Recursion

SDC, OSW CSE 3541

Department of Computer Science & Engineering
ITER, Siksha 'O' Anusandhan Deemed To Be University
Jagamohan Nagar, Jagamara, Bhubaneswar, Odisha - 751030

Text Book(s)

Jeri R. Hanly & Elliot B. Koffman

Problem Solving and Program Design in C

Seventh Edition, Pearson Education

Robert Love

LINUX System Programming

Second Edition, SPD, O'REILLY

Contents

Introduction

- When a function calls itself directly or indirectly its known as recursion.
- **Recursive function:** function that calls itself or that is part of a cycle in the sequence of function calls.
  - A function **`f1`** is also recursive if it calls a function **`f2`** , which under some circumstances calls **`f1`** , creating a cycle in the sequence of calls.
  - The ability to invoke itself enables a recursive function to be repeated with different parameter values.
- Recursion can be used as an alternative to iteration (looping).
- Generally, a recursive solution is less efficient than an iterative solution in terms of computer time due to the overhead for the extra function calls.
- However, in many instances, the use of recursion enables us to specify a very natural, simple solution to a problem that would otherwise be very difficult to solve.

The Nature of Recursion

Problems that lend themselves to a recursive solution have the following characteristics:

- One or more **simple cases** (case for which a straightforward solution is known) of the problem have a straightforward, nonrecursive solution.
- The other cases can be **redefined** in terms of problems that are closer to the simple cases.
- By applying this **redefinition** process every time the recursive function is called, eventually the problem is reduced entirely to simple cases, which are relatively easy to solve.

The recursive algorithms that we write will generally consist of an **if** statement with the following form:

```
if this is a simple case
     solve it
else
     redefine the problem using recursion
```

Recursive Approach

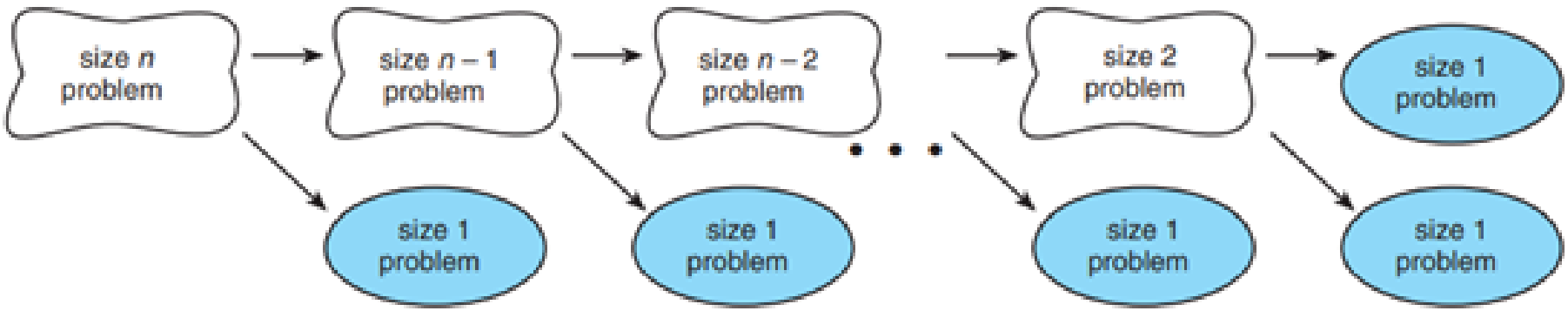


Figure 1: Splitting Problem into Smaller Problems

- The given problem is divided into various sub-problems, the consecutive sub problems are solved to obtain the solution of the given problem.
- Let's assume that for a particular problem of size n, we can split the problem into a problem of size 1, which we can solve (a simple case), and a problem of size n-1.
- We can split the problem of size n-1 into another problem of size 1 and a problem of size n-2, which we can split further.
- If we split the problem n-1 times, we will end up with n problems of size 1, all of which we can solve.

Essential Elements of Recursion

Two most essential elements of the recursion are base case and recursive statement.

