DATA STRUCTURES AND 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 > max then

        max = b

    If c > max then

        max = c

    Display max

Order is very important!!!

Overall Picture

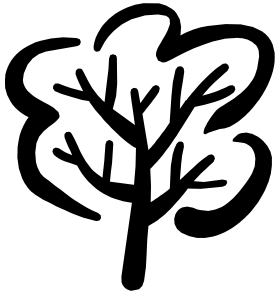
Data Structure and Algorithm Design Goals

Correctness

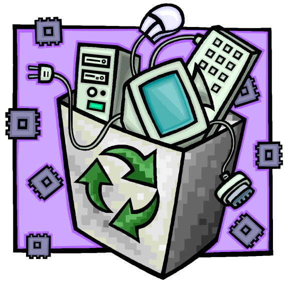
Efficiency

Implementation Goals

Robustness



Reusability



Adaptability

Overall Picture (2)

- 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

Specification of input → ? → Specification of output as a function of input

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

Input instance, adhering to the specification → Algorithm → Output related to the input as required

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

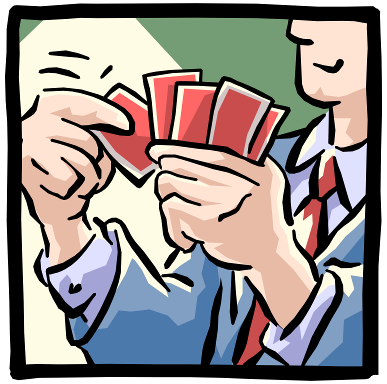
Example: Sorting

INPUT

sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7



OUTPUT

a permutation of the sequence of numbers

b1, b2, b3,….,bn

2 4 5 7 10

Correctness

For any given input the algorithm halts with the output:

- b1 < b2 < b3 < …. < bn
- b1, b2, b3, …., bn is a permutation of a1, a2, a3,….,an

Running time

Depends on

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

Insertion Sort

A

3	4	6	8	9	7	2	5	1

1 j ⟶ n

⟵ i

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**

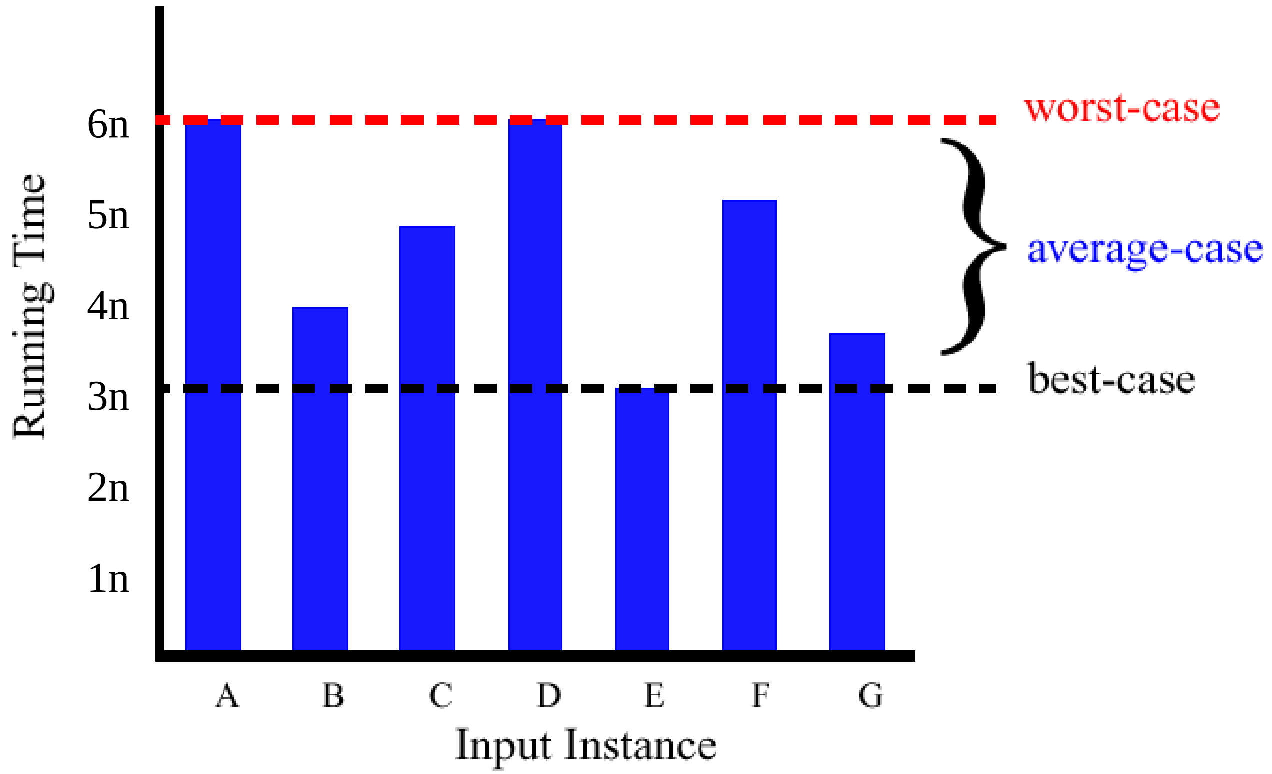
	cost	times
for j=2 **to** *length*(A)	c_1	n
do key=A[j]	c_2	n-1
"insert A[j] into the sorted sequence A[1..j-1]"	0	n-1
i=j-1	c_3	n-1
while i>0 **and** A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
do A[i+1]=A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i--	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1]:=key	c_7	n-1

