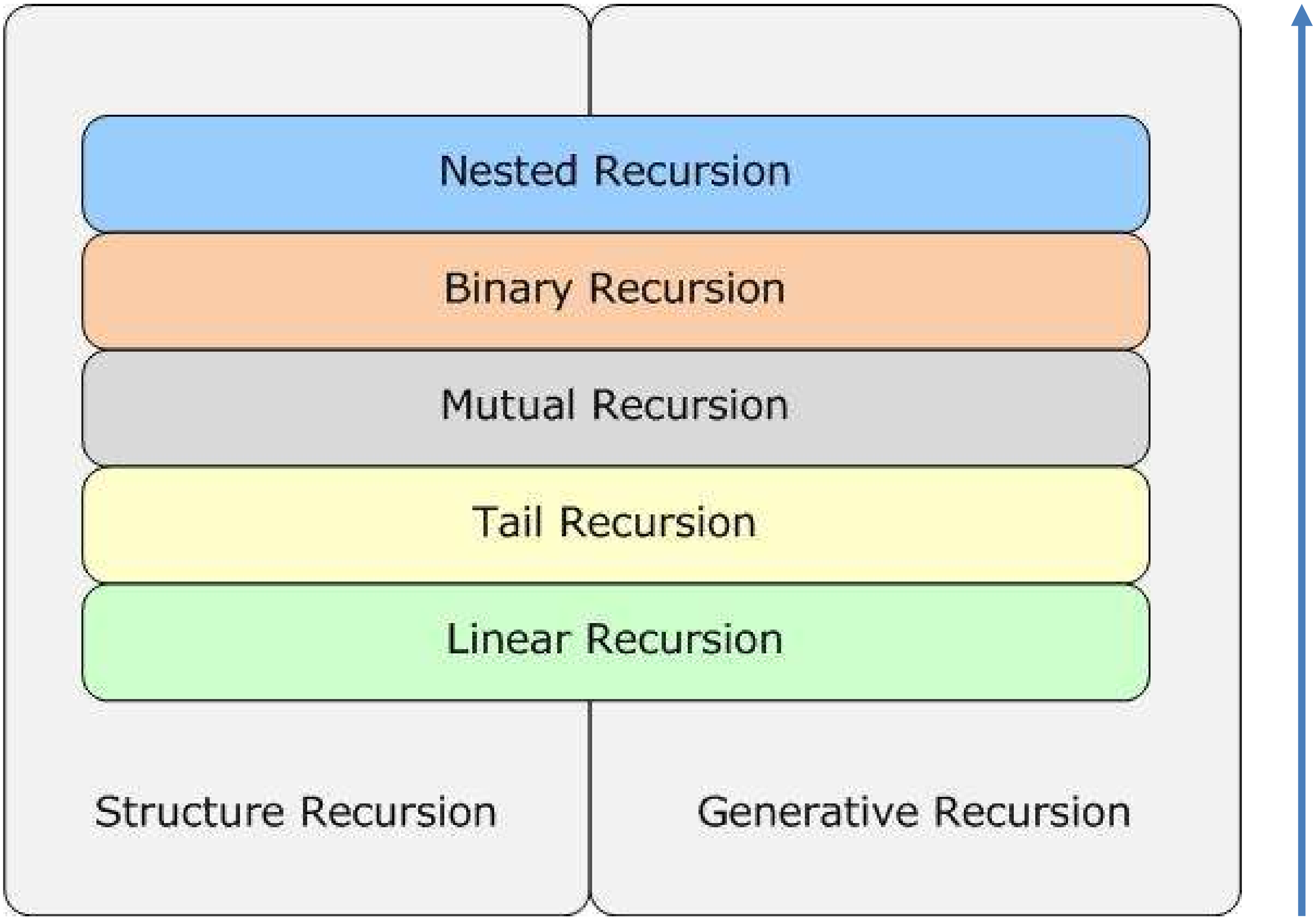
Simple Types of Recursion

- Direct recursion
  - a function calls itself
- Indirect recursion
  - function A calls function B, and function B calls function A.
  - function A calls function B, which calls …, which calls function A





Full Types of Recursion



Linear Recursion

- Linear Recursion is a type of recursion where each function call makes only one recursive call.
  - This means that at each step, the function calls itself just once, without branching into multiple recursive calls.