- **Base Case/ Simple case** is a case where terminating condition becomes true, terminating condition is a condition that indicates the termination of the recursive function and prevent the recursion from the infinite loop.

```
To compute m*n;
-------------------
Base case is reached when the condition n == 1 is true.
ans = m;    /* simple case */
```

- **Recursive Statement** is a statement that holds the call to corresponding function, which will get called recursively every time function gets executed.

```
If n is greater than 1, the statement

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

Design of Recursive Function Multiply

Performs integer multiplication using + operator.
Pre: m and n are defined and n $>$ 0
Post: returns m $\times$ 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);
}
```

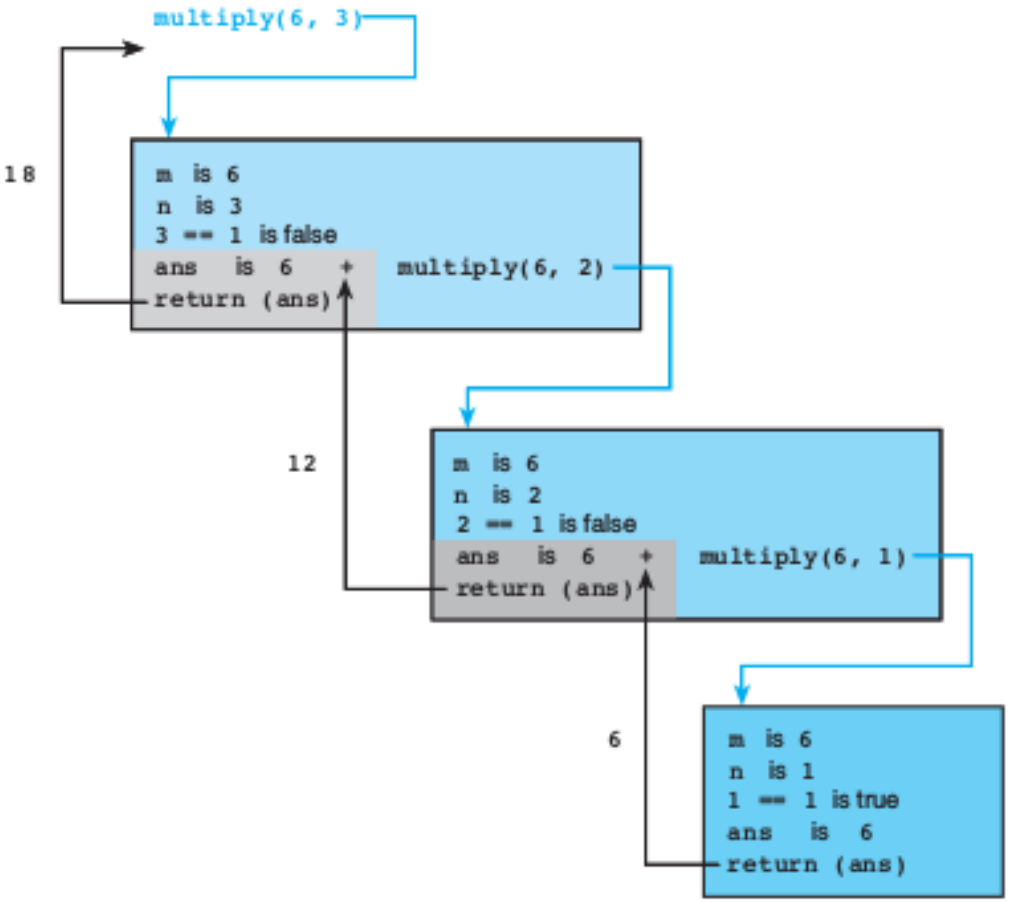
Here, splitting the original problem into the two simpler problems:

- multiply **m** by **n-1**
- add **m** to the result

Tracing Recursive Function

1. Tracing a recursive Function that returns a value
2. Tracing a void function that is recursive

Tracing the recursive function, `multiply`, that returns a value



- **Activation frame:** representation of one call to a function.
- Left figure shows an activation frame corresponding to each call of the function, **`multiply(6,3)`**.
- An activation frame shows the parameter values for each call and summarizes the execution of the call.
- Left figure shows three calls to function **`multiply`**. Parameter **`m`** has the value **`6`** for all three calls; parameter **`n`** has the values **`3`**, **`2`** and finally, **`1`**.
- Here, **three** activation frames generated to solve the problem of multiplying 6 by 3.