Best/Worst/Average Case

- Best case: elements already sorted ® tj=1, running time = f(n), i.e., linear time.
- Worst case: elements are sorted in inverse order
  ® tj=j, running time = f(n2), i.e., quadratic time
- Average case: tj=j/2, running time = f(n2), i.e., quadratic time

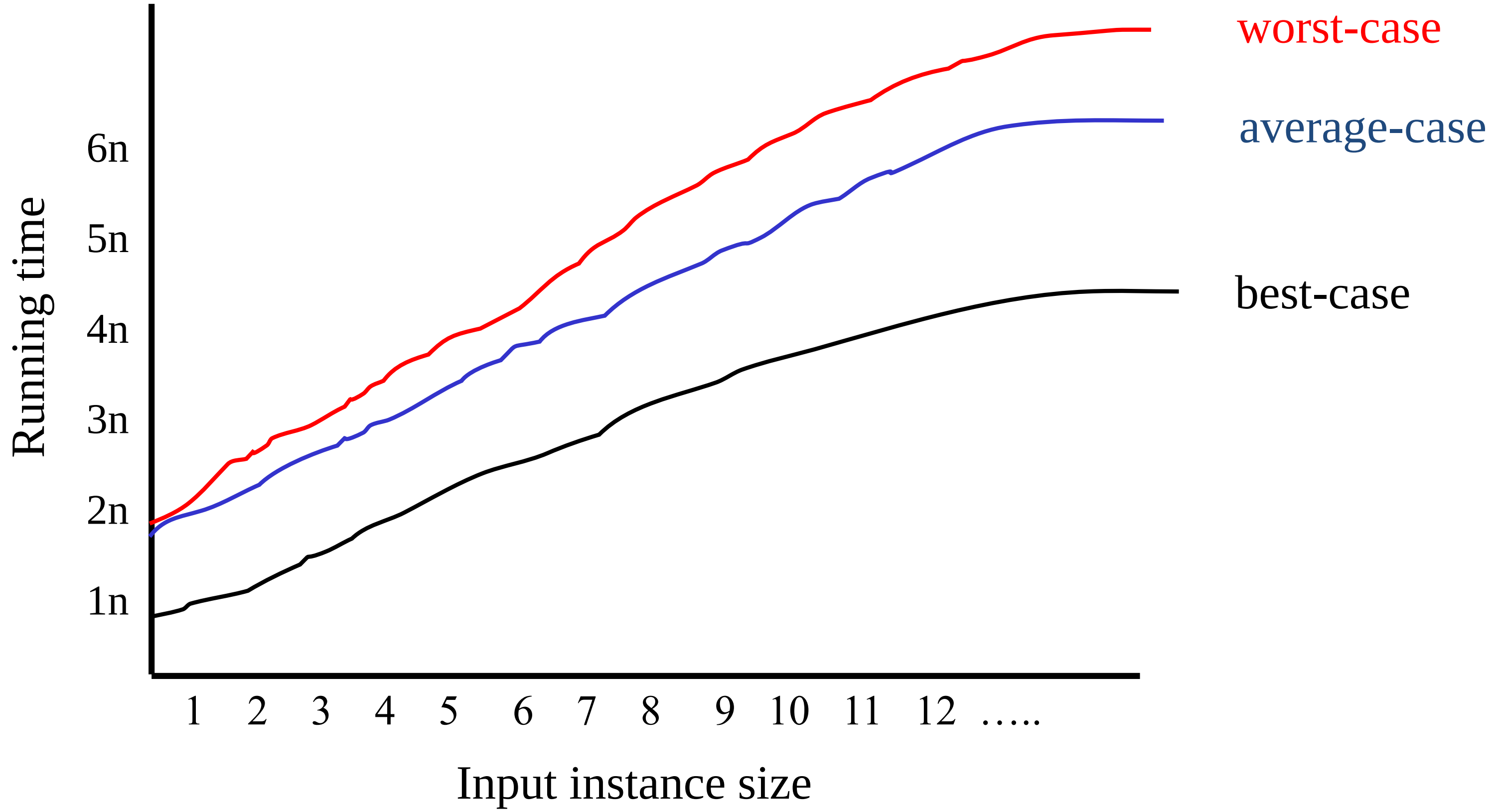
Best/Worst/Average Case (2)

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



Best/Worst/Average Case (3)

For inputs of all sizes:

