Data Structures and Algorithms

Mr. Tahir Iqbal

tahir.iqbal@bahria.edu.pk

Lecture 09: Recursion

Introduction

- In C++, any function can call another function.
- A function can even call itself.
- When a function call itself, it is making a recursive call.
- **Recursive Call**
  - **A function call in which the function being called is the same as the one making the call**.
- Recursion is a powerful technique that can be used in the place of iteration(looping).
- **Recursion**
  - **Recursion is a programming technique in which procedures and functions call themselves.**

Recursive Algorithm

- **Definition**
  - An algorithm that calls itself
- **Approach**
  - Solve small problem directly
  - Simplify large problem into 1 or more smaller sub-problem(s) & solve recursively
  - Calculate solution from solution(s) for sub-problem

Recursive Definition

- **A definition in which something is defined in terms of smaller versions of itself.**
- To do recursion we should know the followings
  - **<u>Base Case:</u>**
    - The case for which the solution can be stated non-recursively
    - The case for which the answer is explicitly known.
  - **<u>General Case:</u>**
    - The case for which the solution is expressed in smaller version of itself. Also known as recursive case.

Example 1

- We can write a function called **power** that calculates the result of raising an integer to a positive power. If X is an integer and N is a positive integer, the formula for X^N is

$$X^N = \underbrace{X * X * X * X * X * \ldots * X}_{N-times}$$

- We can also write this formula as

$$X^N = X * \underbrace{X * X * X * X * \ldots * X}_{(N-1)\_times}$$

- Also as

$$X^N = X * X * \underbrace{X * X * X * \ldots . * X}_{(N-2)\_times}$$

- In fact we can write this formula as

$$X^N = X * X^{N-1}$$

Example 1(Recursive Power Function)

- Now lets suppose that X=3 and N=4
- Now we can simplify the above equation as $X^N = 3^4$

$$3^4 = 3 * 3^3$$

$$3^3 = 3 * 3^2$$

$$3^2 = 3 * 3^1$$

- So the base case in this equation is

$$3^1 = 3$$

```
int Power ( int   x , int   n )
{//recursive approach
   if ( n == 1 )
       return x;     //Base case
   else
       return x * Power (x, n-1);
                          // recursive call
}
int Power ( int   x , int   n )
{ //Iterative approach
   int power = 1;
   for (int i=1; i<=n; i++)
      power = power * x;
   return power;
}
```

```
int Power ( int   x , int   n )
{//recursive approach
        if ( n == 1 )
             return x;     //Base case
        else
          return x * Power (x, n-1);
                         // recursive call
}
```

How Recursion Works

To truly understand how recursion works we need to first explore how any function call works.

When a program calls a subroutine (function) the current function must suspend its processing.

The called function then takes over control of the program. When the function is finished, it needs to return to the function that called it.

The calling function then �much‘wakes up’ and continues processing.

How Recursion Works

One important point in this interaction is that, unless changed through call-by-reference, all local data in the calling module must remain unchanged.

Therefore, when a function is called, some information needs to be saved in order to return the calling module back to its original state (i.e., the state it was in before the call).

We need to save information such as the local variables and the spot in the code to return to after the called function is finished.

How Recursion Works

To do this we use a stack.

Before a function is called, all relevant data is stored in a ***stackframe***.

This stack-frame is then pushed onto the system stack.

After the called function is finished, it simply pops the system stack to return to the original state. By using a stack, we can have functions call other functions which can call other functions, etc.

Because the stack is a first-in, last-out data structure, as the stack-frames are popped, the data comes out in the correct order.

Recursion

Recursion can be seen as building objects from objects that have set definitions. Recursion can also be seen in the opposite direction as objects that are defined from smaller and smaller parts. “Recursion is a different concept of circularity.”(Dr. Britt, Computing Concepts Magazine, March 97, pg.78)

Iteration vs. recursion

- Some things (e.g. reading from a file) are easier to implement iteratively
- Other things (e.g. mergesort) are easier to implement recursively
- Others are just as easy both ways
- When there is no real benefit to the programmer to choose recursion, iteration is the more efficient choice
- It can be proved that two methods performing the same task, one implementing an iteration algorithm and one implementing a recursive version, are equivalent

Example 2 (Recursive Factorial Function)

Factorial Function:

Given a +ive integer n, n factorial is defined as the product of all integers between 1 and n, including n.

So we can write factorial function mathematically as

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

Recursion

Recursion is a powerful problem-solving technique that often produces very clean solutions to even the most complex problems.

Recursive solutions can be easier to understand and to describe than iterative solutions.

Limitations of Recursion

By using recursion, you can often write simple, short implementations of your solution.

