

The slide features a decorative background. On the left, there is a vertical blue wavy band with a repeating pattern of the 'Learn OA' logo and the tagline 'To the Next Level...'. In the top right corner, there are stylized pink icons of a paperclip, a ruler, the Greek letter alpha, a checkmark, and a protractor. The main title is centered on the slide.

# **MODULE 4:** **OBJECT ORIENTED ANALYSIS & DESIGN** **DATA STRUCTURES & ALGORITHMS**

Algorithms Fundamentals

# Data Structures and Algorithms

- **Algorithm**  
Outline, the essence of a computational procedure, step-by-step instructions
- **Program**  
An implementation of an algorithm in some programming language
- **Data structure**  
Organization of data needed to solve the problem

# What is Algorithm?

- A finite set of instructions which accomplish a particular task
- A method or process to solve a problem
- Transforms input of a problem to output
- Algorithm = Input + Process + Output
- Algorithm development is an art – it needs practice, practice and only practice!

# What is a good algorithm?

- It must be correct
- It must be finite (in terms of time and size)
- It must terminate
- It must be unambiguous  
Which step is next?
- It must be space and time efficient
- A program is an instance of an algorithm, written in some specific programming language



# A simple algorithm

- Problem: Find maximum of a, b, c
- Algorithm
  - ✓ Input = a, b, c
  - ✓ Output = max
  - ✓ Process
    - Let max = a
    - If  $b > \text{max}$  then
      - max = b
    - If  $c > \text{max}$  then
      - max = c
    - Display max

Order is very important!!!



# Overall Picture

## Data Structure and Algorithm Design Goals

Correctness



Efficiency



## Implementation Goals

Robustness



Adaptability



Reusability



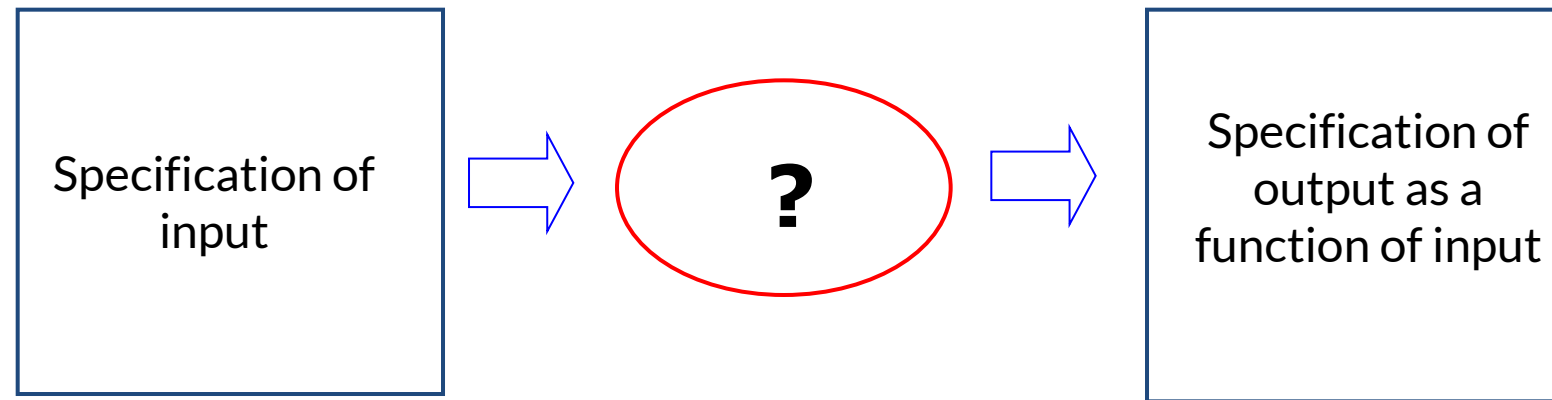


# Overall Picture

- This course is not about:
  - ✓ Programming languages
  - ✓ Computer architecture
  - ✓ Software architecture
  - ✓ Software design and implementation principles
  - ✓ Issues concerning small and large scale programming
- We will only touch upon the theory of complexity and computability