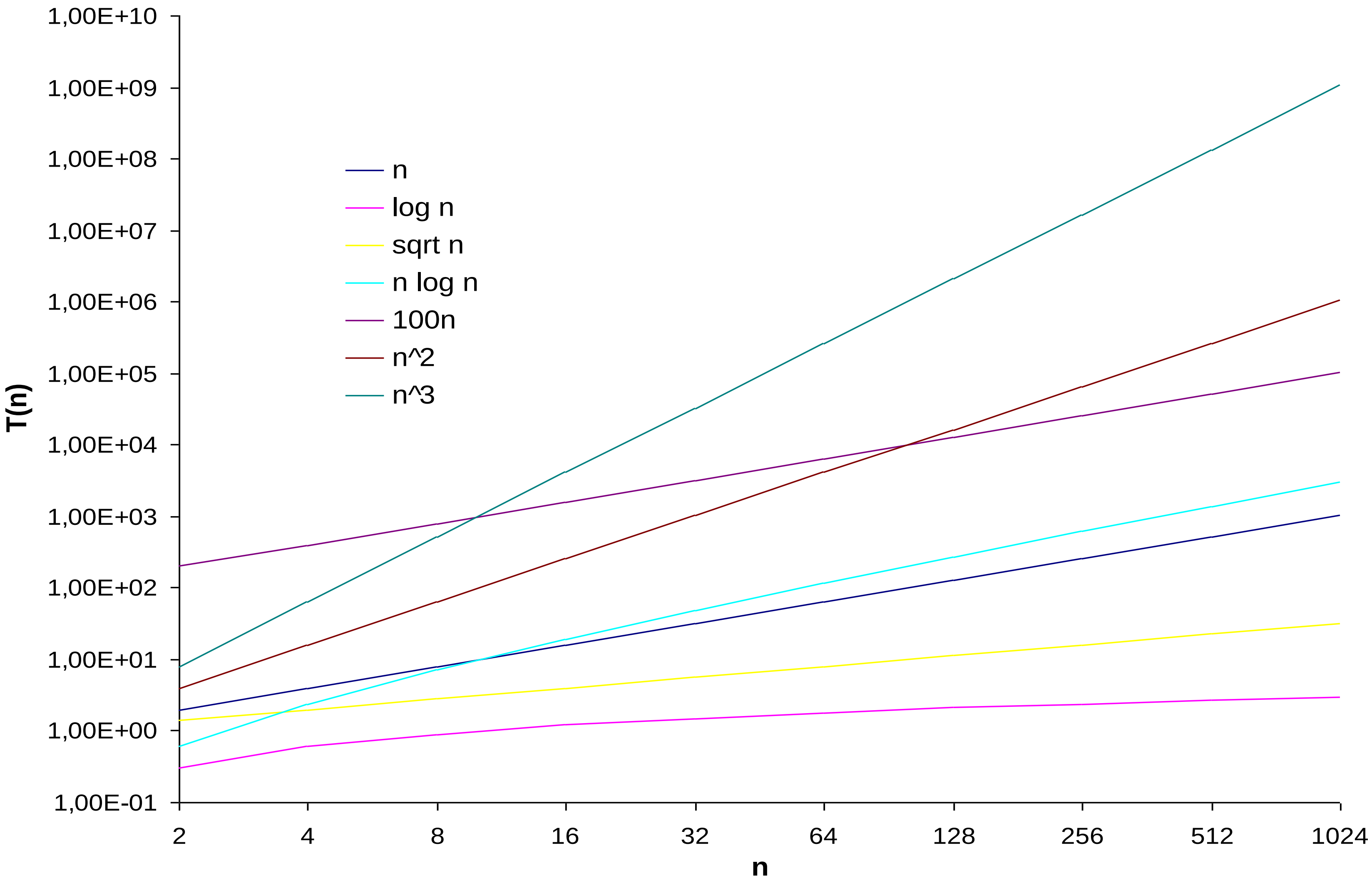


Best/Worst/Average Case (4)

