PROGRAMMING FOR PROBLEM SOLVING -- PPS

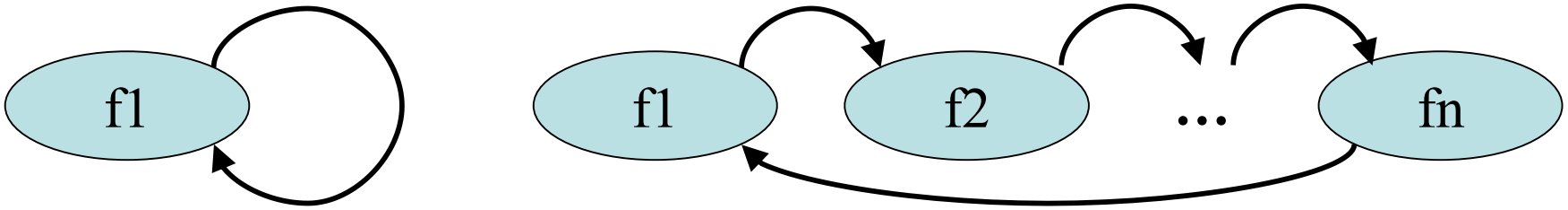
SUBJECT CODE-
BTPS-101-18

PPS

Unit-6 Recursion

Recursive Function

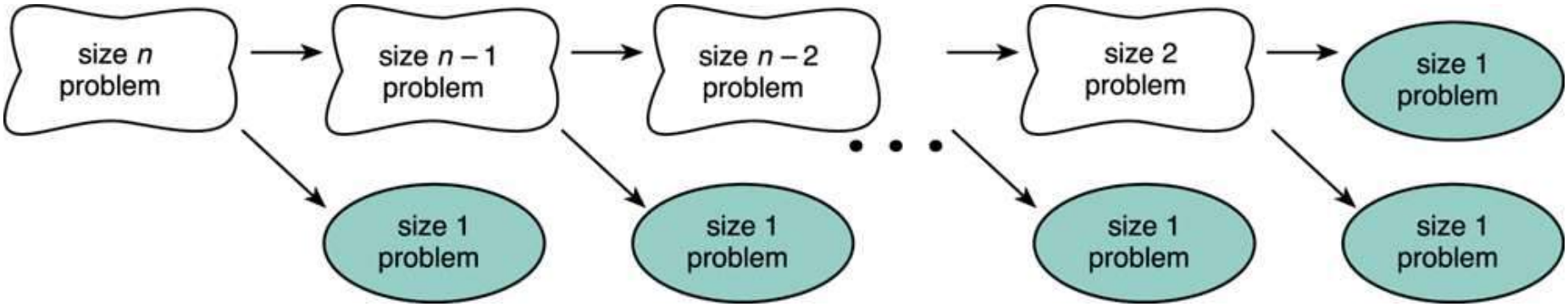
- The **recursive function** is
  - a kind of function that calls itself, or
  - a function that is part of a cycle in the sequence of function calls.