# Algorithmic problem



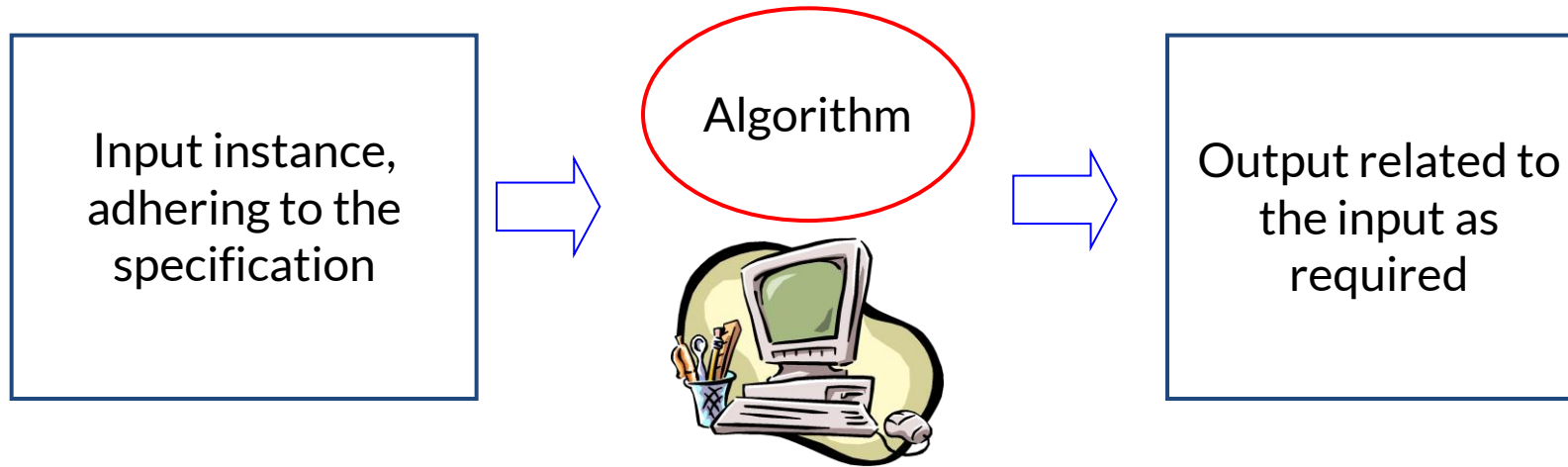
Infinite number of input *instances* satisfying the specification.

For example:

- A sorted, non-decreasing sequence of natural numbers. The sequence is of non-zero, finite length:  
1, 20, 908, 909, 100000, 1000000000.



# Algorithmic Solution



- ✓ Algorithm describes actions on the input instance
- ✓ Infinitely many correct algorithms for the same algorithmic problem

# Example: Sorting

## INPUT

sequence of numbers

$a_1, a_2, a_3, \dots, a_n$

2 5 4 10 7



## OUTPUT

a permutation of the  
sequence of numbers

$b_1, b_2, b_3, \dots, b_n$

2 4 5 7 10

### Correctness

For any given input the algorithm halts with the output:

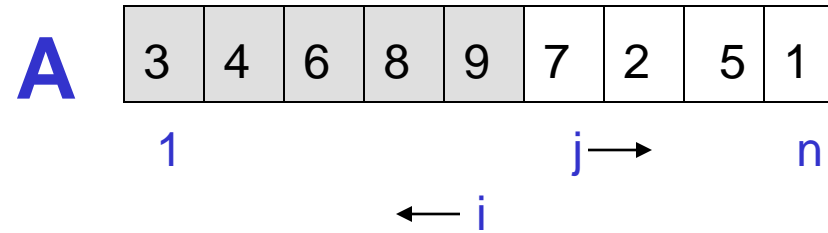
- $b_1 < b_2 < b_3 < \dots < b_n$
- $b_1, b_2, b_3, \dots, b_n$  is a permutation of  $a_1, a_2, a_3, \dots, a_n$

### Running time

Depends on

- number of elements ( $n$ )
- how (partially) sorted they are
- algorithm

# Insertion Sort



## Strategy

- Start “empty handed”
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

```
for j=2 to length(A)
do key=A[j]
  “insert A[j] into the
  sorted sequence A[1..j-1]”
  i=j-1
  while i>0 and A[i]>key
    do A[i+1]=A[i]
    i--
  A[i+1]:=key
```

# Analysis of Algorithms

- Efficiency:
  - ✓ Running time
  - ✓ Space used
- Efficiency as a function of input size:
  - ✓ Number of data elements (numbers, points)
  - ✓ A number of bits in an input number



# The RAM model

- Very important to choose the level of detail.
- The RAM model:
  - ✓ Instructions (each taking constant time):
    - ✓ Arithmetic (add, subtract, multiply, etc.)
    - ✓ Data movement (assign)
    - ✓ Control (branch, subroutine call, return)
  - ✓ Data types – integers and floats