Recursive Function Returning `void`

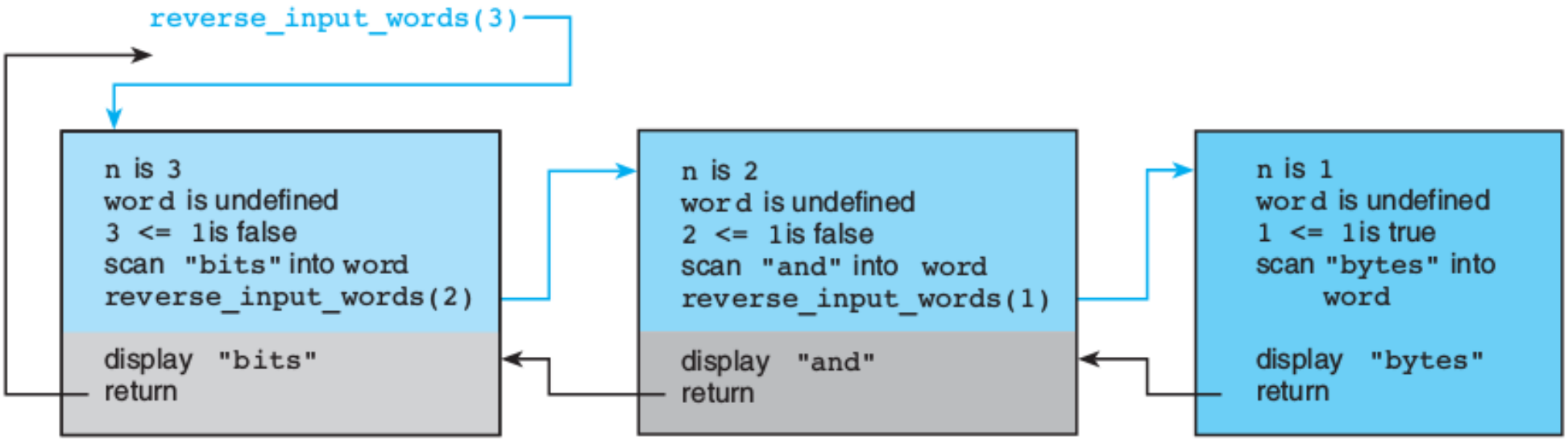
Function, **`reverse_input_words`**, takes n words as input and print them in reverse order on separate lines, where n > 0.

```
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);
  }
}
```

Input
bits
and
bytes

Output in Reverse Order
bytes
and
bits

Tracing a `void` Function That Is Recursive



- **Terminating condition:** a condition that is true when a recursive algorithm is processing a simple case/ base case.
- The body of the function **`reverse_input_words`** consists of an **`if`** statement that evaluates a terminating condition, $\mathbf{n} \leq \mathbf{1}$ like most recursive modules.
- When the terminating condition is true, the function is dealing with one of the problem's simple cases - printing in reverse order a list of just one word.
- If the terminating condition is false ($\mathbf{n} > \mathbf{1}$), the recursive step (following **`else`**) is executed.
- Each **activation frame** begins with a list of the initial values of **`n`** and **`word`** for that frame.
- The value of **`n`** is passed into the function when it is called; the value of the local variable **`word`** is initially undefined.

Parameter and Local Variable Stack

- **Stack:** a data structure in which the last data item added is the first data item processed.
- C uses the stack data structure keeps track of the values of parameters and local variables.
- In this data structure, we add data items (the push operation) and remove them (the pop operation) from the same end of the list, so the last item stored is the first processed.
- We can implement and manipulate stack data structure ourself using arrays or linked lists.
- However, C automatically handles all the stack manipulation associated with function calls, so we can write recursive functions without needing to worry about the stacks.

System Stack

Area of memory where parameters and local variables are allocated when a function is called and deallocated when the function returns.

When and How to Trace Recursive Functions

- Doing a trace by hand of multiple calls to a recursive function is helpful in understanding how recursion works but less useful when trying to develop a recursive algorithm.
- During algorithm development, it is best to trace a specific case simply by trusting any recursive call to return a correct value based on the function purpose.
- Then the hand trace can check whether this value is manipulated properly to produce a correct function result for the case under consideration.
- However, if a recursive function's implementation is flawed, tracing its execution is an essential part of identifying the error.
- The recursive function can be made to trace itself by inserting debugging print statements showing entry to and exit from the function.