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

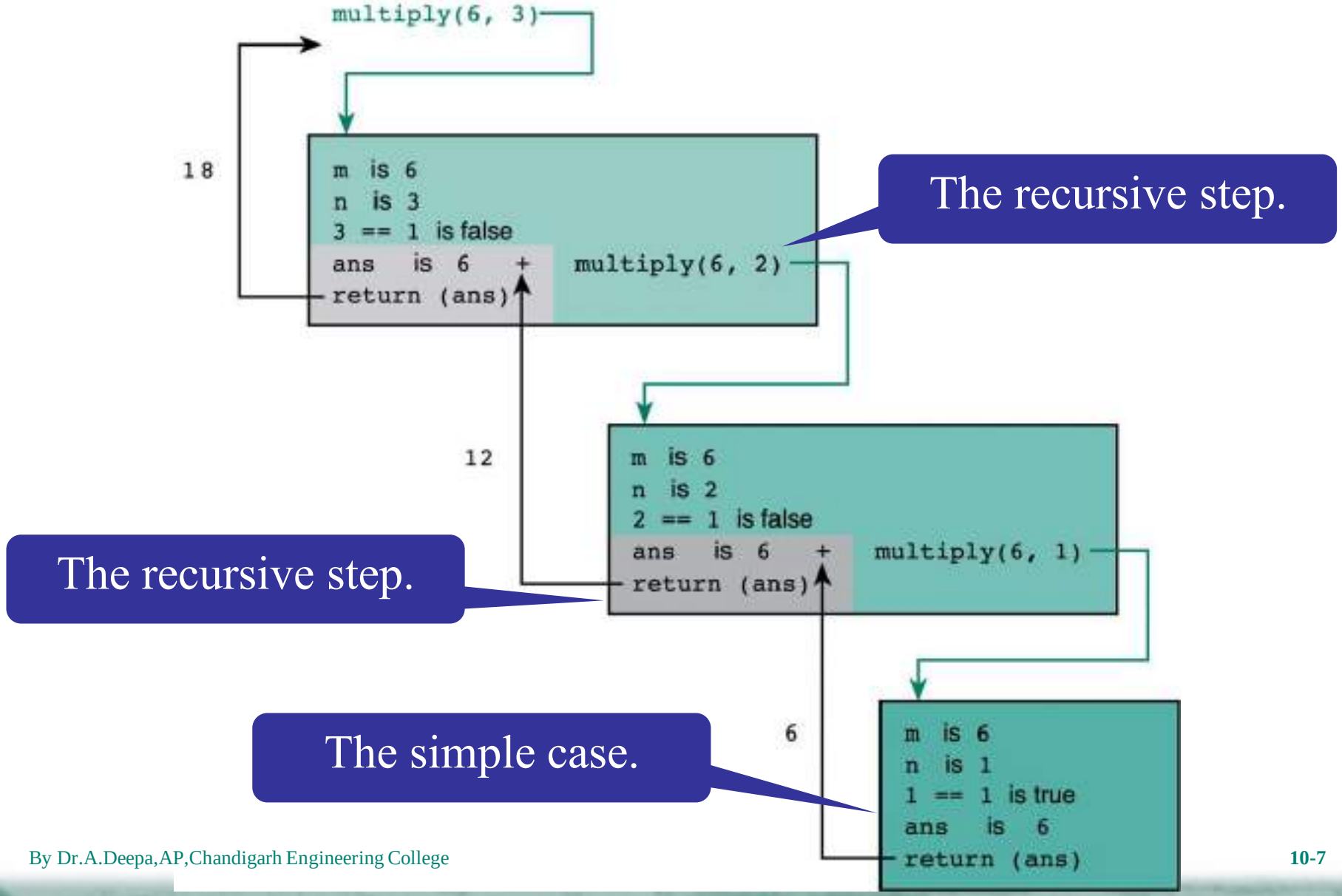
- We can implement the multiplication by addition.

```
1.  /*
2.   *  Performs integer multiplication using + operator.
3.   *  Pre:   m and n are defined and n > 0
4.   *  Post:  returns m * n
5.   */
6.  int
7.  multiply(int m, int n)
8.  {
9.        int ans;
10.
11.       if (n == 1)
12.             ans = m;      /* simple case */
13.       else
14.             ans = m + multiply(m, n - 1);  /* recursive step */
15.
16.       return (ans);
17. }
```

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.

```
4.  int
5.  count(char ch, const char *str)
6.  {
7.
8.        int ans;
9.
10.       if (str[0] == '\0')                          /*  simple case  */
11.             ans = 0;
12.       else                          /*  redefine problem using recursion */
13.             if (ch == str[0])       /*  first character must be counted  */
14.                   ans = 1 + count(ch, &str[1]);
15.             else                    /*  first character is not counted   */
16.                   ans = count(ch, &str[1]);
17.
18.       return (ans);
19. }
```

The terminating condition.