# Analysis of Insertion Sort

Time to compute the **running time** as a function of the **input size**

	<b>cost</b>	<b>times</b>
<b>for</b> j=2 <b>to</b> length(A)	$C_1$	n
<b>do</b> key=A[j]	$C_2$	n-1
"insert A[j] into the sorted sequence A[1..j-1]"	0	n-1
i=j-1	$C_3$	$\sum_{j=2}^{n-1} t_j$
<b>while</b> i>0 <b>and</b> A[i]>key	$C_4$	$\sum_{j=2}^{n-1} (t_j - 1)$
<b>do</b> A[i+1]=A[i]	$C_5$	$\sum_{j=2}^{n-1} (t_j - 1)$
i--	$C_6$	$\sum_{j=2}^{n-1} (t_j - 1)$
A[i+1]:=key	$C_7$	n-1

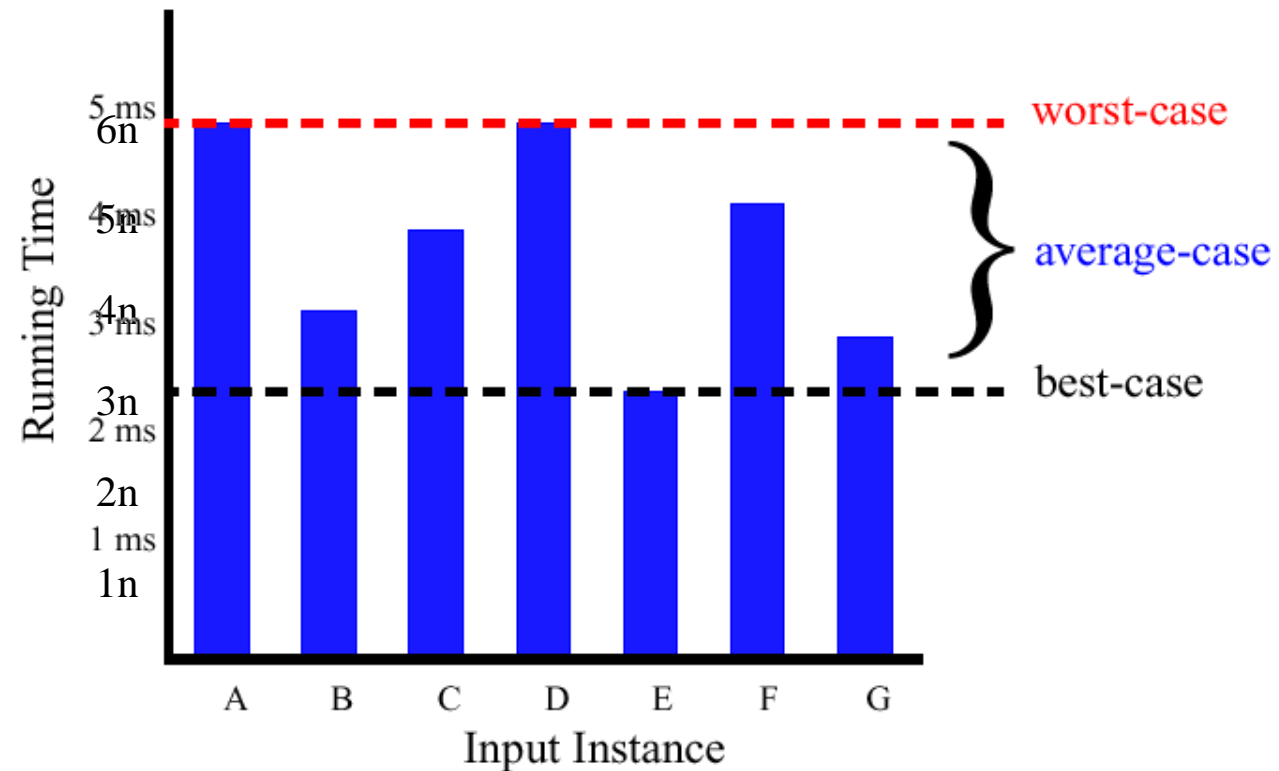


## Best Case / Worst Case / Average Case

- Best case: elements already sorted ®  $t_j=1$ , running time =  $f(n)$ , i.e., linear time.
- Worst case: elements are sorted in inverse order  
®  $t_j=j$ , running time =  $f(n^2)$ , i.e., quadratic time
- Average case:  $t_j=j/2$ , running time =  $f(n^2)$ , i.e., quadratic time

