

CSC 172– Data Structures and Algorithms

Lecture #12

Spring 2018

Please put away all electronic devices



Announcement

- What should you study this week:
 - Chapter 7: Recursion (E-textbook)

Agenda

- Recursion
- More recursion

FIBONACCI SEQUENCE AND RECURSION

Agenda

- The worst algorithm you can think of!
- An iterative solution
- A better iterative solution
- The repeated squaring trick

FIBONACCI SEQUENCE

Fibonacci sequence

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- $F[0] = 0$
- $F[1] = 1$
- $F[2] = F[1] + F[0] = 1$
- $F[3] = F[2] + F[1] = 2$
- $F[4] = F[3] + F[2] = 3$
- $F[n] = F[n-1] + F[n-2]$

Time to watch a video!

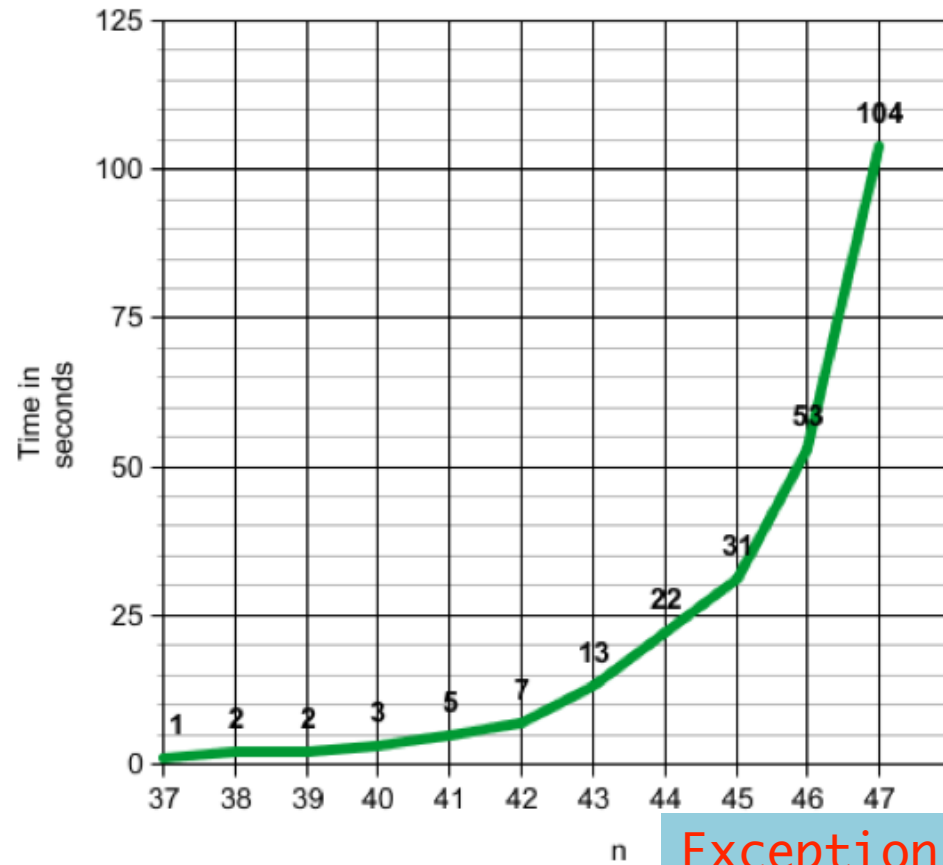
- Fibonacci Number and Golden Ratio
- http://www.youtube.com/watch?v=aB_KstBiou4

Recursion – fib1()

```
/**
 *-----
 *  the most straightforward algorithm to compute F[n]
 *-----
 */
long fib1(int n) {
    if (n <= 1) return n;
    return fib1(n-1) + fib1(n-2);
}
```

Typical Runtime

Fib1 run time



Exception in thread "main"
java.lang.StackOverflowError
at Fibonacci.fib1([Fibonacci.java:7](#))
at Fibonacci.fib1([Fibonacci.java:7](#))
at Fibonacci.fib1([Fibonacci.java:7](#))
at Fibonacci.fib1([Fibonacci.java:7](#))

On large numbers

- Looks like the run time is doubled for each $n++$
- We won't be able to compute $F[140]$ if the trend continues
- The age of the universe is 15 billion years $< 2^{60}$ sec
- The function looks ... exponential
 - Is there a theoretical justification for this?