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

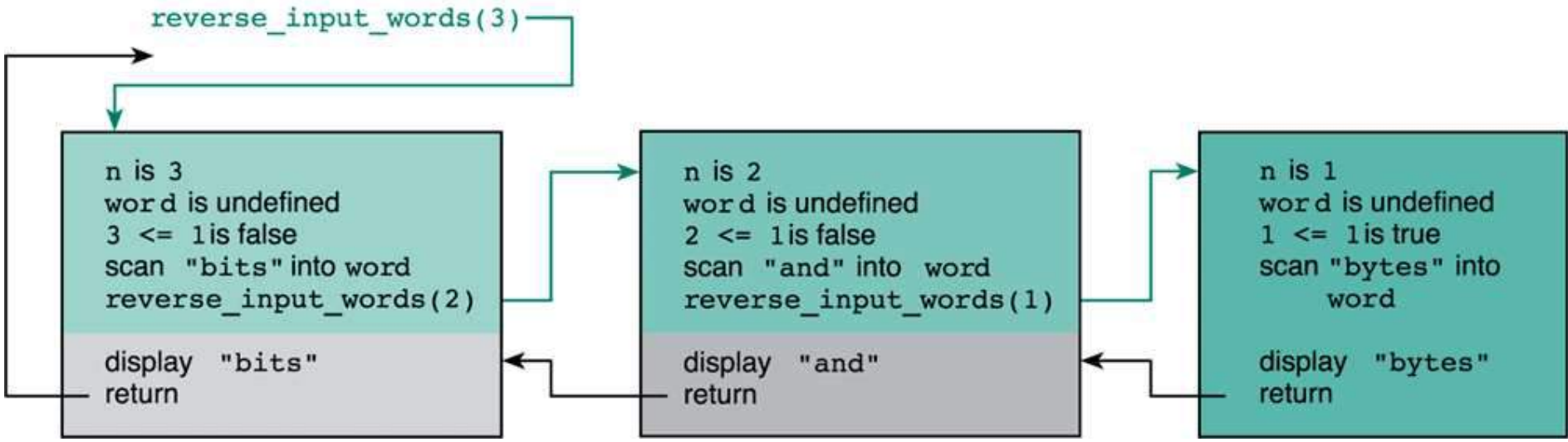
- The recursive concept can be used to reverse an input string.
  - It can also be done without recursion.

```
6.  reverse_input_words(int n)
7.  {
8.        char word[WORDSIZ];  /*  local variable for storing one word          */
9.
10.       if (n <= 1) {    /* simple case: just one word to get and print      */
11.
12.             scanf("%s", word);
13.             printf("%s\n", word);
14.
15.       } else {  /* get this word;
16.                  reverse order;
17.
18.             scanf("%s", word);
19.             reverse_input_words(n - 1);
20.             printf("%s\n", word);
21.       }
22. }
```

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

The first scanned word is last printed.

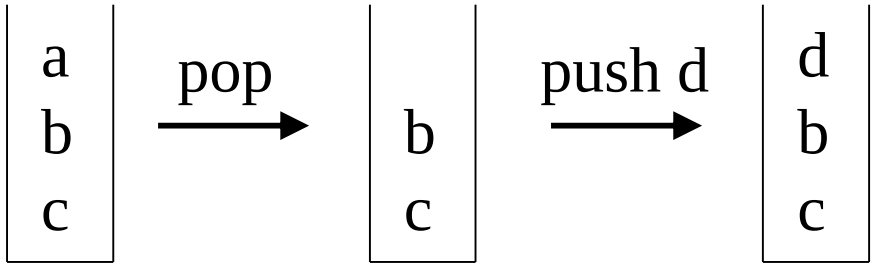
A Recursive Function that Reverses Input Words (2/2)



- 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.

```
7.  int
8.  multiply(int m, int n)
9.  {
10.       int ans;
11.
12.   printf("Entering multiply with m = %d, n = %d\n", m, n);
13.
14.       if (n == 1)
15.            ans = m;      /* simple case */
16.       else
17.            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!**.