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Tail Recursion

- Tail recursion is a specialized form of linear recursion where the recursive function called is usually the last call of the function.
  - No further computation is required after the recursive call returns.
  - The smart compiler can automatically convert this recursion into loop to avoid nested function calls.

```
sub foo (int a)
{
   if(a == 1)
     return 1;
   else
     return foo(a-1);
}
```

Tail Recursive Call

Tail of the function

Example

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        # Not tail-recursive (multiplication happens after recursion)
        return n * factorial(n - 1)


def tail_factorial(n, accumulator=1):
    if n <= 1:
        return accumulator  # The final result is returned directly
    else:
        # Recursive call is the last operation
        return tail_factorial(n - 1, n * accumulator)
```

Mutual Recursion

- Mutual recursion is also known as indirect recursion.
  - Two or more than two functions call each other recursively, creating a cycle of function calls.

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

Example

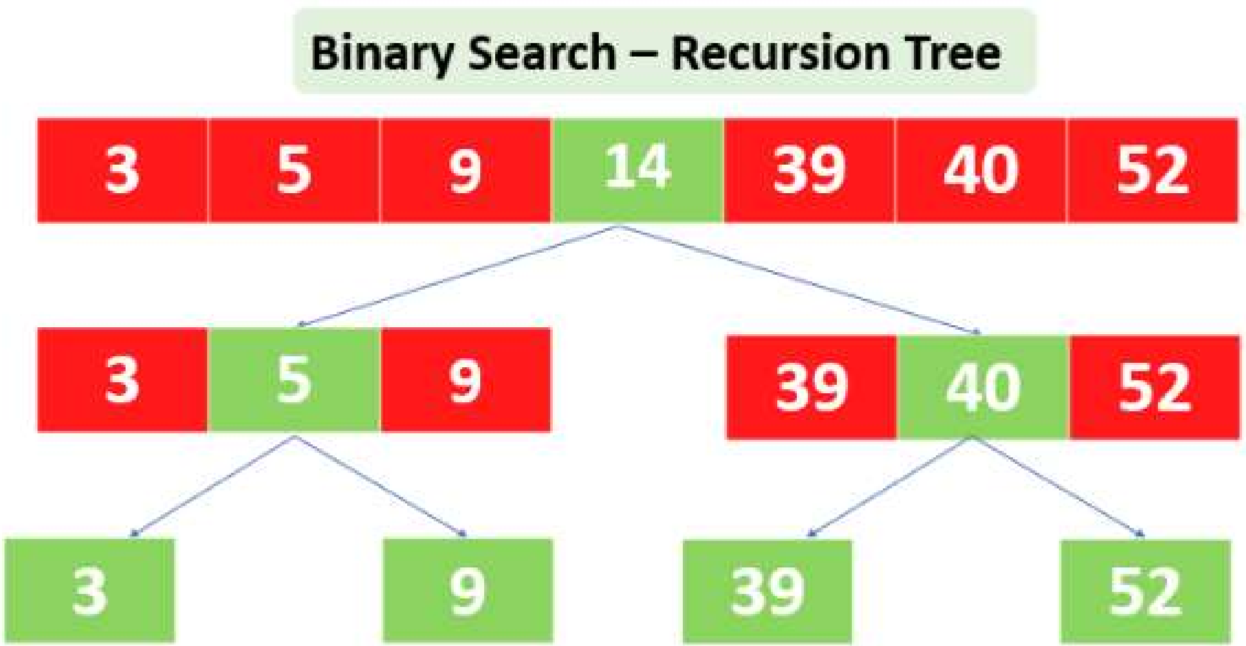
- Let's define two mutually recursive functions:
  - is_even(n): Checks if a number is even.
  - is_odd(n): Checks if a number is odd.

```
def is_even(n):
    if n == 0:
        return True
    else:
        return is_odd(n - 1)