However, just because an algorithm *can* be implemented in a recursive manner doesn’t mean that it *should* be implemented in a recursive manner.

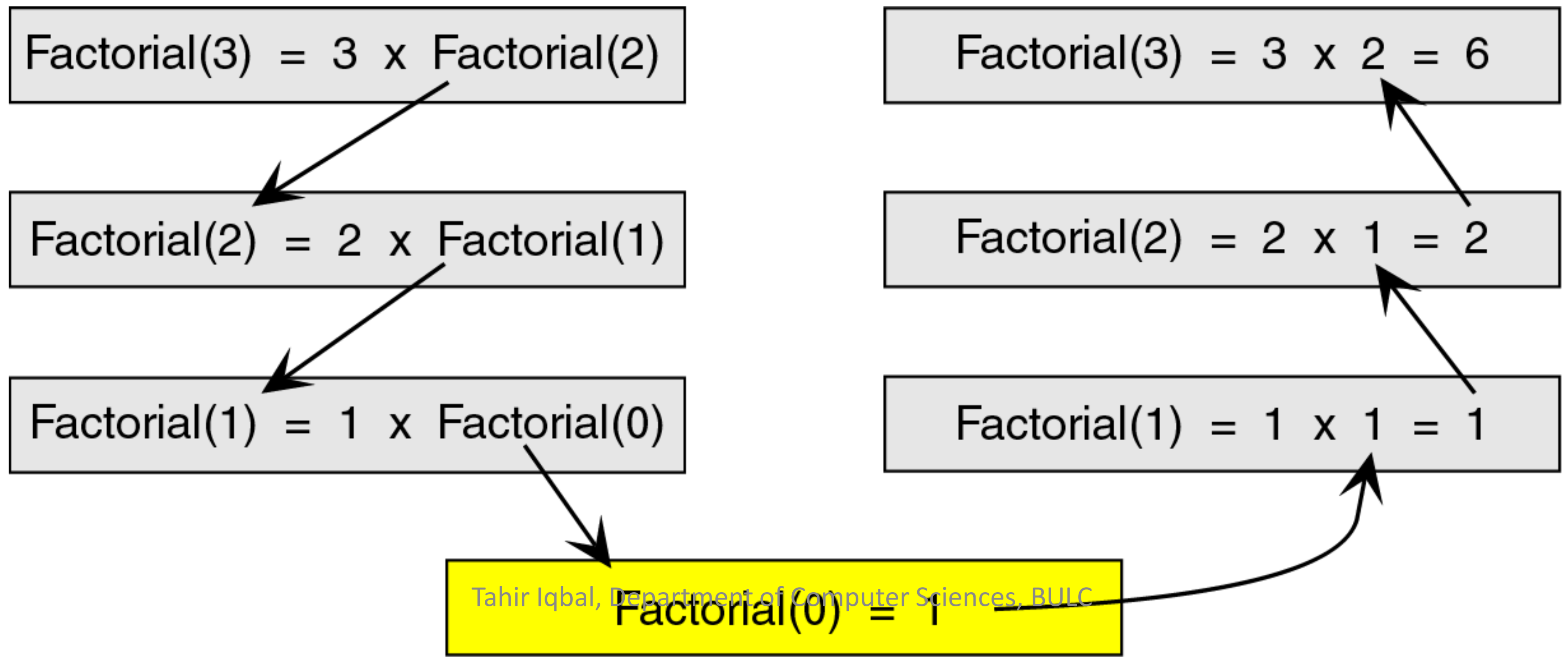
Recursive solutions may involve extensive overhead because they use calls.

When a call is made, it takes time to build a stackframe and push it onto the system stack.

Conversely, when a return is executed, the stackframe must be popped from the stack and the local variables reset to their previous values – this also takes time.