```
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. }
```

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.

```
 4.  */
 5.  int
 6.  fibonacci(int n)
 7.  {
 8.        int ans;
 9.
10.        if (n == 1  ||  n == 2)
11.              ans = 1;
12.        else
13.              ans = fibonacci(n - 2)  +  fibonacci(n - 1);
14.
15.        return (ans);
16.  }
```

Recursive gcd Function

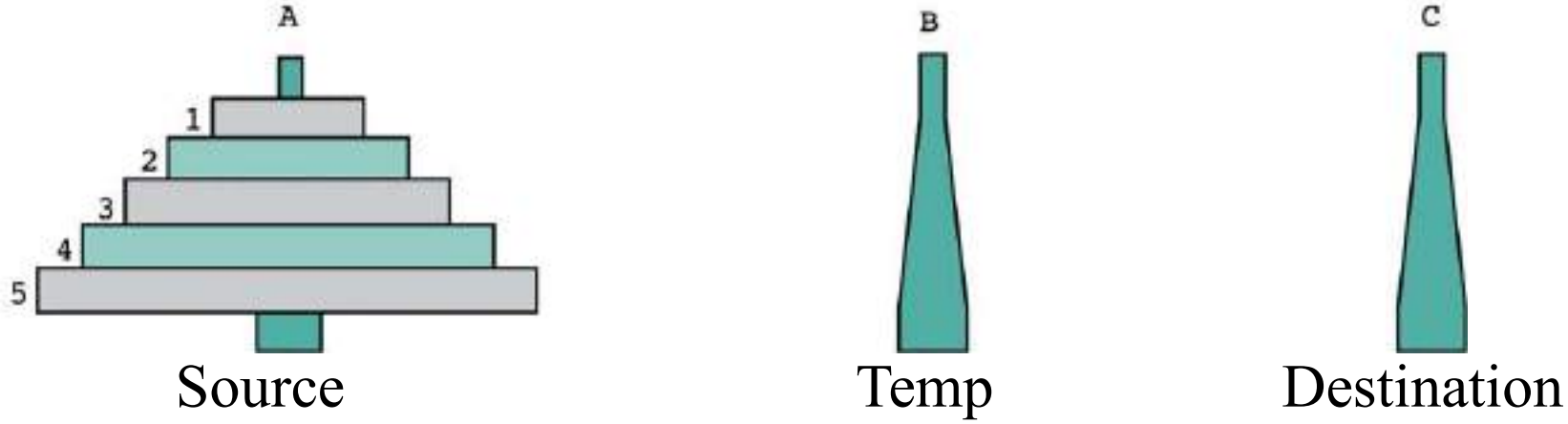
- Generally speaking, if the algorithm to a problem is defined recursively in itself, we would tend to use the recursive function.
- e.g., the greatest common divisor (GCD) of two integers m and n can be defined recursively.
  - gcd(m,n) is n if n divides m evenly;
  - gcd(m,n) is gcd(n, remainder of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