def is_odd(n):
    if n == 0:
        return False
    else:
        return is_even(n - 1)
```

Binary Recursion

- Binary Recursion is a type of recursion where a function calls itself twice during each recursive step.
  - At each level of recursion, the problem is split into two smaller subproblems, and both are solved recursively.



Example

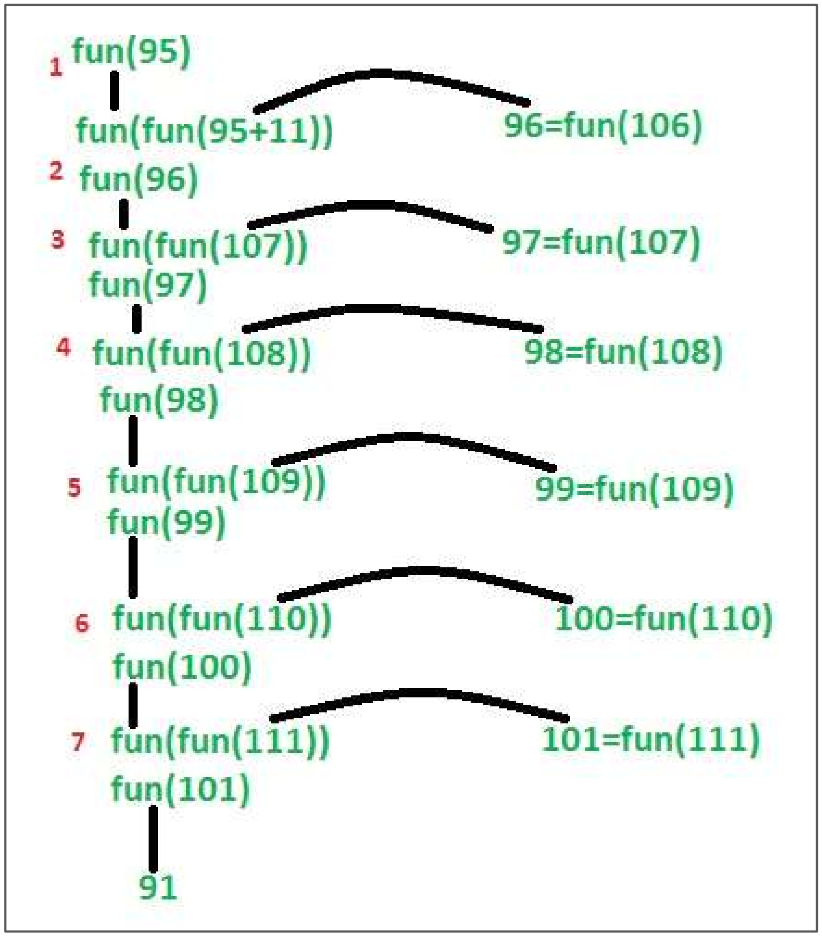
- Write a program to calculate Fibonacci.

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

```python
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        # Two recursive calls
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Nested Recursion

- Nested Recursion is a type of recursion where the argument of a recursive function is itself a recursive call.
  - Instead of passing a simple value (like n-1 or n/2) as an argument, we pass the result of another recursive function call.



Example

- The Ackermann function is defined as:

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

where m and n are non-negative integers

```python
def ackermann(m, n):
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        # Nested Recursion
        return ackermann(m - 1, ackermann(m, n - 1))
```

Recursion

EXAMPLES OF RECURSION

Example 1: Count Down

countDown(5)

print "happy recursion day"

```
5
4
3
2
1
happy recursion day
```

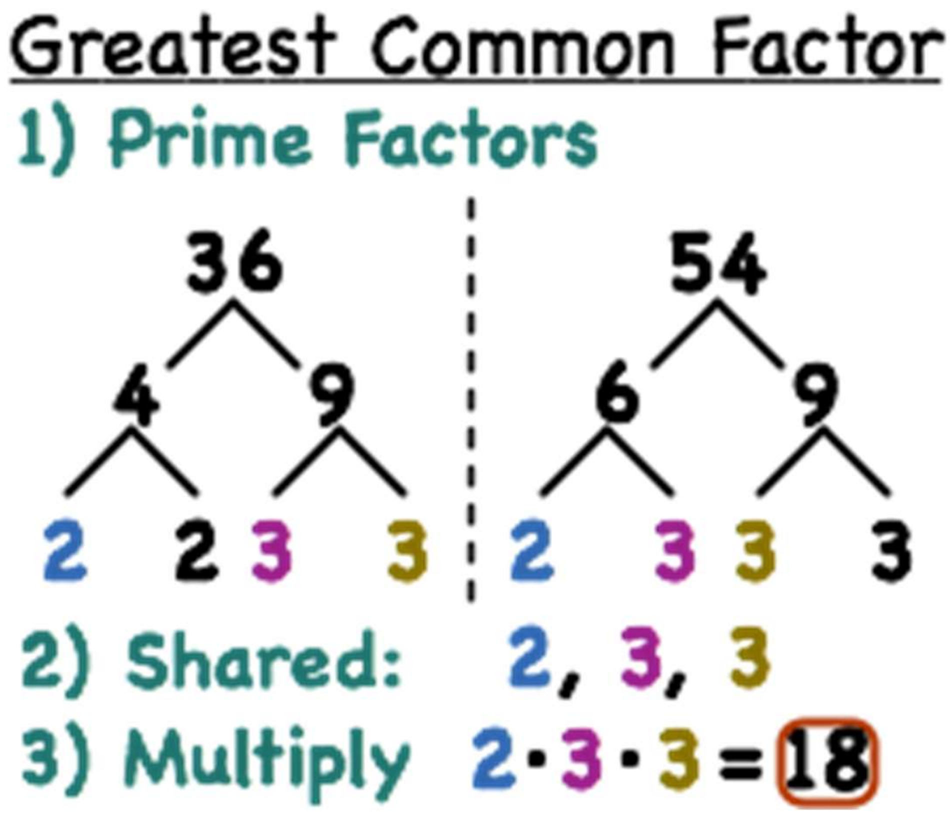
Count Down

```
def countDown(n):
    if n < 0:
        return
    print(n)
    countDown(n - 1)

countDown(5)
print("happy recursion day")
```