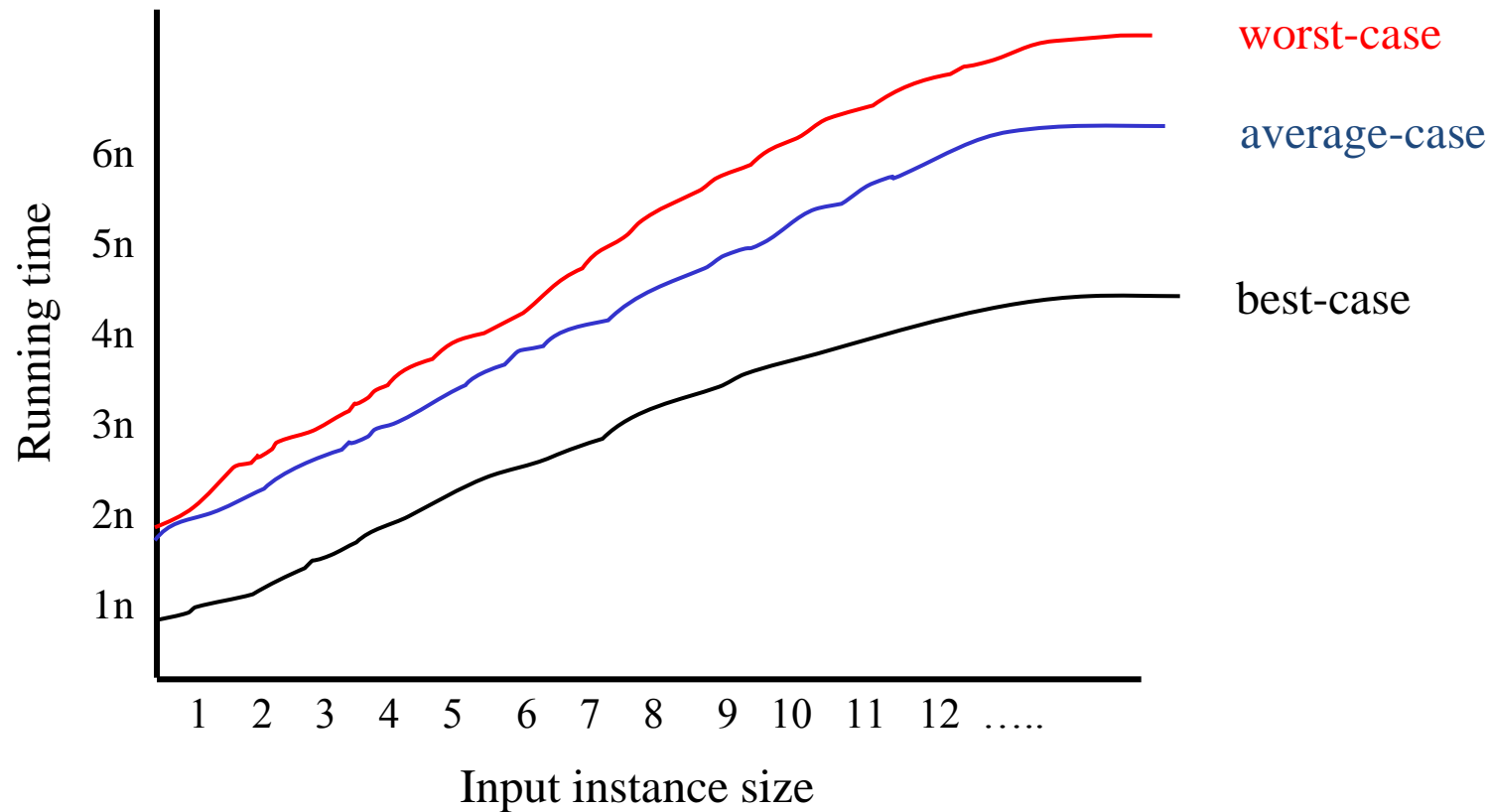
# Best Case / Worst Case / Average Case

For a specific size of input  $n$ , investigate running times for different input instances:



# Best Case / Worst Case / Average Case

For inputs of all sizes:



# Best Case / Worst Case / Average Case

## Worst case is usually used:

- It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the worst-case time complexity is of crucial importance
- For some algorithms worst case occurs fairly often
- The average case is often as bad as the worst case
- Finding the average case can be very difficult

## Example 2: Searching

### INPUT

sequence of numbers (database)

a single number (query)

$a_1, a_2, a_3, \dots, a_n; q$

2 5 4 10 7; 5

2 5 4 10 7; 9

### OUTPUT

an index of the found  
number or NIL

$j$

2

NIL

## Example 2: Searching

```
j=1
while j<=length(A) and A[j]!=q
  do j++
if j<=length(A) then return j
else return NIL
```

- Worst-case running time:  $f(n)$ , average-case:  $f(n/2)$
- We can't do better. This is a lower bound for the problem of searching in an arbitrary sequence.