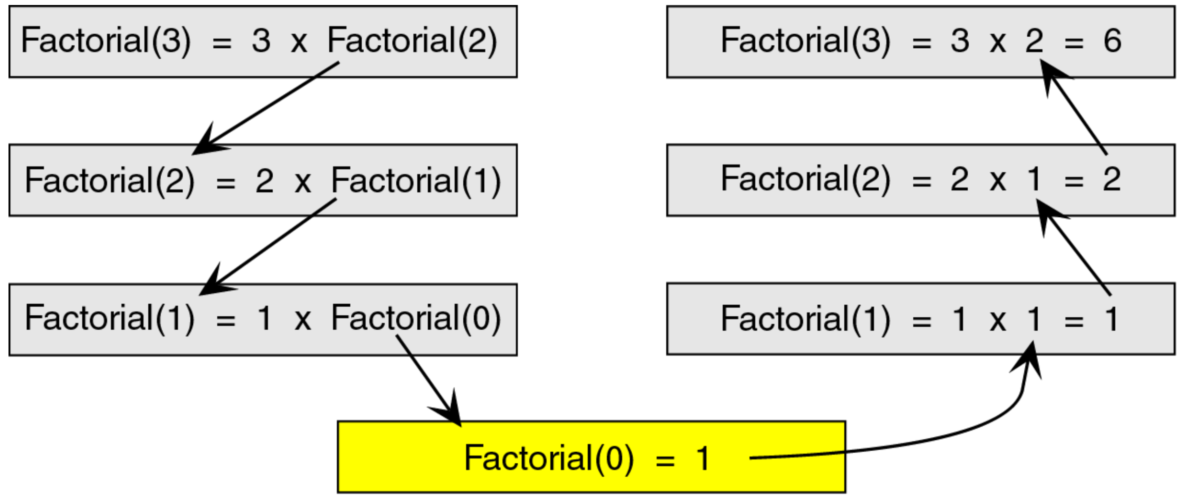
Factorial - Recursion

To see how the recursion works, let’s break down the factorial function to solve factorial(3)



Factorial(3) = 3 x Factorial(2)
Factorial(2) = 2 x Factorial(1)
Factorial(1) = 1 x Factorial(0)
Factorial(0) = 1
Factorial(3) = 3 x 2 = 6
Factorial(2) = 2 x 1 = 2
Factorial(1) = 1 x 1 = 1

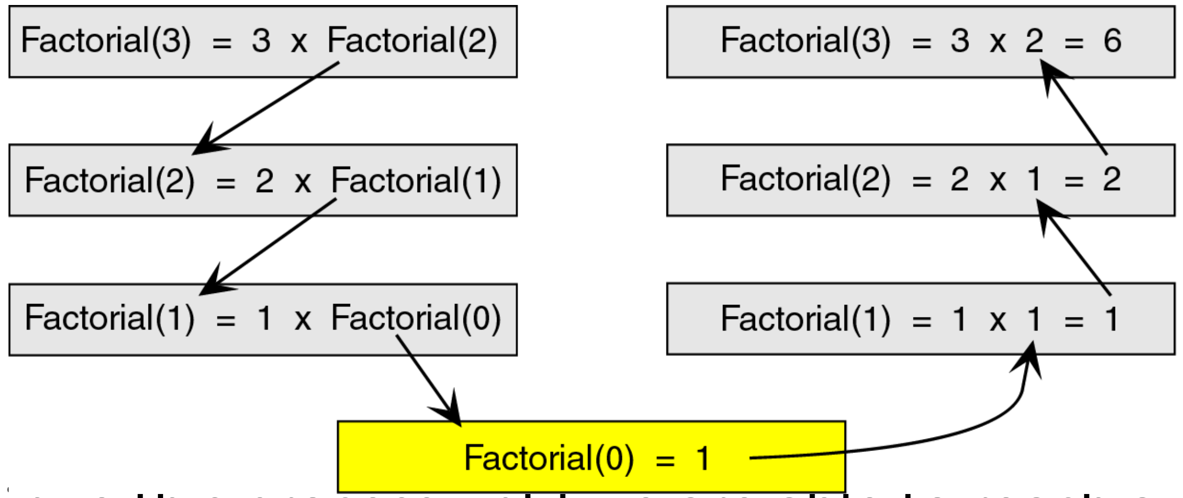
Factorial - Breakdown


Factorial(3) = 3 x Factorial(2)
Factorial(2) = 2 x Factorial(1)
Factorial(1) = 1 x Factorial(0)
Factorial(0) = 1
Factorial(3) = 3 x 2 = 6
Factorial(2) = 2 x 1 = 2
Factorial(1) = 1 x 1 = 1

Here, we see that we start at the top level, factorial(3), and simplify the problem into
3 x factorial(2).

Now, we have a slightly less complicated problem in factorial(2), and we simplify this problem into 2 x factorial(1).

Breakdown


Factorial(3) = 3 x Factorial(2)
Factorial(2) = 2 x Factorial(1)
Factorial(1) = 1 x Factorial(0)
Factorial(0) = 1
Factorial(3) = 3 x 2 = 6
Factorial(2) = 2 x 1 = 2
Factorial(1) = 1 x 1 = 1

We continue this process until we are able to reach a problem that has a known solution.

In this case, that known solution is factorial(0) = 1 (BASE CASE).

The functions then return in reverse order to complete the solution.