Example 2: Greatest common divisor

- The Greatest Common Divisor (GCD) is the largest factor common to both



GCD Algorithm 1

- It is based on the property:

$$GCD(a, b) = GCD(a - b, b) \;\; if \; a > b$$
$$GCD(a, b) = GCD(a, b - a) \;\; if \; b > a$$

With the base case:

$$GCD(a, 0) = |a|$$
$$GCD(0, b) = |b|$$

GCD – Alg 1

$$
\begin{aligned}
\mathbf{GCD(68, 119)} &= \mathrm{GCD}(68, 51) \\
&= \mathrm{GCD}(17, 51) \\
&= \mathrm{GCD}(17, 34) \\
&= \mathrm{GCD}(17, 17) \\
&= \mathbf{17}
\end{aligned}
$$

GCD – Alg1

```
def gcd(a, b):
    if a == 0:
        return abs(b)
    if b == 0:
        return abs(a)
    if a > b:
        return gcd(a - b, b)
    else:
        return gcd(a, b - a)


print(gcd(48, 18))   # Output: 6
print(gcd(56, 98))   # Output: 14
```

GCD Algorithm 2

- The Euclidean Algorithm is an efficient method to compute the GCD.
- It is based on the property:

$$GCD(a, b) = GCD(b, a \; mod \; b)$$

With the base case:

$$GCD(a, 0) = |a|$$

GCD Algorithm 2

```python
def gcd(a, b):
    if b == 0:
        return abs(a)
    else:
        return gcd(b, a % b)

# Example usage
print(gcd(48, 18))  # Output: 6
print(gcd(56, 98))  # Output: 14
```

Reversed string

- Given a string, I want a reversed version of that string.

  reverse("reenigne")

  => “engineer”

- The reverse of an empty string is an empty string.

Reversed string

```
def reverse_string(s):
    if len(s) == 0:
        return s
    return s[-1] + reverse_string(s[:-1])


print(reverse_string("hello"))  # Output: "olleh"
print(reverse_string("recursion"))  # Output: "noisrucer"
```

Sum of digits

sumDigits(314159265)

=> 36

Sum of Digits Algorithm

- Sum of digits of 0 is 0.
- Sum of digits of N > 0:
  - Find last digit + sum of digits except last.

N % 10

N // 10

```python
def sum_of_digits(n):
    if n == 0:
        return 0
    return (n % 10) + sum_of_digits(n // 10)

# Example usage
print(sum_of_digits(1234))  # Output: 10 (1+2+3+4)
print(sum_of_digits(9876))  # Output: 30 (9+8+7+6)
```

Palindrome

isPalindrome("Refer")

=> True

isPalindrome("Referrer")

=> False

Civic
Level
Madam
Malayalam
Radar
Reviver
Rotator
Terret

Palindrome Algorithm

- Base Case:
  - If the string has 0 or 1 character, it is a palindrome (return True).
- Recursive Case:
  - Check if the first and last characters are the same.
  - If they match, recursively check the substring excluding those two characters.