## Example 3: Searching

### INPUT

- sorted non-descending sequence of numbers (database)
- a single number (query)

$a_1, a_2, a_3, \dots, a_n; q$

2 4 5 7 10; 5

2 4 5 7 10; 9

### OUTPUT

- an index of the found number or NIL

j

2

NIL

# Binary search

- Idea: Divide and conquer, one of the key design techniques

```
left=1
right=length(A)
do
  j=(left+right)/2
  if A[j]==q then return j
  else if A[j]>q then right=j-1
  else left=j+1
while left<=right
return NIL
```

## Binary search – analysis

- How many times the loop is executed:
  - ✓ With each execution its length is cut in half
  - ✓ How many times do you have to cut  $n$  in half to get 1?
  - ✓  $\lg n$

# Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

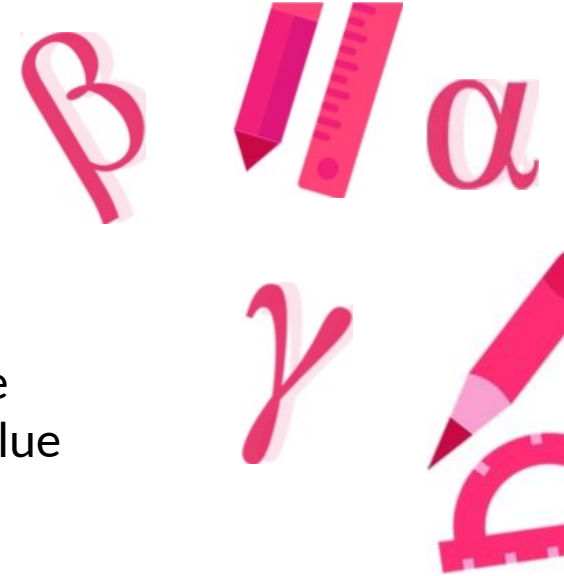
For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation



## Big-O Notation

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

## Omega Notation

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

## Theta Notation

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



# Recursion

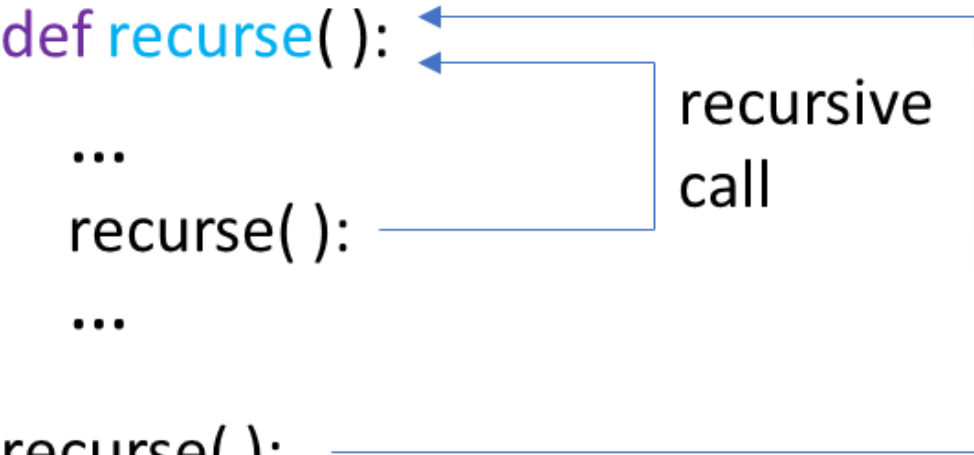




# Recursive Function

- The recursive function is
  - ✓ a kind of function that calls itself, or
  - ✓ a function that is part of a cycle in the sequence of function calls.

```
def recurse( ):  
    ...  
    recurse( ):  
    ...  
recurse( ):
```

