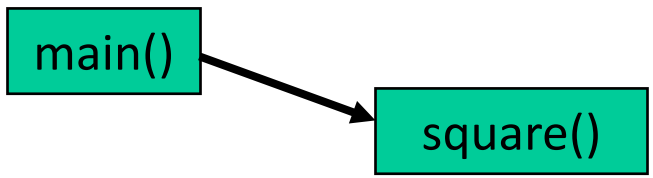
CIS 351-Data Structure-Recursion
Mar 31, 2020

Dr. Farzana Rahman

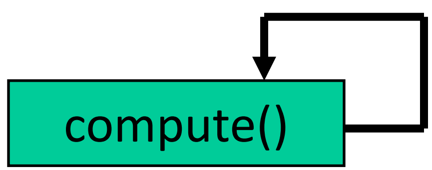
Syracuse University

Introduction to Recursion

- So far, we have seen methods that call other functions.
  - For example, the `main()` method calls the square`()` function.



- Recursive Method:
  - A recursive method is a method that calls itself.



Why use Recursive Methods?

- In computer science, some problems are more easily solved by using recursive functions.
- If you go on to take a computer science algorithms course, you will see lots of examples of this.
- **For example:**
  - Traversing through a directory or file system.
  - Traversing through a tree of search results.
- For today, we will focus on the basic structure of using recursive methods.

World's Simplest Recursion Program

```
public class Recursion
{
  public static void main (String args[])
  {
     count(0);
     System.out.println();
  }

  public static void count (int index)
  {
     System.out.print(index);
     if (index < 2)
          count(index+1);
  }
}
```

This program simply counts from 0-2:

012

This is where the recursion occurs.

You can see that the count() function

calls itself.

Visualizing Recursion

- To understand how recursion works, it helps to visualize what's going on.
- To help visualize, we will use a common concept of *Stack*.
- A stack basically operates like a container of trays in a cafeteria. It has only two operations:
  - Push: you can push something onto the stack.
  - Pop: you can pop something off the top of the stack.
- Let's see an example stack in action.

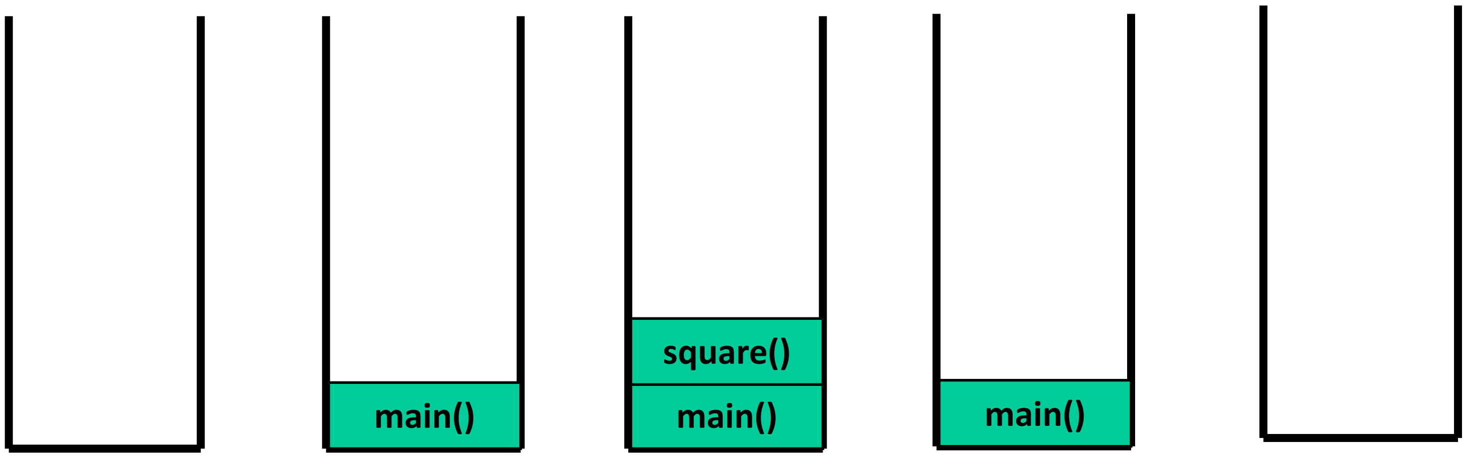
Stacks and Methods

- When you run a program, the computer creates a stack for you.
- Each time you invoke a method, the method is placed on top of the stack.
- When the method returns or exits, the method is popped off the stack.
- The diagram on the next page shows a sample stack for a simple Java program.