Palindrome Algorithm

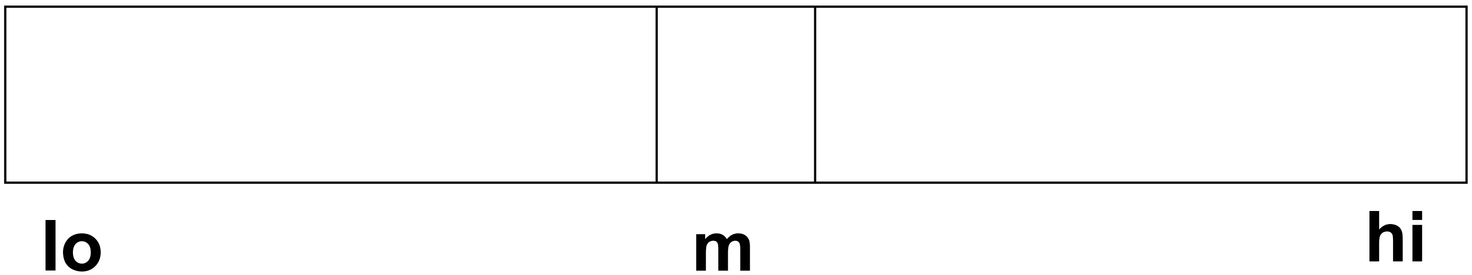
```
def is_palindrome(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return is_palindrome(s[1:-1])

# Example usage
print(is_palindrome("madam"))   # Output: True
print(is_palindrome("racecar")) # Output: True
print(is_palindrome("hello"))   # Output: False
```

Binary Search

- Assume an array `a` that is sorted in ascending order, and an item `x`
- We want to write a function that searches for `x` within the array `a`, returning the index of `x` if it is found, and returning `-1` if `x` is not in the array

Binary Search Algorithm



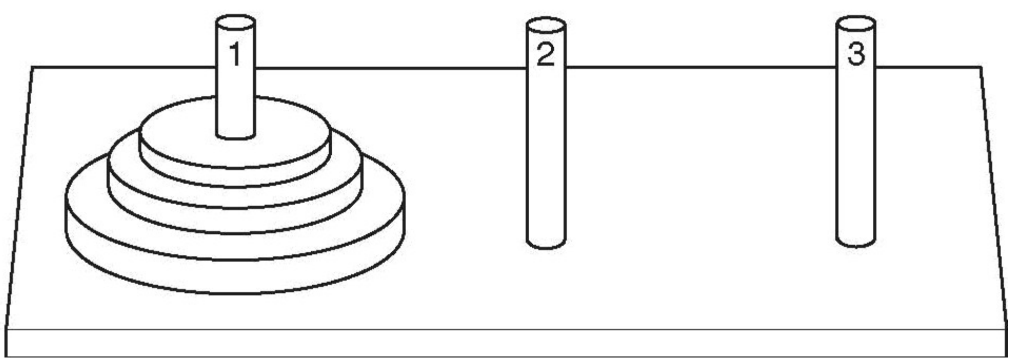
If `a[m] == x`, we found `x`, so return `m`