```
11. int
12. gcd(int m, int n)
13. {
14.         int ans;
15.
16.         if (m % n == 0)
17.                 ans = n;
18.         else
19.                 ans = gcd(n, m % n);
20.
21.         return (ans);
22. }
```

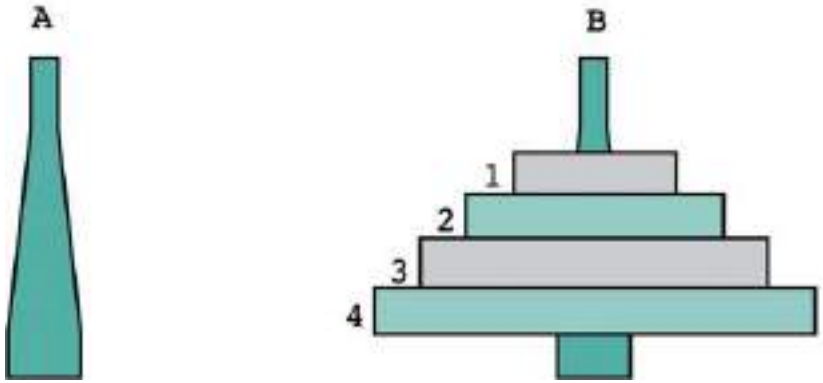
(continued)

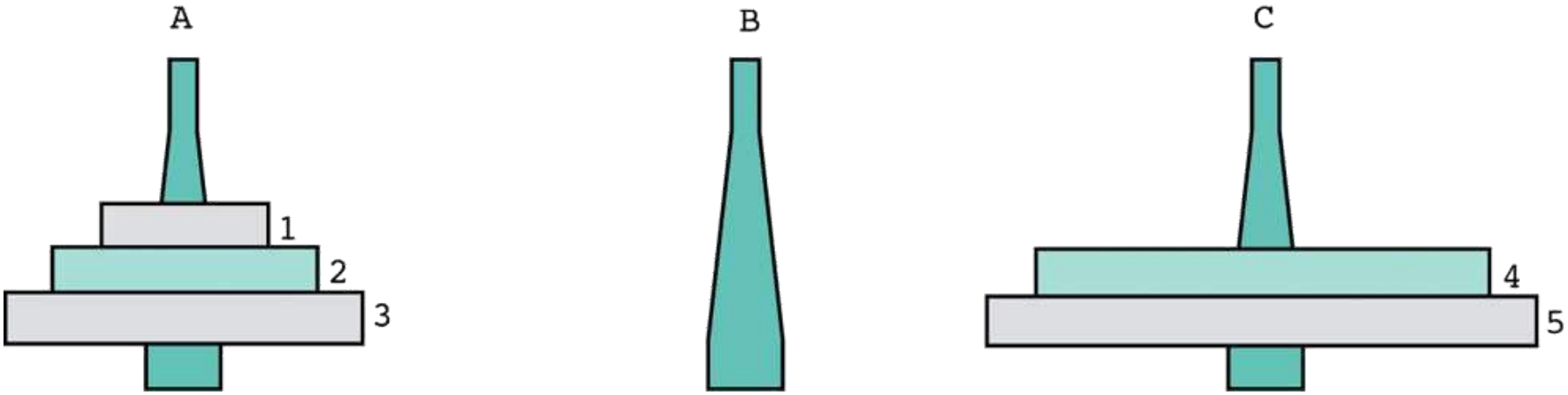
A Classical Case: Towers of Hanoi

- 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

```
1.  /*
2.   *  Displays instructions for moving n disks from from_peg to to_peg using
3.   *  aux_peg as an auxiliary.  Disks are numbered 1 to n (smallest to
4.   *  largest). Instructions call for moving one disk at a time and never
5.   *  require placing a larger disk on top of a smaller one.
6.   */
7.  void
8.  tower(char from_peg,     /* input - characters naming            */
9.        char to_peg,      /*         the problem's                */
10.       char aux_peg,     /*         three pegs                   */
11.       int n)            /* input - number of disks to
12. {
13.       if (n == 1) {
14.             printf("Move disk 1 from peg %c to peg %c\n
                                                  from_peg, to_peg);
15.       } else {
16.             tower(from_peg, aux_peg, to_peg, n - 1);
17.             printf("Move disk %d from peg %c to peg %c\n", n, from_peg, to_peg);
18.             tower(aux_peg, to_peg, from_peg, n - 1);
19.       }
20. }
```

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
```