Activation record

- Every method call results in an activation record which contains:
  - Local variables and their values.
  - The location (in the caller) of the call.

Stacks and Methods



Time: 0

Empty Stack

Time 1:

Push: main()

Time 2:

Push: square()

Time 3:

Pop: square()

returns a value.

method exits.

Time 4:

Pop: main()

returns a value.

method exits.

Factorials

- Computing **factorials** are a classic problem for examining recursion.
- A factorial is defined as follows:

  n! = n * (n-1) * (n-2) …. * 1;
- For example:

  1! = 1 (Base Case)

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

  4! = 4 * 3 * 2 * 1 = 24

  5! = 5 * 4 * 3 * 2 * 1 = 120

If you study this table closely, you will start to see a **pattern**

Seeing the Pattern

- Seeing the pattern in the factorial example is difficult at first.
- But, once you see the pattern, you can apply this pattern to create a recursive solution to the problem.
- Divide a problem up into:
  - What it can do (usually a **base case**)
  - What it cannot do
    - What it cannot do resembles original problem
    - The function launches a new copy of itself (recursion step) to solve what it cannot do.

Factorials

- Computing factorials are a classic problem for examining recursion.
- A factorial is defined as follows:

  n! = n * (n-1) * (n-2) …. * 1;

- For example:

  1! = 1 (Base Case)

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

  4! = 4 * 3 * 2 * 1 = 24

  5! = 5 * 4 * 3 * 2 * 1 = 120

The **pattern** is as follows:

You can compute the factorial of

any number (n) by taking **n** and

multiplying it by the factorial of **(n-1)**

For example:

5! = 5 * 4!

(which translates to 5! = 5 * 24 = 120)

The Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example--the factorial function:
  - n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

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

- **As a Java method:**

```
// recursive factorial function
public static int  recursiveFactorial(int n)
{
    // basis case
    if  (n  ==  0)  return  1;
    // recursive case
    else return  n  *  recursiveFactorial(n- 1);
}
```

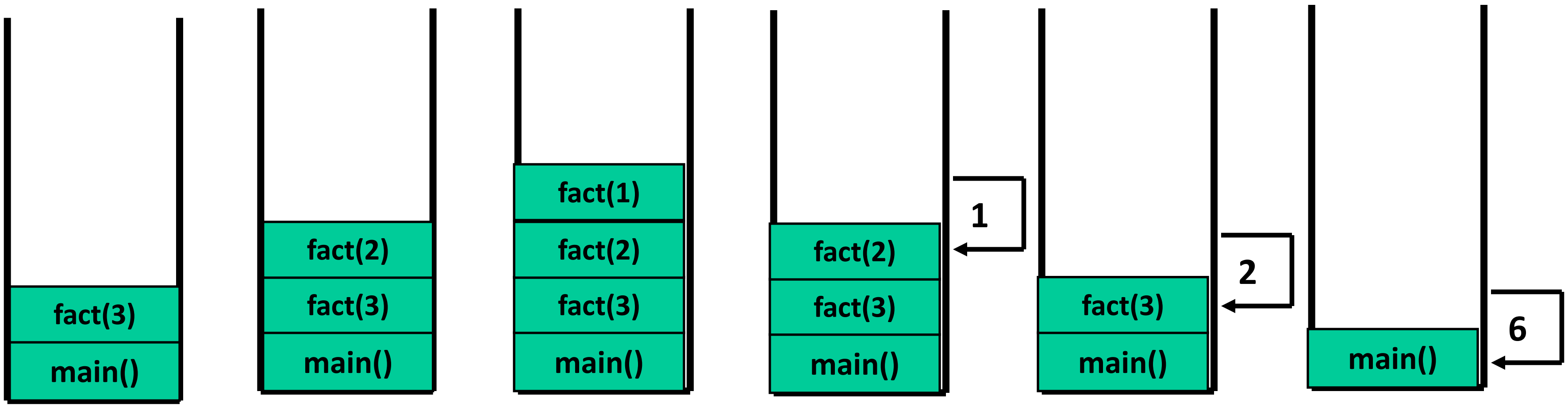
Recursive Solution

```
public class FindFactorialRecursive
{
   public static void main (String args[])
   {
       for (int i = 1; i < 10; i++)
             System.out.println ( i + "! = " +
   findFactorial(i));
   }

   public static int findFactorial (int number)
   {
       if (  (number == 1)  || (number == 0)  )
             return 1;
       else
             return (number * findFactorial (number-1));
   }
}
```

Base Case

Finding the factorial of 3



Time 2:

Push: fact(3)

Time 3:

Push: fact(2)

Time 4:

Push: fact(1)

Time 5:

Pop: fact(1)

returns 1.