If `a[m] > x`, recursively search `a[lo..m-1]`

If `a[m] < x`, recursively search `a[m+1..hi]`

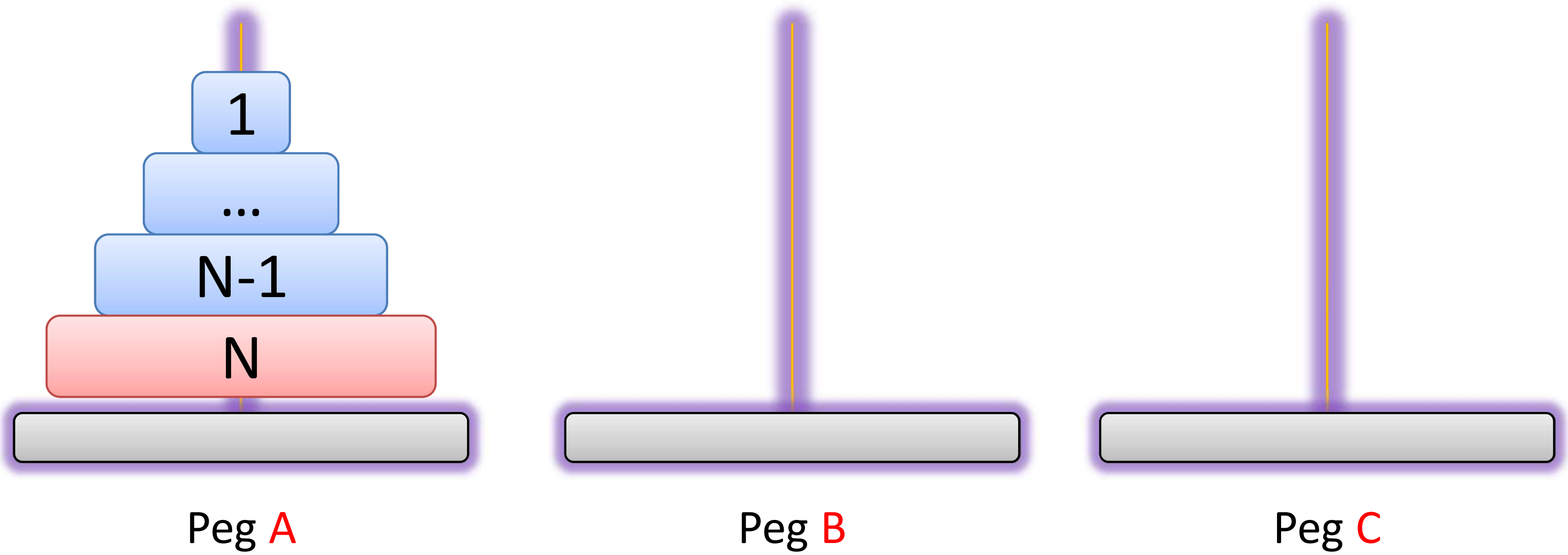
Example 7: Towers of Hanoi

- Setup: 3 pegs, one has n disks on it, the other two pegs empty. The disks are arranged in increasing diameter, top→ bottom
- Objective: move the disks from peg 1 to peg 3, observing
  - only one disk moves at a time
  - all remain on pegs except the one being moved
  - a larger disk cannot be placed on top of a smaller disk at any time