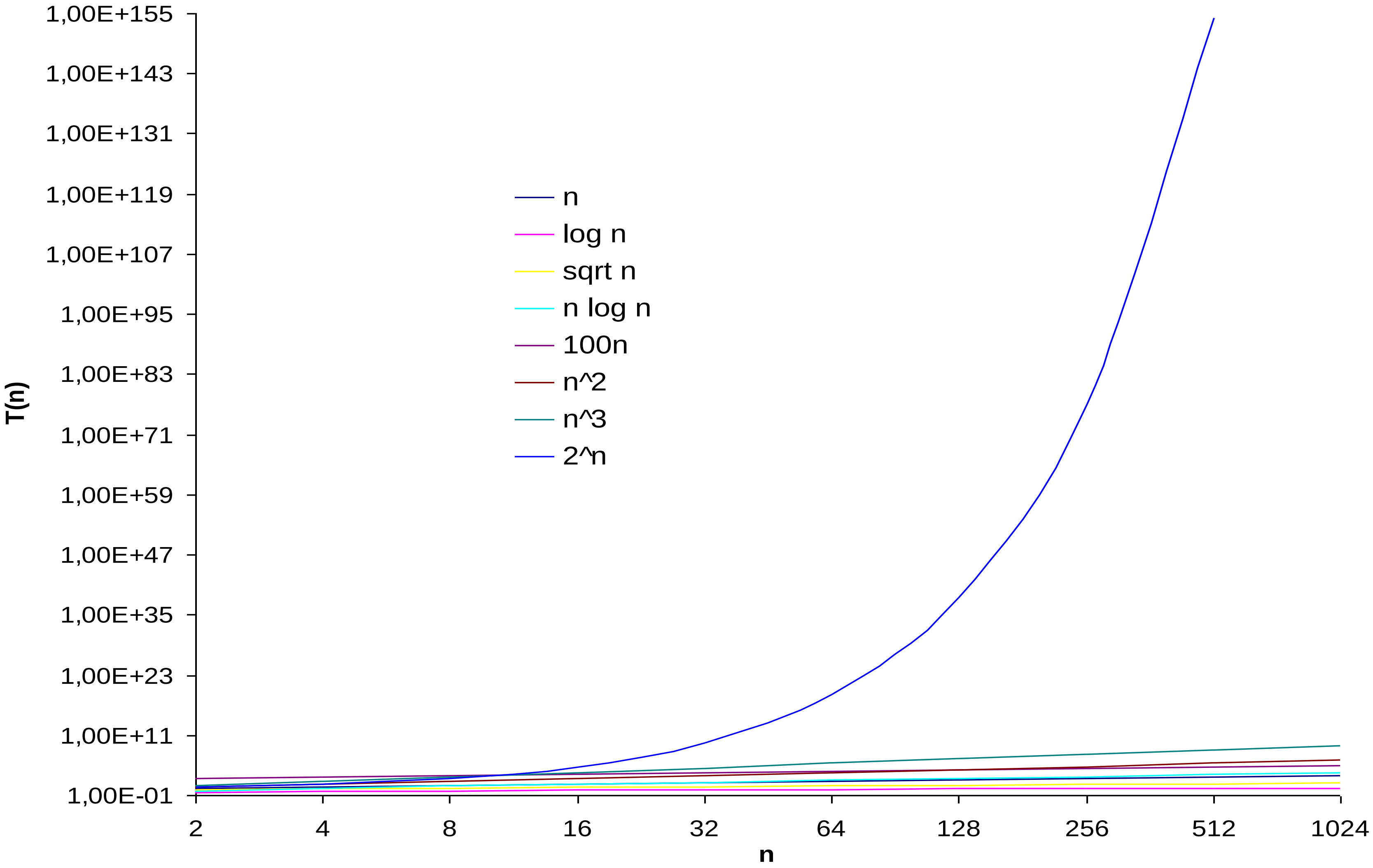
Worst case is usually used:

- 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

Growth Functions



Growth Functions (2)



That’s it?

- Is insertion sort the best approach to sorting?
- Alternative strategy based on divide and conquer
- MergeSort
  - ✓ sorting the numbers <4, 1, 3, 9> is split into
  - ✓ sorting <4, 1> and <3, 9> and
  - ✓ merging the results
  - ✓ Running time f(n log n)

Example 2: Searching

INPUT

sequence of numbers (database)

a single number (query)

