Recursion
Recursion
Overview

14.1  Recursive Functions for Tasks
14.2  Recursive Functions for Values
14.3  Thinking Recursively
What is recursion?

- When a function is defined in terms of itself
- A recursive function is a function that calls itself

```c
void foo(int n) {
    return foo(n-1);
}
```
Objectives

- To know what is a recursive function and the benefits of using recursive
- To determine the base cases in a recursive function
- To understand how recursive function calls are handled in a call stack
- To solve problems using recursion
- To use an overloaded helper function to derive a recursive function
- To understand the relationship and difference between recursion and iteration
Factorial Function

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

int factorial(int n) {
    if (n == 0) { // Base Case
        return 1;
    } else {
        return n * factorial(n - 1); // Recursive call
    }
}
Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(3)
Computing Factorial

factorial(3) = 3 * factorial(2)

factorial(0) = 1;
factorial(n) = n*factorial(n-1);
Computing Factorial

\[
\text{factorial}(3) = 3 \times \text{factorial}(2) \\
= 3 \times (2 \times \text{factorial}(1))
\]

\[
\text{factorial}(0) = 1; \\
\text{factorial}(n) = n \times \text{factorial}(n-1);
\]
Computing Factorial

\[
\text{factorial}(3) = 3 \times \text{factorial}(2)
\]
\[
= 3 \times (2 \times \text{factorial}(1))
\]
\[
= 3 \times (2 \times (1 \times \text{factorial}(0)))
\]

\[
\text{factorial}(0) = 1;
\]
\[
\text{factorial}(n) = n \times \text{factorial}(n-1);
\]
Computing Factorial

factorial(3) = 3 * factorial(2)
= 3 * (2 * factorial(1))
= 3 * ( 2 * (1 * factorial(0)))
= 3 * ( 2 * ( 1 * 1)))

factorial(0) = 1;
factorial(n) = n*factorial(n-1);
Computing Factorial

factorial(3) = 3 * factorial(2)
  = 3 * (2 * factorial(1))
  = 3 * ( 2 * (1 * factorial(0)))
  = 3 * ( 2 * ( 1 * 1)))
  = 3 * ( 2 * 1)

factorial(0) = 1;
factorial(n) = n*factorial(n-1);
Computing Factorial

\[
\text{factorial}(3) = 3 \times \text{factorial}(2) \\
= 3 \times (2 \times \text{factorial}(1)) \\
= 3 \times (2 \times (1 \times \text{factorial}(0))) \\
= 3 \times (2 \times (1 \times 1)) \\
= 3 \times (2 \times 1) \\
= 3 \times 2
\]

factorial(0) = 1;
factorial(n) = n \times \text{factorial}(n-1);
factorial(3) = 3 * factorial(2)  
= 3 * (2 * factorial(1))  
= 3 * ( 2 * (1 * factorial(0)))  
= 3 * ( 2 * ( 1 * 1)))  
= 3 * ( 2 * 1)  
= 3 * 2  
= 6
Trace Recursive factorial

factorial(4)

Executes factorial(4)

Main method

3

Stack

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(4)

5

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(1)

Space Required for factorial(0)
Note the change in the function stack. Each call to factorial pushes the function onto the stack.
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)
return 4 * factorial(3)

Step 1: executes factorial(3)
return 3 * factorial(2)

Step 2: executes factorial(2)
return 2 * factorial(1)

Step 3: executes factorial(1)
return 1 * factorial(0)

Step 4: executess factorial(0)
return 1

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Space Required for factorial(4)
Space Required for factorial(3)
Space Required for factorial(2)
Space Required for factorial(1)

Stack

Main method
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)
return 4 * factorial(3)

Step 1: executes factorial(3)
return 3 * factorial(2)

Step 2: executes factorial(2)
return 2 * factorial(1)

Step 3: executes factorial(1)
return 1 *

Executes factorial(1)

Step 4: executes factorial(0)
return 1

Step 5: return 1
Step 6: return 1
Step 7: return 2
Step 8: return 6

Space Required for factorial(4)

Space Required for factorial(3)

Space Required for factorial(2)

Stack

Main method
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

return 1

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Main method

3

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(1)

Space Required for factorial(0)

Stack

Main method

Execution factorial(0)
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)
return 4 * factorial(3)

Step 1: executes factorial(3)
return 3 * factorial(2)

Step 2: executes factorial(2)
return 2 * factorial(1)

Step 3: executes factorial(1)
return 1 * factorial(0)