Recursive Function multiply with Print Statements to Create Trace and Output from multiply(8, 3)

```
/* Includes calls to printf to trace execution */
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 */
  printf("multiply(%d, %d) returning %d\n", m, n, ans);
  return (ans);}
```

Output on Recursive Function Tracing

```
Entering multiply with m= 8, n = 3
Entering multiply with m= 8, n = 2
Entering multiply with m= 8, n = 1
multiply(8, 1) returning 8
multiply(8, 2) returning 16
multiply(8, 3) returning 24
```

Recursive Mathematical Functions

Many mathematical functions can be defined recursively. An example is the factorial of a number n(n!)

- 0! is 1
- n! is n $\times$ (n - 1)!, for n > 0

Recursive Function factorial

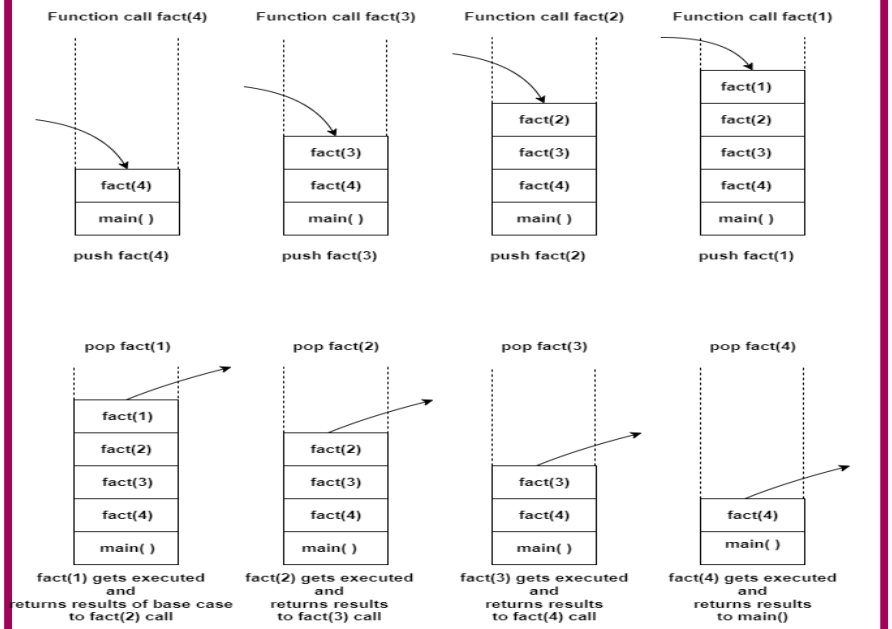
```
/* Compute n! using a recursive
definition. Pre: n >= 0 */
int factorial(int n){
  int ans;
  if (n == 0)
      ans = 1;
  else
     ans = n * factorial(n - 1);
  /* Return function result */
  return (ans);
}
```

Iterative Function factorial

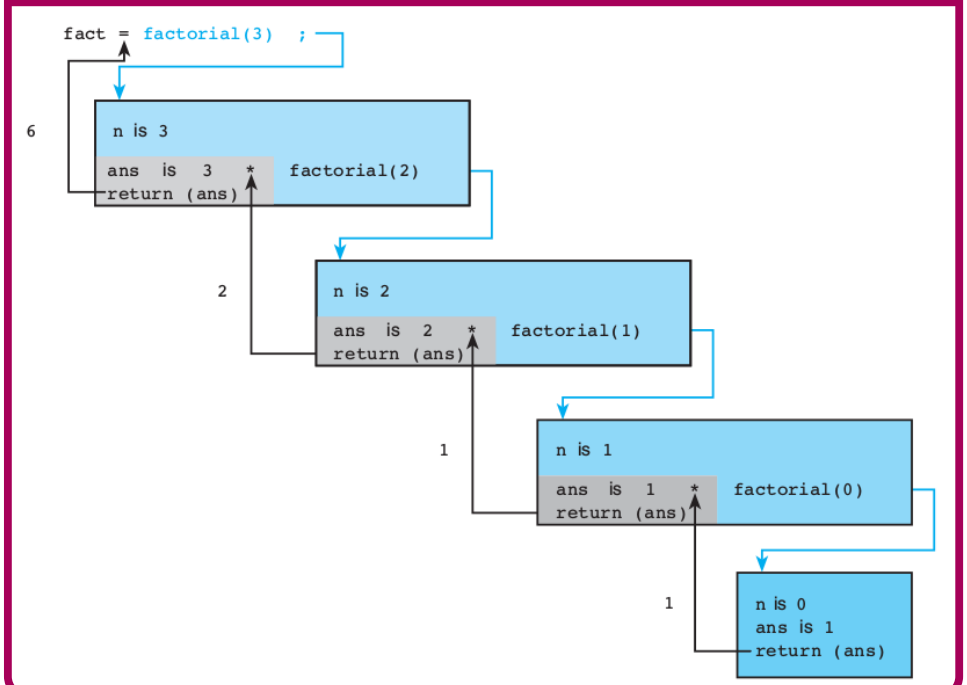
```
/* Compute n!, Pre: n >= 0 */
int factorial(int n){
 int i,product = 1;// local vars
/* Compute the product:
nx(n-1)x(n-2)x...x 2 x 1 */
for (i = n; i > 1; --i) {
 product = product * i;
}
/* Return function result */
return (product);
}
```