$a_1, a_2, a_3, \ldots, a_n;\ q$ $\longrightarrow$ j

2	5	4	10	7;	5		2
2	5	4	10	7;	9		*NIL*

OUTPUT

an index of the found number or NIL

Searching (2)

```
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)

$a_1, a_2, a_3, \ldots, a_n;\ q \longrightarrow$

2 4 5 7 10; 5

2 4 5 7 10; 9

OUTPUT

- an index of the found number or NIL

$\xrightarrow{j}$

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 cult in half
  - ✓ How many times do you have to cut n in half to get 1?
  - ✓ lg n

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 > max then

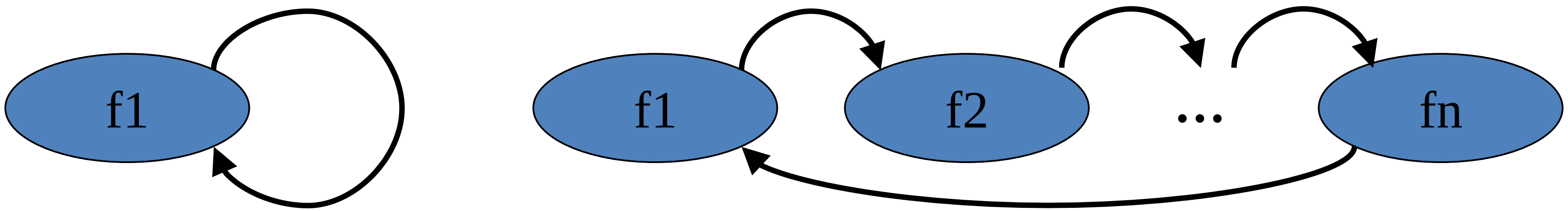
      max = b
    - If c > max then

      max = c
    - Display max
- Order is very important!!!

Recursion

Recursive Function

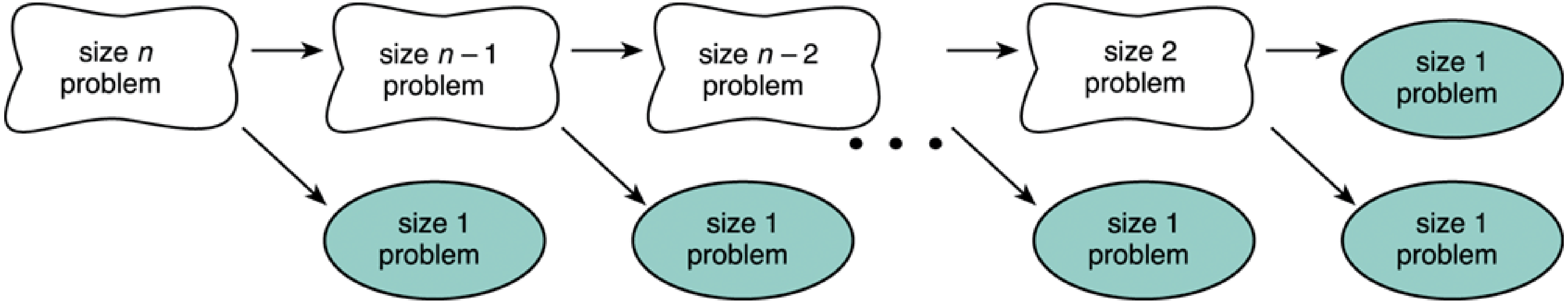
- The recursive function is
  - ✓ a kind of function that calls itself, or
  - ✓ a function that is part of a cycle in the sequence of function calls.



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.

An Example of Recursive Function

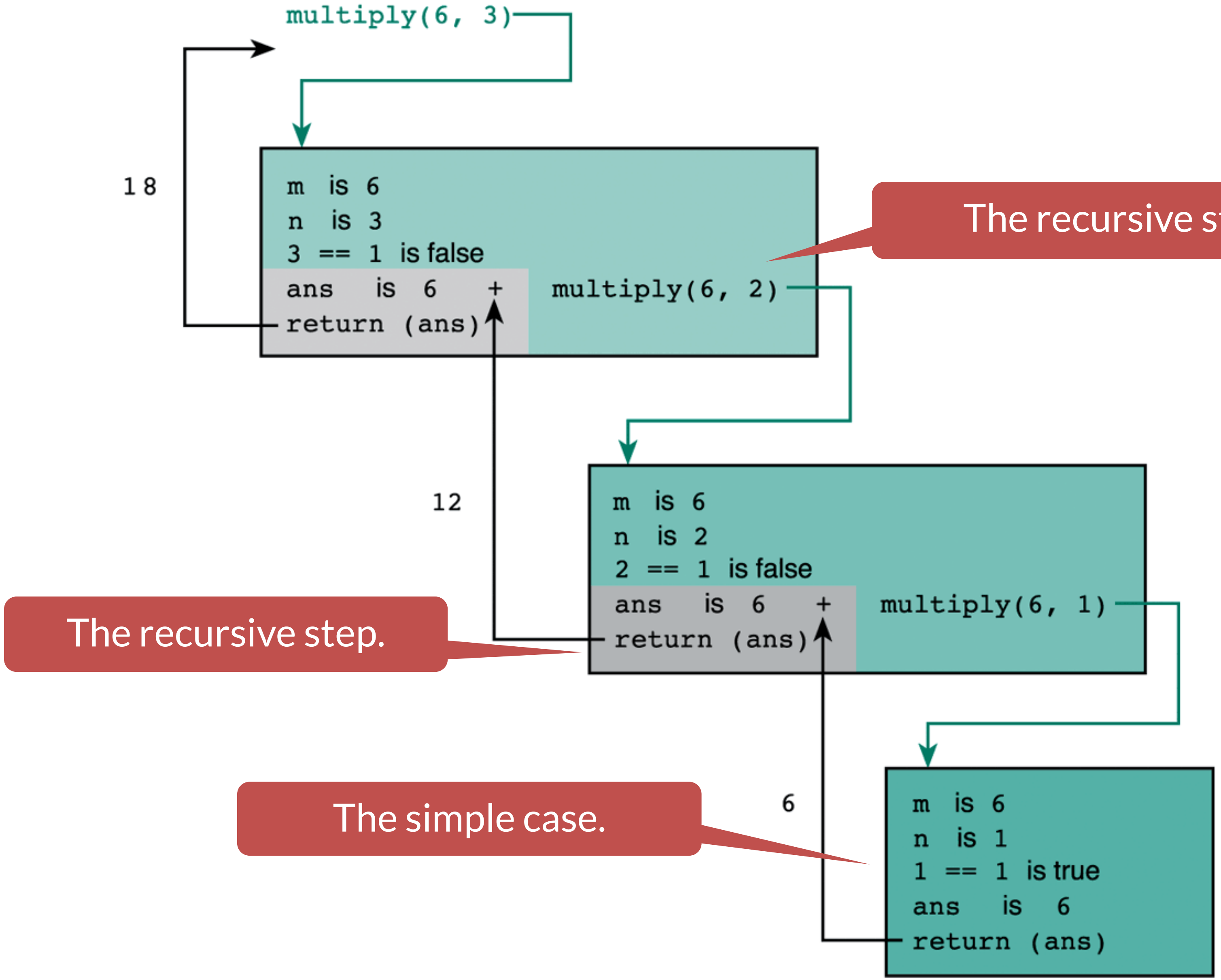
- We can implement the multiplication by addition.