Step 4: executes factorial(0)
return 1

Step 5: return 1
Step 6: return 1
Step 7: return 2
Step 8: return 6
Step 4: executes factorial(0) returns 1

Space Required for factorial(3)
Space Required for factorial(2)
Space Required for factorial(4)

Stack

Main method
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)
return 4 * factorial(3)

Step 1: executes factorial(3)
return 3 * factorial(2)

Step 2: executes factorial(2)
return 2 * factorial(1)

Step 3: executes factorial(1)
return 1 * factorial(0)

Step 4: executes factorial(0)
returns factorial(0)

Step 5: return 1

Space Required for factorial(3)
Space Required for factorial(2)
Space Required for factorial(1)
Space Required for factorial(0)

Stack
Main method
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

return 1

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Step 9: return 24

Main method

3

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(1)

Space Required for factorial(0)

Space Required for factorial(4)

Stack

Main method
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

return 1

Step 5: return 1

Step 6: return 1

Step 7: return 2

return factorial(2)

Main method

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(4)

Stack

Main method

3

Space Required for factorial(3)

Space Required for factorial(2)

4
Trace Recursive factorial animation

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

return 1

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Main method

Space Required
for factorial(3)

Space Required
for factorial(2)

Space Required
for factorial(1)

Space Required
for factorial(4)

Stack

Return factorial(3)

Animation

Trace Recursive factorial
Trace Recursive factorial

Main method

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(4)

Stack

Step 9: return 24

Step 8: return 6

Step 7: return 2

Step 6: return 1

Step 5: return 1

Step 4: executes factorial(0)

return 1

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 0: executes factorial(4)

return 4 * factorial(3)

return 1 * ... factorial(1)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

returns factorial(4)
factorial(4) Stack Trace
Fibonacci Numbers

Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89...
indices: 0 1 2 3 4 5 6 7 8 9 10 11

fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index -1) + fib(index -2); assume index >=2

\[
\begin{align*}
\text{fib(3)} &= \text{fib(2)} + \text{fib(1)} = (\text{fib(1)} + \text{fib(0)}) + \text{fib(1)} = (1 + 0) \\
&\quad + \text{fib(1)} = 1 + \text{fib(1)} = 1 + 1 = 2
\end{align*}
\]
Fibonacci Numbers, cont.

```
return fibonacci(3)
  return fibonacci(2) + fibonacci(1)
    return fibonacci(1) + fibonacci(0)
      return 1
    return 0
        return 1
```
Characteristics of Recursion

All recursive methods have the following characteristics:

- One or more base cases (*the simplest case*) are used to stop recursion. We know how to solve this case.
- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

In general, to solve a problem using recursion, you break it into subproblems. If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively. This subproblem is almost the same as the original problem in nature with a smaller size.
Problem Solving Using Recursion

Let us consider a simple problem of printing a message for $n$ times. You can break the problem into two subproblems:

1. print the message one time
2. print the message for $n-1$ times.

*The second problem is the same as the original problem with a smaller size. The base case for the problem is $n==0$.*/
Print Message Recursively

```cpp
void printMsg(char *aMessage, int aTimes) {
    if(aTimes >= 1) {
        cout << aMessage << aTimes << endl;
        printMsg(aMessage, aTimes-1);
    }
}
```
Think Recursively

Many of the problems presented in the early chapters can be solved using recursion if you *think recursively*. For example, the palindrome problem can be solved recursively as follows:

```c
bool isPalindrome(const char* const aString) {
    if(strlen(aString) <= 1) {
        return true;
    } else if(aString[0] != aString[strlen(aString)-1]) {
        return false;
    } else {
        return isPalindrome(substring(aString, 1, strlen(aString)-2));
    }
}
```
Recursive Selection Sort

1. Find the largest number in the list and swaps it with the last number.
2. Ignore the last number and sort the remaining ints in the list recursively.
Towers of Hanoi

- There are \( n \) disks labeled 1, 2, 3, \ldots, \( n \), and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.
Towers of Hanoi, cont.

Original position

Step 1: Move disk 1 from A to B

Step 2: Move disk 2 from A to C

Step 3: Move disk 1 from B to C

Step 4: Move disk 3 from A to B

Step 5: Move disk 1 from C to A

Step 6: Move disk 2 from C to B

Step 7: Move disk 1 from A to B
Solution to Towers of Hanoi

The Towers of Hanoi problem can be decomposed into three subproblems.
Solution to Towers of Hanoi

- Move the first $n-1$ disks from A to C with the assistance of tower B.
- Move disk $n$ from A to B.
- Move $n-1$ disks from C to B with the assistance of tower A.