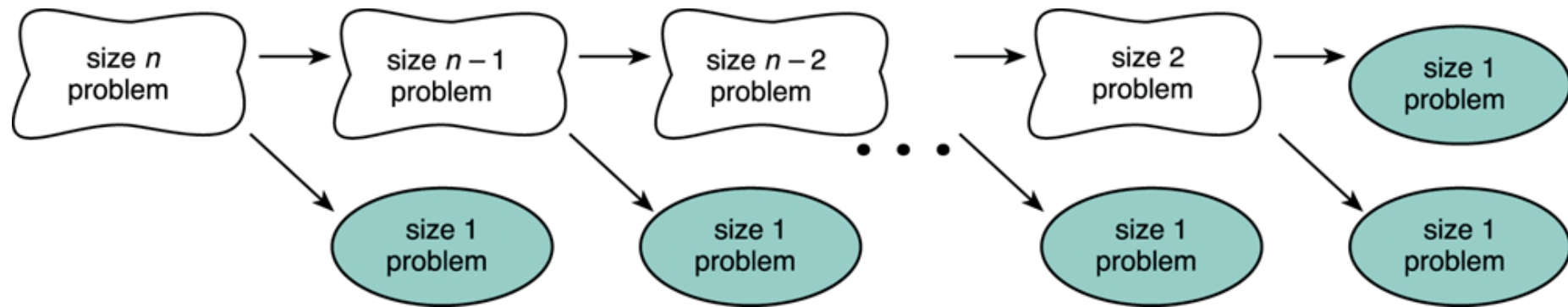


The diagram illustrates the concept of a recursive call. A box labeled "recursive call" has two arrows pointing to the recursive calls within the function definition and the call outside the definition.

# Problems Suitable for Recursive Functions

- One or more simple cases of the problem have a straightforward solution.
- The other cases can be redefined in terms of problems that are closer to the simple cases.
- The problem can be reduced entirely to simple cases by calling the recursive function.
  - ✓ If (this is a simple case)  
    solve it
  - else  
    redefine the problem using recursion

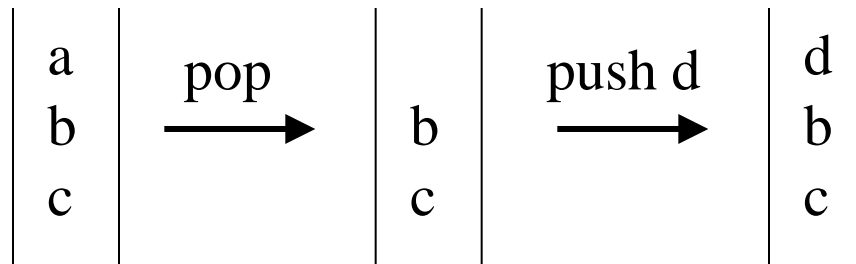
# Splitting a Problem into Smaller Problems



- Assume that the problem of size 1 can be solved easily (i.e., the simple case).
- We can recursively split the problem into a problem of size 1 and another problem of size  $n-1$ .

# How C Maintains the Recursive Steps

- C keeps track of the values of variables by the stack data structure.
  - ✓ Recall that stack is a data structure where the last item added is the first item processed.
  - ✓ There are two operations (push and pop) associated with stack.



## How C Maintains the Recursive Steps

- Each time a function is called, the execution state of the caller function (e.g., parameters, local variables, and memory address) are pushed onto the stack.
- When the execution of the called function is finished, the execution can be restored by popping up the execution state from the stack.
- This is sufficient to maintain the execution of the recursive function.
  - ✓ The execution state of each recursive step are stored and kept in order in the stack.



# Recursive factorial Function

- Many mathematical functions can be defined and solved recursively.  
✓ The following is a function that computes  $n!$ .

```
1.  /*
2.   * Compute n! using a recursive definition
3.   * Pre: n >= 0
4.   */
5.  int
6.  factorial(int n)
7.  {
8.      int ans;
9.
10.     if (n == 0)
11.         ans = 1;
12.     else
13.         ans = n * factorial(n - 1);
14.
15.     return (ans);
16. }
```



# Iterative factorial Function

- The previous factorial function can also be implemented by a for loop.
  - ✓ The iterative implementation is usually more efficient than recursive implementation.

```
1.  /*
2.   * Computes n!
3.   * Pre: n is greater than or equal to zero
4.   */
5.  int
6.  factorial(int n)
7.  {
8.      int i,          /* local variables */
9.      product = 1;
10.
11.     /* Compute the product n x (n-1) x (n-2) x ... x 2 x 1 */
12.     for (i = n; i > 1; --i) {
13.         product = product * i;
14.     }
15.
16.     /* Return function result */
17.     return (product);
18. }
```

Thank You!