```
/*
 *  Performs integer multiplication using + operator.
 *  Pre:   m and n are defined and n > 0
 *  Post:  returns m * n
 */
int
multiply(int m, int n)
{
      int ans;

      if (n == 1)
            ans = m;      /* simple case */
      else
            ans = m + multiply(m, n - 1);  /* recursive step */

      return (ans);
}
```

The simple case is “m*1=m.”

The recursive step uses the following equation: “m*n = m+m*(n-1).”

Trace of Function multiply(6,3)



Terminating Condition

- The recursive functions always contains one or more terminating conditions.
  - ✓ A condition when a recursive function is processing a simple case instead of processing recursion.

- Without the terminating condition, the recursive function may run forever.
  - ✓ e.g., in the previous multiply function, the if statement “if (n == 1) …” is the terminating condition.

Recursive Function to Count a Character in a String

- We can count the number of occurrences of a given character in a string.
  - ✓ e.g., the number of ‘s’ in “Mississippi” is 4.

```
/*
 *  Count the number of occurrences of character ch in string str
 */
int
count(char ch, const char *str)
{

      int ans;

      if (str[0] == '\0')                            /*  simple case  */
            ans = 0;
      else                         /*  redefine problem using recursion */
            if (ch == str[0])      /*  first character must be counted  */
                  ans = 1 + count(ch, &str[1]);
            else                   /*  first character is not counted   */
                  ans = count(ch, &str[1]);

      return (ans);
}
```

The terminating condition.

A Recursive Function that Reverses Input Words (1/2)

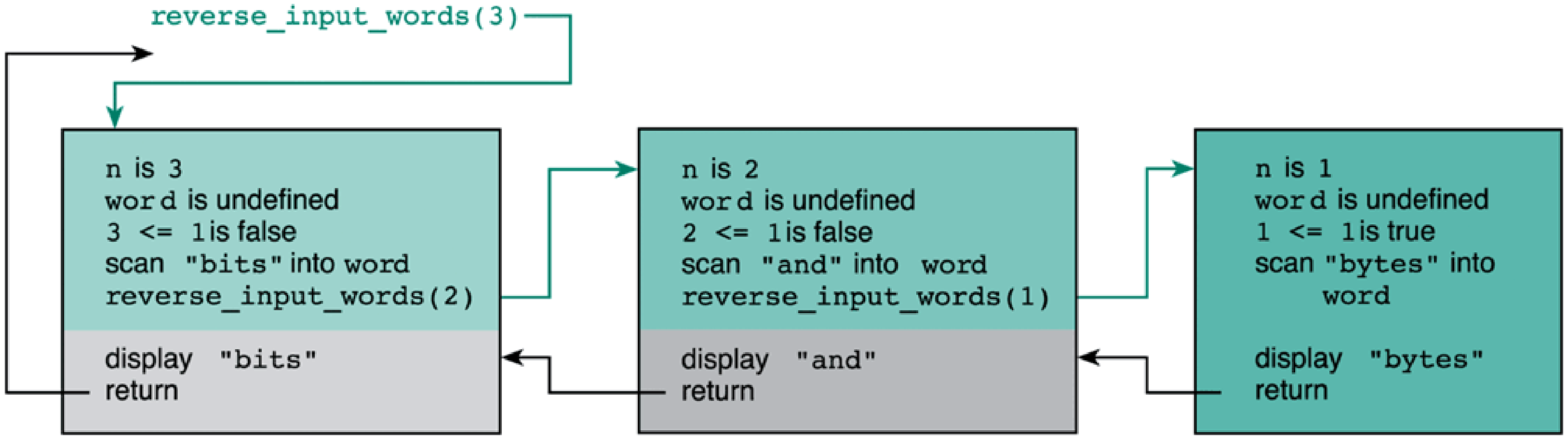
- The recursive concept can be used to reverse an input string.
  - ✓ It can also be done without recursion.

```
/*
 *  Take n words as input and print them in reverse order on separate lines.
 *  Pre: n > 0
 */
void
reverse_input_words(int n)
{
      char word[WORDSIZ];  /*  local variable for storing one word            */

      if (n <= 1) {   /* simple case: just one word to get and print         */

            scanf("%s", word);
            printf("%s\n", word);

      } else {  /* get this word; get and print the rest of the words in
                   reverse order; then print this word                       */

            scanf("%s", word);
            reverse_input_words(n - 1);
            printf("%s\n", word);
      }
}
```

The scanned word will not be printed until the recursion finishes.

The first scanned word is last printed.

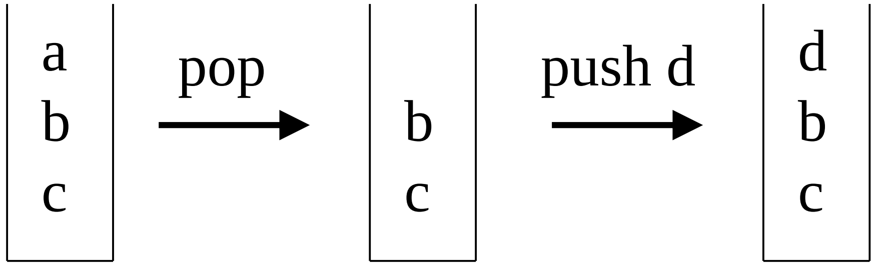
A Recursive Function that Reverses Input Words (2/2)



- Note that the recursive function is just an alternative solution to a problem.
  - ✓ You can always solve the problem without recursion.

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.

How to Trace Recursive Functions

- The recursive function is not easy to trace and to debug.
  - ✓ If there are hundreds of recursive steps, it is not useful to set the breaking point or to trace step-by-step.
- A naïve but useful approach is inserting printing statements and then watching the output to trace the recursive steps.