ANALYSIS OF FIB1()

Guess and induct

- Let $T[n]$ be the time $\text{fib1}(n)$ takes
- For $n=0, 1$, suppose it takes d mili-sec
- $T[0] = T[1] = d$
- For $n > 1$, suppose it takes c mili-sec in $\text{fib1}(n)$ not counting the recursive calls
- To estimate $T[n]$, we can
 - Guess a formula for it
 - Prove by induction that it works

$$T[n] = c + T[n-1] + T[n-2] \quad \text{when } n > 1$$

The guess

- Bottom-up iteration

- $T[0] = T[1] = d$

- $T[2] = c + 2d$

- $T[3] = 2c + 3d$

- $T[4] = 4c + 5d$

- $T[5] = 7c + 8d$

- $T[6] = 12c + 13d$

Fibonacci Sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- Can you guess a formula for $T[n]$?

- $T[n] = (F[n+1] - 1)c + F[n+1]d$

The Proof


- The base cases: $n=0,1$
- The hypothesis: suppose
 - $T[m] = (F[m+1] - 1)*c + F[m+1]*d$ for all $m < n$
- The induction step:
 - $$\begin{aligned} T[n] &= c + T[n-1] + T[n-2] \\ &= c + (F[n] - 1)*c + F[n]*d \\ &\quad + (F[n-1] - 1)*c + F[n-1]*d \\ &= (F[n+1] - 1)*c + F[n]*d \end{aligned}$$

How does this help?

$$F[n] = \frac{\phi^n - (-1/\phi)^n}{\sqrt{5}}$$

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.6$$

The golden ratio



So, there are constants C , D such that

$$C\phi^n \leq T[n] \leq D\phi^n$$

This explains the exponential-curve we saw

$$T(n) = \Theta(\phi^n)$$

From intuition to formality

- Suppose `fib1 (140)` runs on a computer with $C = 10^{-9}$:

$$10^{-9}(1.6)^{140} \geq 3.77 \cdot 10^{19} > 100 \cdot \text{age of univ.}$$

- A Linear time algorithm using ArrayList
- A linear time algorithm using arrays
- A linear time algorithm with constant space

BETTER ALGORITHMS FOR COMPUTING $F[N]$

An algorithm using ArrayList

*Time Complexity: $T(n) = \theta(n)$
Space Complexity: $S(n) = \theta(n)$*

```
long fib2(int n) {  
    // this is one implementation option  
    if (n <= 1) return n;  
    ArrayList<Long> A = new ArrayList<>();  
    A.add((long) 0);  
    A.add((long) 1);  
    for (int i=2; i<=n; i++) {  
        A.add(A.get(i-1)+ A.get(i-2));  
    }  
    return A.get(n);  
}
```

Guess how large an n we can handle this time?

Data

n	10^6	10^7	10^8	10^9
# seconds	1	1	9	Eats up all Heap Space

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at java.util.Arrays.copyOf(Arrays.java:3210)
at java.util.Arrays.copyOf(Arrays.java:3181)
at java.util.ArrayList.grow(ArrayList.java:261)
at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:235)
at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:227)
at java.util.ArrayList.add(ArrayList.java:458)
at Fibonacci.fib2(Fibonacci.java:16)
```

How about an array?

Time Complexity: $T(n) = \theta(n)$
Space Complexity: $S(n) = \theta(n)$

```
long fib3(int n) {  
    if (n <= 1) return n;  
  
    long[] A = new long[n+1];  
    A[0] = 0;  
    A[1] = 1;  
    for (int i=2; i<=n; i++) {  
        A[i] = A[i-1]+A[i-2];  
    }  
    return A[n];  
}
```

Guess how large an n we can handle this time?

Data

n	10^6	10^7	10^8	10^9
# seconds	1	1	1	Stack Overflow

```
Exception in thread "main"  
java.lang.OutOfMemoryError: Java heap space  
at Fibonacci.fib3(Fibonacci.java:24)  
at Fibonacci.main(Fibonacci.java:53)
```


Dynamic programming!

Time Complexity: $T(n) = \theta(n)$
Space Complexity: $S(n) = \theta(1)$