Time 6:

Pop: fact(2)

returns 2.

Time 7:

Pop: fact(3)

returns 6.

Inside findFactorial(3):

if (number <= 1) return 1;

else return (3 * factorial (2));

Inside findFactorial(2):

if (number <= 1) return 1;

else return (2 * factorial (1));

Inside findFactorial(1):

if (number <= 1) return 1;

else return (1 * factorial (0));

Components of repetitive control

Initialize

Establish an initial state that will be modified toward the termination condition

These two make sure termination condition will occur

Test

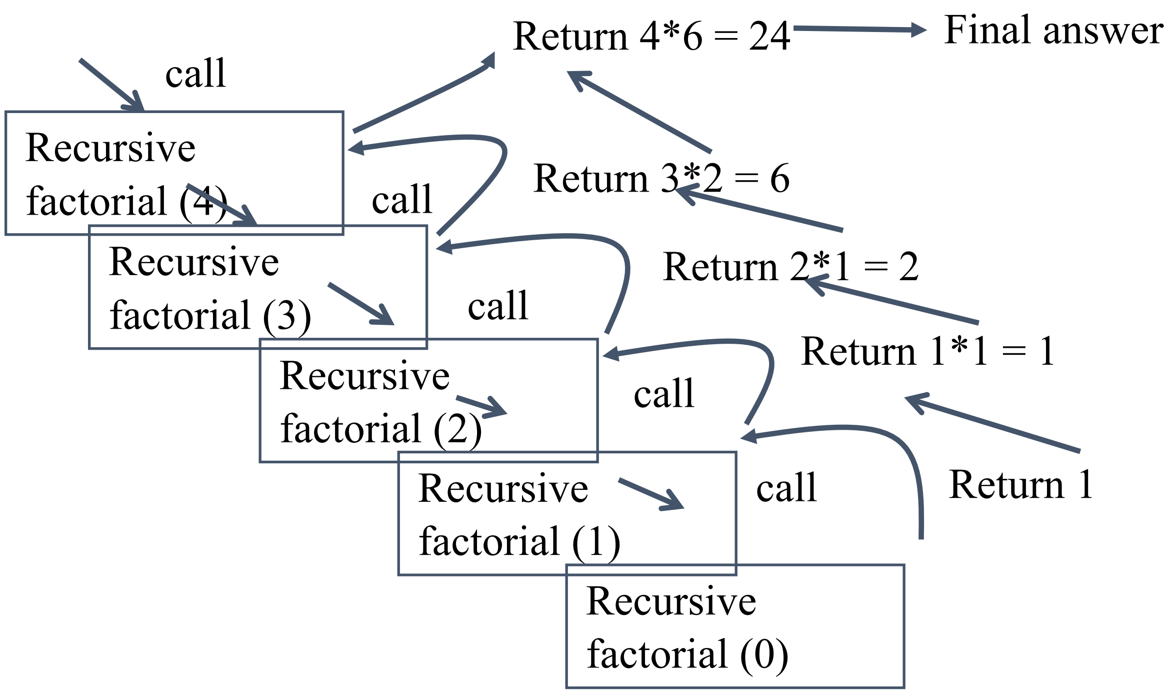
Compare the current state to the termination condition and terminate the repetition if equal

Modify

Change the state in such a way that it moves toward the termination condition

Visualizing recursion

```
public static int recursiveFactorial(int n) {
  // recursive factorial function
  if (n == 0) return 1;                    // basis case
  else return n * recursiveFactorial(n-1);  // recursive case
}
```



Linear Recursion

Test for base cases

- Begin by testing for a set of base cases (there should be at least one).
- Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.

Recur once

- Perform a single recursive call
- This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
- Define each possible recursive call so that it makes progress towards a base case.

Recursive Problem Solving

- Recursion is not always a best approach though !
- Recursion is often a good idea when a problem can be solved by breaking it into one or more smaller problems of the same form. The process is:
  - Figure out how to solve the easy case, i.e. the base case.
  - Figure out how to move the hard case toward the easy case.