```
/*
 *   ***  Includes calls to printf to trace execution ***
 *   Performs integer multiplication using + operator.
 *   Pre:    m and n are defined and n > 0
 *   Post:   returns m * n
 */
int
multiply(int m, int n)
{
        int ans;

   printf("Entering multiply with m = %d, n = %d\n", m, n);

        if (n == 1)
              ans = m;      /* simple case */
        else
              ans = m + multiply(m, n - 1); /* recursive step */
```

Watch the input arguments passed into each recursive step.

(continued)

Recursive factorial Function

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

```
/*
 *  Compute n! using a recursive definition
 *  Pre:  n >= 0
 */
int
factorial(int n)
{
      int ans;

      if (n == 0)
            ans = 1;
      else
            ans = n * factorial(n - 1);

      return (ans);
}
```

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.

```
/*
 * Computes n!
 * Pre: n is greater than or equal to zero
 */
int
factorial(int n)
{
    int i,              /* local variables */
        product = 1;

    /* Compute the product n x (n-1) x (n-2) x ... x 2 x 1 */
    for  (i = n;  i > 1;  --i) {
        product = product * i;
    }

    /* Return function result */
    return (product);
}
```

Recursive fibonacci Function

- In our midterm, you are required to write a fibonacci function.
  - ✓ It can be easily solved by the recursive function
  - ✓ But the recursive function is inefficient because the same fibonacci values may be computed more than once.

```
/*
 *  Computes the nth Fibonacci number
 *  Pre: n > 0
 */
int
fibonacci(int n)
{
      int ans;

      if (n == 1  ||  n == 2)
            ans = 1;
      else
            ans = fibonacci(n - 2)  +  fibonacci(n - 1);

      return (ans);
}
```

Recursive gcd Function

- Generally speaking, if the algorithm to a problem is defined recursively in itself, we would tend to use the recursive function.
- e.g., the greatest common divisor (GCD) of two integers m and n can be defined recursively.
  - ✓ gcd(m,n) is n if n divides m evenly;
  - ✓ gcd(m,n) is gcd(n, remainder of m divided by n) otherwise.

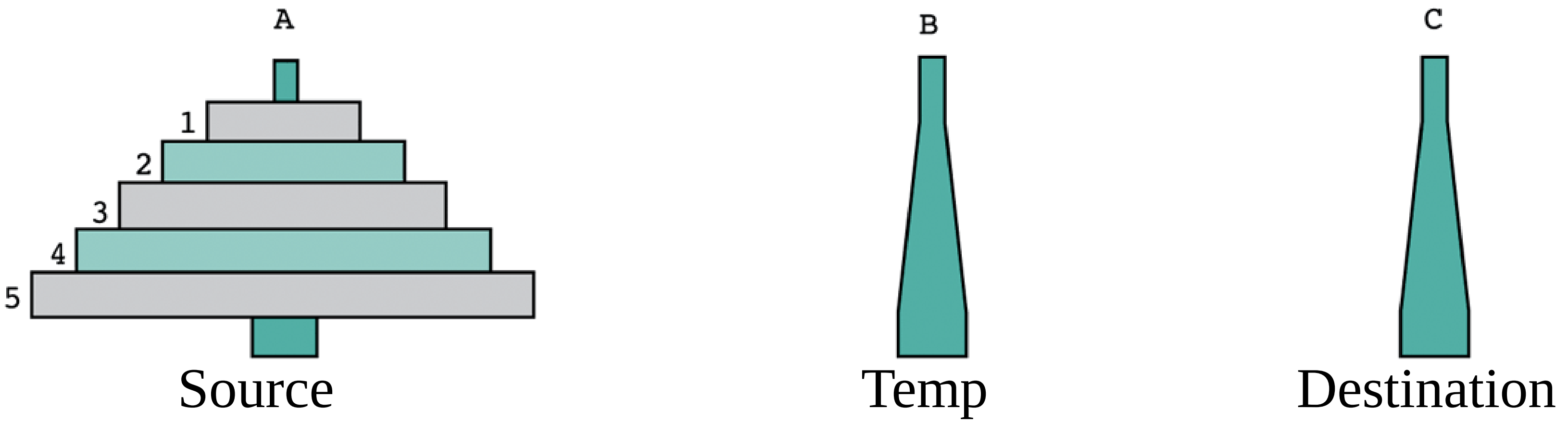
Recursive gcd Function (Contd.)