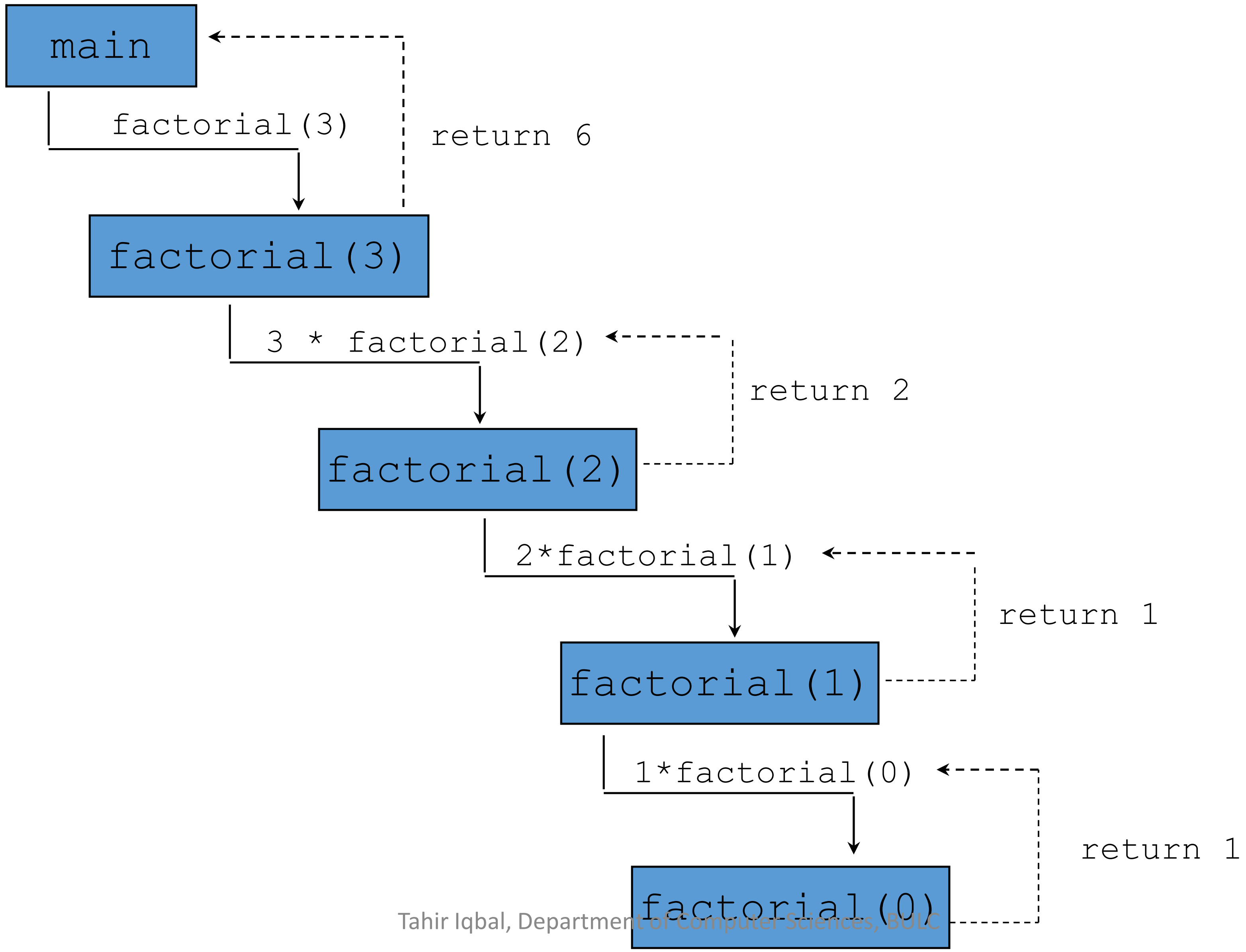
Example 2 (Recursive Factorial Function)

```
int factorial(n)
  {
        if(n == 0)                              //Base Case
          return 1;
        else
          return  n * factorial(n-1);          //Recursion
  }
```

Evaluation of Factorial Example

```
To evaluate Factorial(3)
evaluate 3 * Factorial(2)
        To evaluate Factorial(2)
        evaluate 2 * Factorial(1)
                To evaluate Factorial(1)
                evaluate 1 * Factorial(0)
                        Factorial(0) is 1
                        Return 1
                Evaluate 1 * 1
                Return 1
        Evaluate 2 * 1
        Return 2
Evaluate 3 * 2
Return 6
```

Recursive Programming



Rules For Recursive Function

1. In recursion, it is essential for a function to call itself, otherwise recursion will not take place.
2. To stop the recursive function it is necessary to base the recursion on test condition and proper terminating statement such as *exit()* or *return* must be written using if() statement.
3. When a recursive function is executed, the recursive calls are not implemented instantly. All the recursive calls are pushed onto the stack until the terminating condition is not detected, the recursive calls stored in the stack are popped and executed.
4. During recursion, at each recursive call new memory is allocated to all the local variables of the recursive functions with the same name.