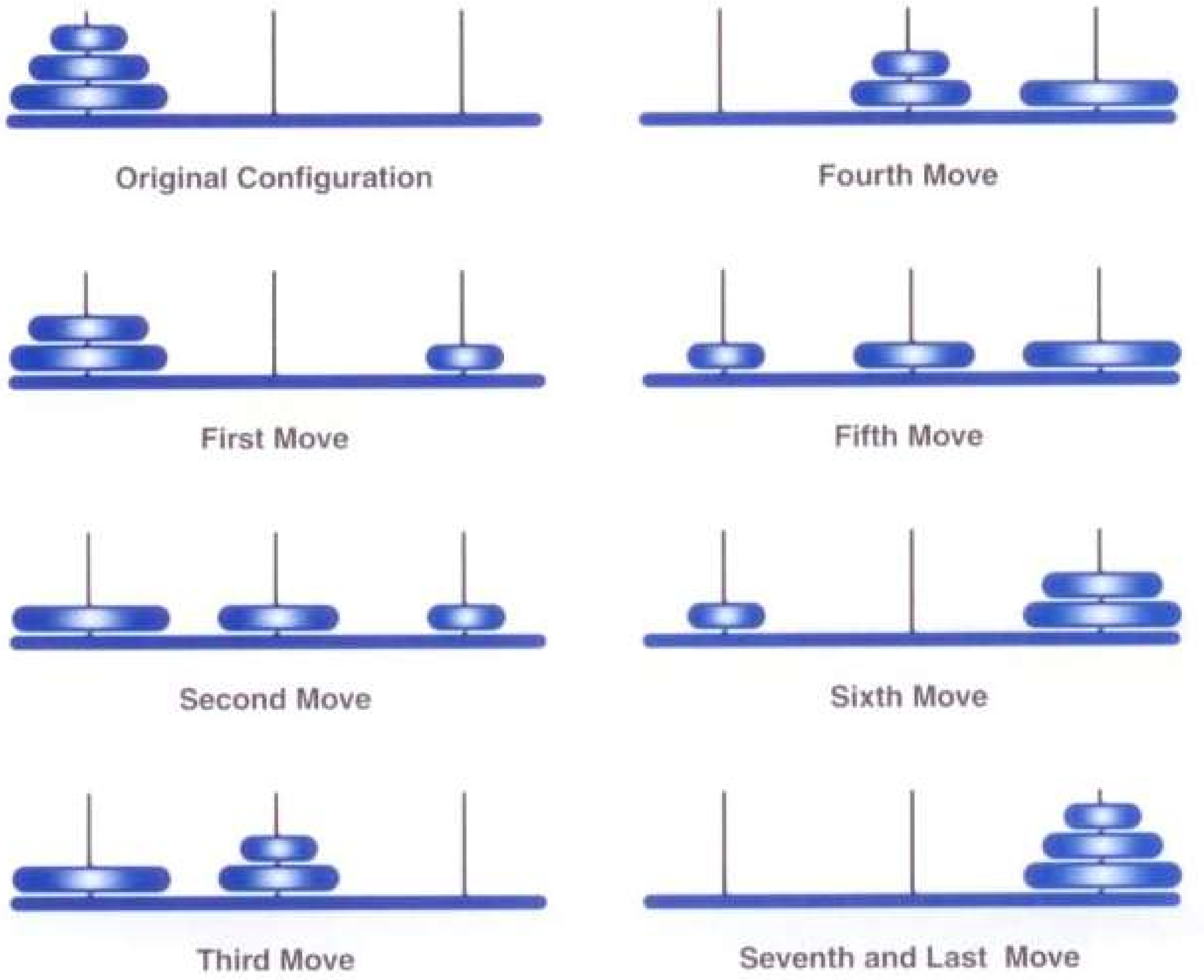


Towers of Hanoi

N disks A → C = ? disks A → B + Disks N A → C + N-1 disks B → C



Towers of Hanoi Algorithm



Towers of Hanoi Algorithm

If n==0, do nothing (base case)

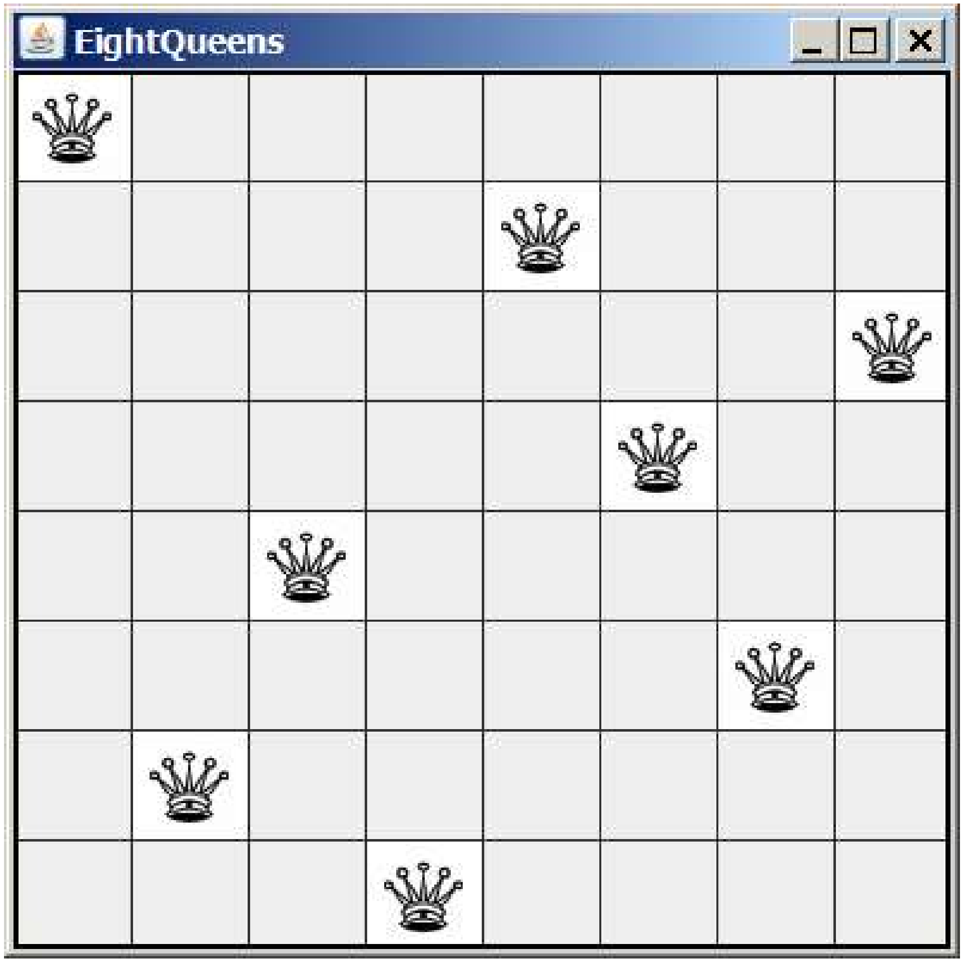
If n>0, then

a. Move the topmost n-1 disks from peg1 to peg2
b. Move the n^{th} disk from peg1 to peg3
c. Move the n-1 disks from peg2 to peg3