Recursion pseudocode

- Nearly every recursive method ends up looking like the following:

```
recursiveMethod(input)
{
     if (input represents a base case)
     {
          handle the base case directly.
     }
     else
     {
          call recursiveMethod one or more times
          passing it only part of the input.
     }
}
```

There may be more than one base case.

Recursion pseudocode variation

Sometimes there is nothing to do for the base case. We just want to stop:

```
recursiveMethod(input)
{
      if (input is NOT the base case)
      {
           call recursiveMethod one or more times
           passing it only part of the input.
      }

      // No else statement.  Nothing to do for the base case.
}
```

Recursive tracing

- Consider the following recursive method:

```
public static int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n / 10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

  – What is the result of the following call?

```
mystery(648)
```

A recursive trace

```
mystery(648):
  ▪ int a = 648 / 10;           // 64
  ▪ int b = 648 % 10;           //  8
  ▪ return mystery(a + b);      // mystery(72)

    mystery(72):
    ▪ int a = 72 / 10;          // 7
    ▪ int b = 72 % 10;          // 2
    ▪ return mystery(a + b);    // mystery(9)

        mystery(9):
        ▪ return 9;
```

Recursive tracing 2

- Consider the following recursive method:

```
public static int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}
```

  - What is the result of the following call?

    `mystery(348)`

A recursive trace 2

```
mystery(348)
  ▪ int a = mystery(34);
      • int a = mystery(3);
          return (10 * 3) + 3;      // 33
      • int b = mystery(4);
          return (10 * 4) + 4;      // 44
      • return (100 * 33) + 44;     // 3344

  ▪ int b = mystery(8);
      return (10 * 8) + 8;          // 88

  – return (100 * 3344) + 88;       // 334488
```

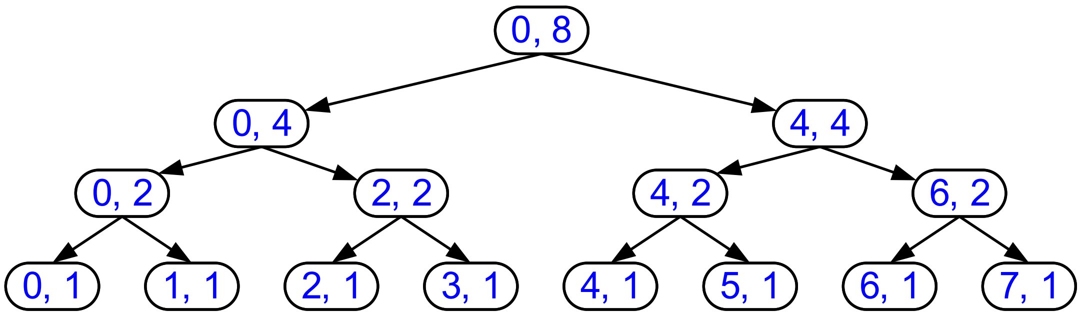
– What is this method really doing?

Ex: A Binary Recursive Method

Problem: add all the numbers in an integer array A:

```
Algorithm BinarySum(A, i, n):
    Input: An array A and integers i and n
    Output: The sum of the n integers in A starting at index i
      if n = 1 then
         return A[i ]
      return BinarySum(A, i, n/2) + BinarySum(A, i + n/2, n/2)
```

- **Example trace:**



$$
\begin{aligned}
F_2 &= F_1 + F_0 = 1 + 0 = 1 \\
F_3 &= F_2 + F_1 = 1 + 1 = 2 \\
F_4 &= F_3 + F_2 = 2 + 1 = 3 \\
F_5 &= F_4 + F_3 = 3 + 2 = 5 \\
F_6 &= F_5 + F_4 = 5 + 3 = 8 \\
F_7 &= 13 \; (8{+}5) \\
F_8 &= 21 \; (13{+}8) \\
F_9 &= 34 \; (21{+}13) \\
F_{10} &= 55 \; (34{+}21) \\
&\cdots
\end{aligned}
$$

Fibonacci numbers

Ex: Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

- Recursive algorithm (first attempt):

```
Algorithm BinaryFib(k):
   Input: Nonnegative integer k
   Output: The kth Fibonacci number F_k
      if k = 1 then
 return k
      else
 return BinaryFib(k - 1) + BinaryFib(k-2)
```

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!