```
/*
 *  Displays the greatest common divisor of two integers
 */

#include <stdio.h>

/*
 *  Finds the greatest common divisor of m and n
 *  Pre:  m and n are both > 0
 */
int
gcd(int m, int n)
{
      int ans;

      if (m % n == 0)
            ans = n;
      else
            ans = gcd(n, m % n);

      return (ans);
}
```

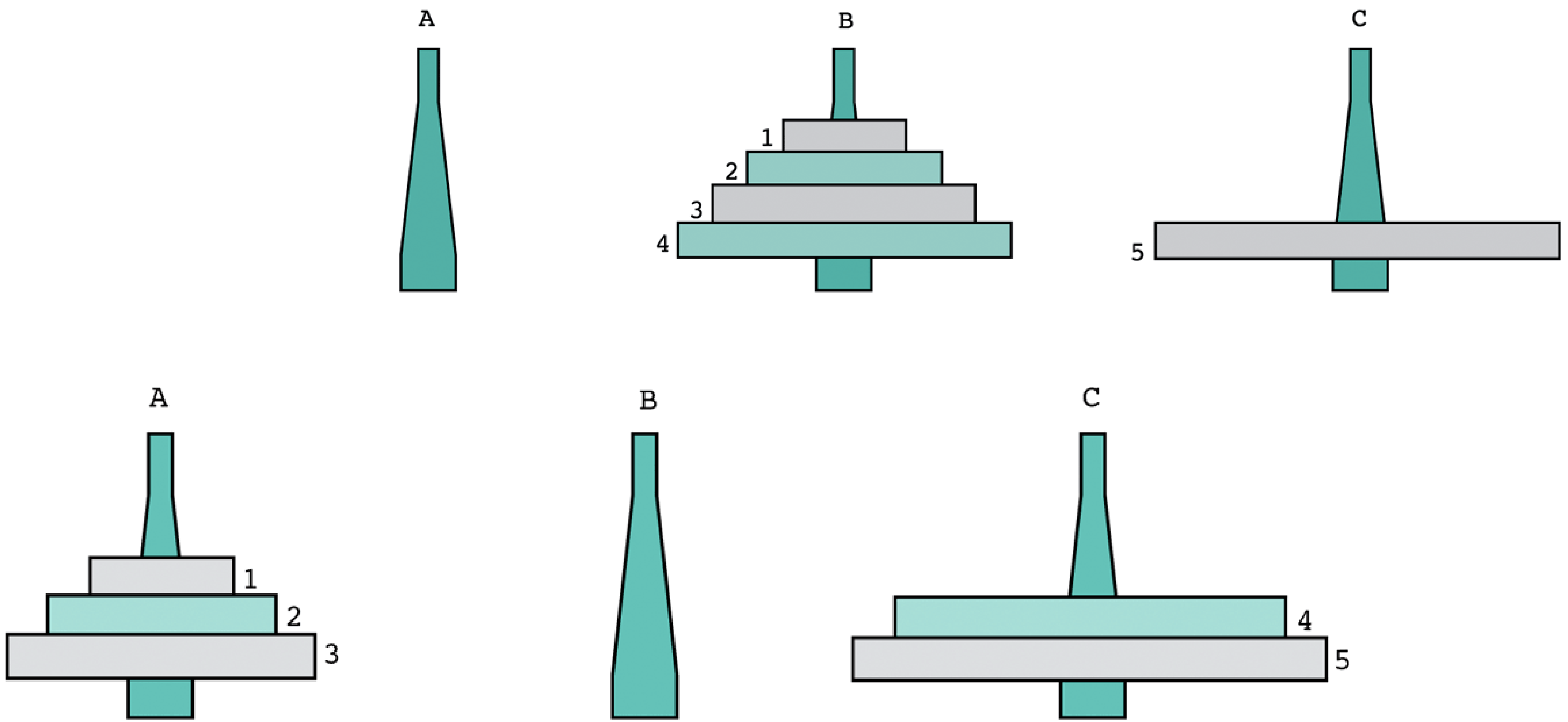
(continued)

A Classical Case: Towers of Hanoi

- The towers of Hanoi problem involves moving a number of disks (in different sizes) from one tower (or called "peg") to another.
  - ✓ The constraint is that the larger disk can never be placed on top of a smaller disk.
  - ✓ Only one disk can be moved at each time
  - ✓ Assume there are three towers available.



A Classical Case: Towers of Hanoi



A Classical Case: Towers of Hanoi

- This problem can be solved easily by recursion.

Algorithm:

- if n is 1 then
  move disk 1 from the source tower to the destination tower
- Else

  1. move n-1 disks from the source tower to the temp tower.

  2. move disk n from the source tower to the destination tower.
  3. move n-1 disks from the temp tower to the source tower.

A Classical Case: Towers of Hanoi

```
/*
 *  Displays instructions for moving n disks from from_peg to to_peg using
 *  aux_peg as an auxiliary.  Disks are numbered 1 to n (smallest to
 *  largest). Instructions call for moving one disk at a time and never
 *  require placing a larger disk on top of a smaller one.
 */
void
tower(char from_peg,     /* input - characters naming            */
      char to_peg,       /*         the problem's                */
      char aux_peg,      /*         three pegs                   */
      int n)             /* input - number of disks to move      */
{
      if (n == 1) {
            printf("Move disk 1 from peg %c to peg %c\n", from_peg, to_peg);
      } else {
            tower(from_peg, aux_peg, to_peg, n - 1);
            printf("Move disk %d from peg %c to peg %c\n", n, from_peg, to_peg);
            tower(aux_peg, to_peg, from_peg, n - 1);
      }
}
```

The recursive step

The recursive step

A Classical Case: Towers of Hanoi

- The execution result of calling Tower('A', 'B', 'C',3);

Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

Thank You!