Usage of Stack and Tracing of Recursion for `factorial` Function

Recursive Factorial Function



Tracing of factorial(3)



Types of Recursion

There's mainly two types of recursion.

❶ Direct recursion

If the recursive statement is in the function itself then it's known as direct recursion.

❷ Indirect recursion

If the recursive statement is in another function or we can say if the call is made by another function its known as indirect recursion call.

Direct Recursion & Indirect Recursion

❶ Direct recursion

```
int add(int n){
  int ans;
  if (n==1)
     return n;
  else{
     ans=n+add(n-1);
     return ans;
  }
}
void main(){
     printf("%d", add(5));
}
```

❷ Indirect recursion

```
void odd(); void even();
int num = 1;
void odd(){
 if (num <= 10){
   printf (" %d ", num + 1);
   num++;
   even();
 }
 return;
}
void even ()  {
 if(num <= 10)  {
   printf (" %d ", num - 1);
   num++;
   odd();
 }
 return;
}
int main(){
 odd();
 return 0;}
```

A Critique of Recursion vs Iteration

Recursion

- When a function calls itself directly or indirectly.
- Implemented using function calls.
- Base case and recursive relation are specified.
- The function state approaches the base case.
- Uses stack memory to store local variables and parameters.
- It will cause stack overflow error and may crash the system if the base case is not defined or is never reached.

Iteration

- When some set of instructions are executed repeatedly.
- Implemented using loops.
- Includes initializing control variable, termination condition, and update of the control variable.
- The control variable approaches the termination value.
- Does not use memory except initializing control variables.
- It will cause an infinite loop if the control variable does not reach the termination value.

- Generally, a recursive solution is less efficient than an iterative solution in terms of computer time due to the overhead for the extra function calls
- However, in many instances, the use of recursion enables us to specify a very natural, simple solution to a problem that would otherwise be very difficult to solve. For this reason, recursion is an important and powerful tool in problem solving and programming.

Case Study: Tower of Hanoi

Algorithm:

Step-❶ if n is 1 then

Step-❷ move disk 1 from_peg to to_peg

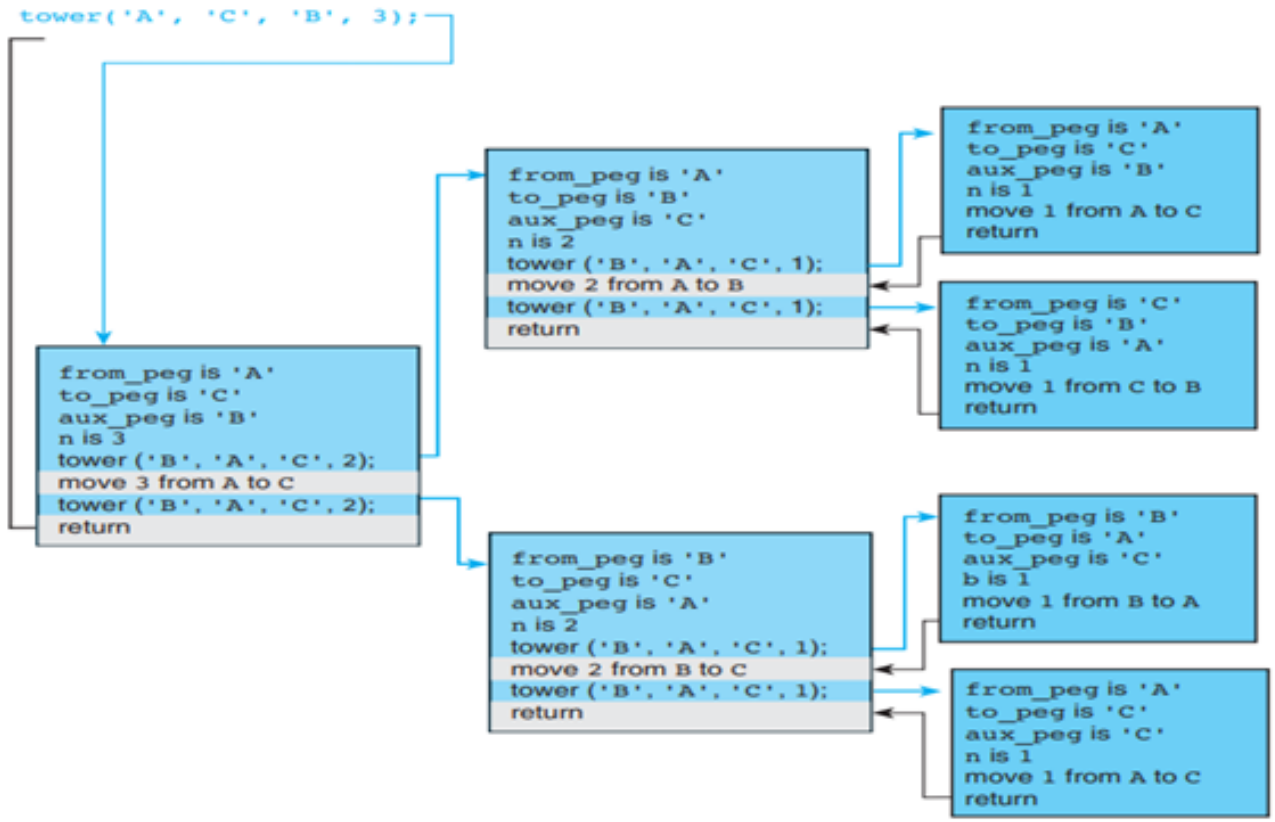
Step-❸ else move n-1 disk from_peg to auxiliary_peg using the topeg.

Step-❹ move disk n from the from_peg to the to_peg.

Step-❺ move n-1 disks from the auxiliary_peg to the to_peg using the from_peg.

```
//Recursive function:
void tower(char from_peg, char to_peg, char aux_peg, int n) {
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); }
}
```

Recursive Call Tracing Through Activation Frame



Output:

```
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
```

Common Programming Errors

- If recursive statement has no base case, recursion may result into infinite loop of function calls till the whole memory is used.
- A run time error message noting stack overflow or an action violation is an indicator that a recursive function is not terminating.
- The recopying of large arrays or other data structures can quickly consume all available memory. Such copying should be done inside a recursive function only.
- On most systems, pressing a control character sequence (e.g., Control S) will temporarily stop output to the screen.

Try Yourself

1. Write a program to find Fibonacci series using recursive function.
2. Write a program for recursive selection sort.
3. Write a program to find GCD of two integers using recursive function.
4. Write a program for counting the occurrences of a character in a string.
5. Write a program to find capital letters in a string using recursive function.
6. Write a recursive C function that accumulates the sum of the values in an n-element array.

Exercise

1. What problem do you notice in the following recursive function? Show two possible ways to correct the problem.

```
int silly(int n){
  if(n <= 0)
     return (1);
  else if(n%2 == 0)
     return (n);
   else
     silly(n - 3);
}
```

2. Guess the output of the code snippet.

```
int function(int n);
 int main(){
 int n=10;
 printf("%d\n",function(n);
 return 0;
}
```

```
int function(int n){
 if(n>0)
    return(n+function(n-1));
}
```

3. Express each of the following algebraic formulas in a recursive form.

$$y = (x_1 + x_2 + x_3 + \cdots + x_n)$$

THANK YOU