The Runtime Stack during Recursion

- To understand how recursion works at run time, we need to understand what happens when a function is called.
- Whenever a function is called, a block of memory is allocated to it in a run-time structure called the **stack.**
- This block of memory will contain
    - the function's **local variables,**
    - local copies of the function's **call-by-value parameters,**
    - pointers to its **call-by-reference parameters**, and
    - a **return address,** in other words where in the program the function was called from. When the function finishes, the program will continue to execute from that point.

Exercise

1. The problem of computing the sum of all the numbers between 1 and any positive integer N can be recursively defined as:

$$\sum_{i=1}^{N} = N + \sum_{i=1}^{N-1} = N + (N-1) + \sum_{i=1}^{N-2}$$

$= \quad$ etc.

Exercise

```
int sum(int n)
{
if(n==1)
  return n;
else
  return n + sum(n-1);
}
```

Reversing List - Iterative

```
void reverse(Node* first) {
    if (first==NULL) return;
    Node* next =NULL;            Node* prev = NULL;
    Node* curr = first;
    while (curr != NULL) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
     }
     first = prev;
}
```

Reversing List - Recursive

```
node* Reverse(node* temp)
{
if(temp->next == NULL)
                                        {
            first = temp;
            return temp;
        }
        Reverse(temp->next)->next = temp;
        return temp;
}
```

Tail Recursion

- A recursive function is called **tail recursive** if only one recursive call appears in the function and that recursive call is the last statement in the function body.
- This approach provides decreased stack implementation time and gives faster execution.
- Smart compilers can detect tail recursion and convert it to iterative to optimize code

```
node* find_value(node *temp, int value)
{
    if  (temp == NULL)
        return NULL;
   if  (temp->data == value)
        return temp;
   return find_value(temp->next, value);
 }
```

Indirect Recursion

A function is *indirectly* recursive if it contains a call to another function which ultimately calls originally calling function.

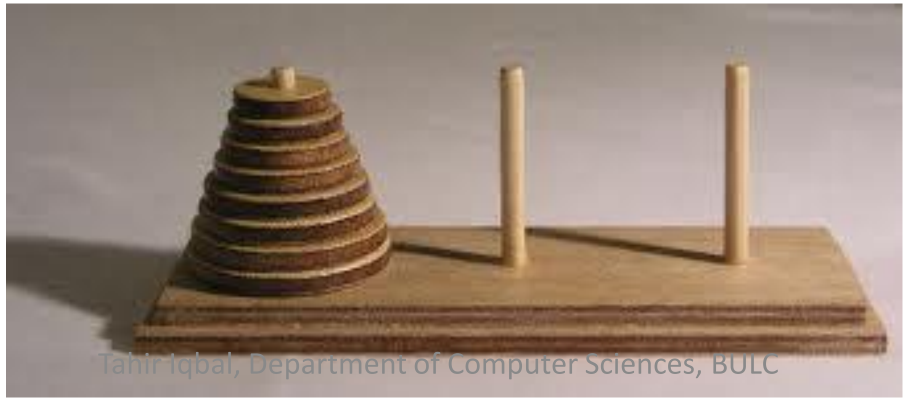
The following pair of functions is indirectly recursive. Since they call each other, they are also known as *mutually recursive* functions.

```
int foo(int x)
 {    if (x <= 0) return x;
       return bar(x);
 }
int bar(int y)
 {    return foo(y - 1);
 }
```

Tower of Hanoi

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.



Tower of Hanoi

```
void move(int n,char *s,char *i,char *d)
 // s stands for source tower
 // d stands for destination tower
 // i stands for intermediate tower
 {
     if(n>0)
     {
          move(n-1,s,d,i);
          // move n-1 disks from source to intermediate tower
         cout<<" disk "<<n<<" is moved from "<<s<<" to "<<d<<endl;
         // move the disk from to source to destination
          move(n-1,i,s,d);
         // move n-1 disks from intermediate to destination
     }
 }
```

Conclusion

A recursive solution solves a problem by solving a smaller instance of the same problem.

It solves this new problem by solving an even smaller instance of the same problem.

Eventually, the new problem will be so small that its solution will be either obvious or known.

This solution will lead to the solution of the original problem.