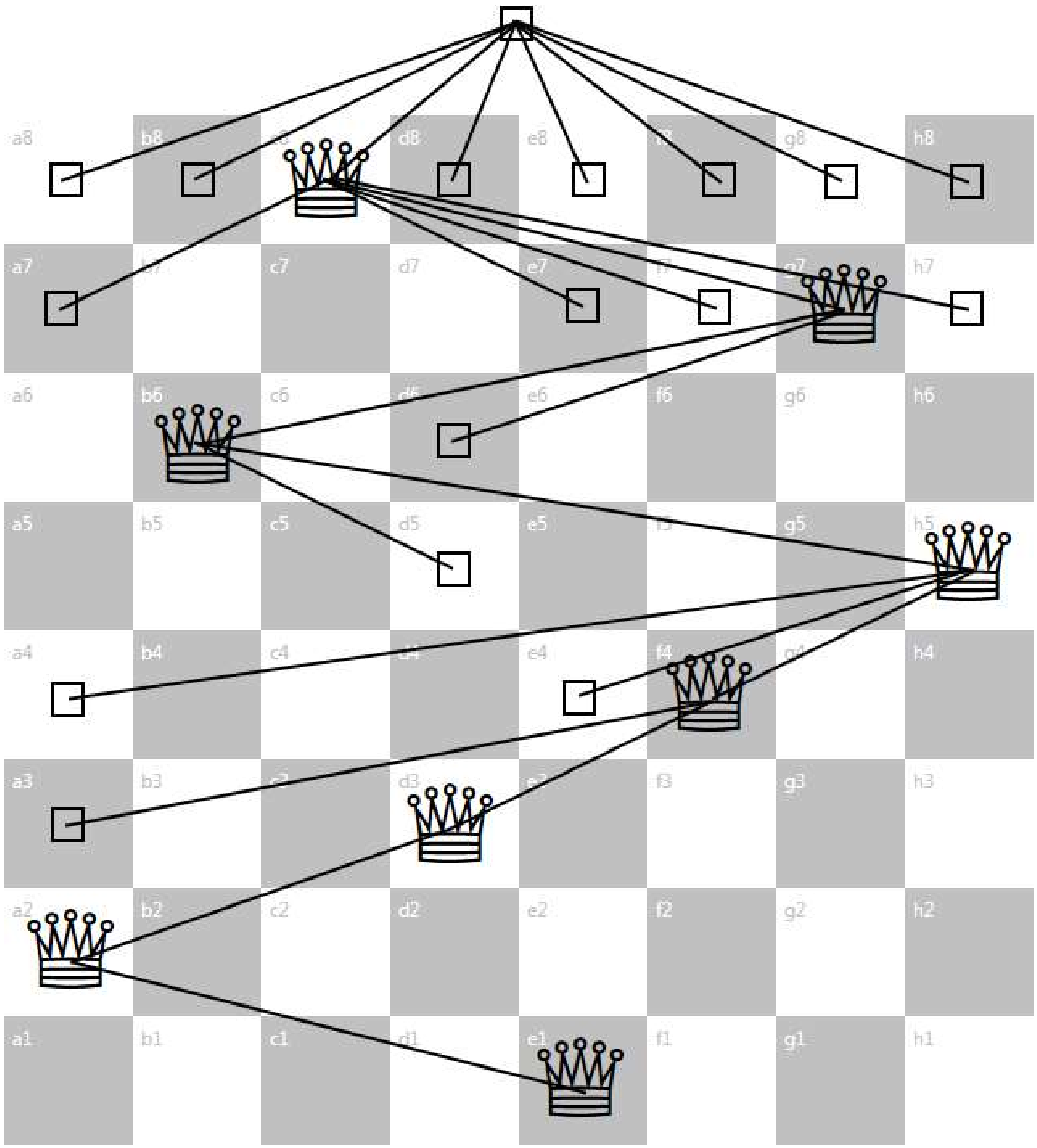
end if

Example 8: Eight Queens

- Place eight queens on the chessboard such that no queen attacks any other one.



Eight Queens Algorithm



The End.