```
long fib4(long n) {  
    if (n <= 1) return n;  
    long a=0, b=1, temp = 0;  
    for (long i=2; i<= n; i++) {  
        temp = a + b; // F[i] = F[i-2] + F[i-1]  
        a = b;        // a = F[i-1]  
        b = temp;     // b = F[i]  
    }  
    return temp;  
}
```

Guess how large an n we can handle this time?

Data

n	10^8	10^9	10^{10}	10^{11}
# seconds	1	3	35	359

```
long n= 1000000000;
```

```
Total Time = 2312683996  
3311503426941990459
```

```
long n= 100000000000000000000000L;
```

```
Exception in thread "main"  
java.lang.OutOfMemoryError: Java heap  
space  
at java.lang.Long.valueOf(Long.java:840)
```

Conclusion

- Iteration is better/faster than recursion...

May not be always!

- The repeated squaring trick

AN EVEN FASTER ALGORITHM

Math helps!

- We can re-formulate the problem a little:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} F[3] \\ F[2] \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \overset{2}{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}$$

$$\begin{bmatrix} F[n+1] \\ F[n] \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

How to we compute A^n quickly?

- Want

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

- But can we even compute 3^n quickly?

First algorithm

```
long power1(int n, int base) {  
    long ret=1;  
    for (int i=0; i<n; i++)  
        ret *= base;  
    return ret;  
}
```

When $n = 10^{10}$ it took 44 seconds

Second algorithm

```
long power2 (int base, long n) {  
    Long ret;  
    if(n== 0) return 1;  
    if(n% 2 == 0) {  
        ret= power2(base, n/2);  
        return ret * ret;  
    }else{  
        ret= power2(base, (n-1)/2);  
        return base * ret * ret;  
    }  
}
```

$$3^n = \begin{cases} 3^{n/2} \cdot 3^{n/2} & \text{if } n \text{ is even} \\ 3 \cdot 3^{n/2} \cdot 3^{n/2} & \text{otherwise} \end{cases}$$

Time Complexity: $T(n) = O(\log n)$

When $n = 10^{19}$ it took < 1 second

Two 2x2 multiplication

```
int[][] matrix_multiplication_2x2(int[][] a, int[][] b) {  
    int[][] result = new int[2][2];  
  
    result[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0];  
    result[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1];  
    result[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0];  
    result[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1];  
    return result;  
}
```

Time Complexity: $T(n) = \theta(1)$

Second Algorithm For Matrix

```
int[][] power2Matrix(int[][] matrix, long n) {  
    int[][] ret;  
    if (n == 1) return matrix;  
    if (n % 2 == 0) {  
        ret = power2Matrix(matrix, n/2);  
        return matrix_multiplication_2x2(ret, ret) ;  
    } else {  
        ret = power2Matrix(matrix, (n-1)/2);  
        return matrix_multiplication_2x2(matrix,  
            matrix_multiplication_2x2(ret, ret));  
    }  
}
```

Time Complexity: $T(n) = O(\log n)$

Runtime analysis

- **First** algorithm $O(n)$
- **Second** algorithm $O(\log n)$
- We can apply the second algorithm to the Fibonacci problem: fib4() has the following data

n	10^8	10^9	10^{10}	10^{19}
# seconds	1	1	1	1

Conclusion

- Recursion is
— powerful!



Recursion

Examples: From Textbook

1. Largest Number / Cumulative Sum
2. Greatest common divisor (GCD)
3. Log
4. Power
5. Many problems on List ADT

All these problems can be solved using either **recursive** or **iterative** algorithms.

Objectives

- Thinking **recursively**
- **Tracing** execution of a recursive method
- Writing recursive algorithms
- **Towers of Hanoi** problem with recursion
- Backtracking to solve search problems, as in mazes

RECURSIVE THINKING

Recursive Thinking

- Recursion is:
 - A **problem-solving approach**, that can ...
 - Generate **simple solutions** to ...
 - **Certain kinds** of problems that would be **difficult to solve in other ways**
- Recursion splits a problem:
 - Into one or more simpler versions of itself

Recursive Thinking (cont.)

General Recursive Algorithm

if the problem can be solved directly for the current value of n

Solve it

else

Recursively apply the algorithm to **one or more** problems involving **smaller** values of n

Combine the **solutions to the smaller problems** to get the solution to the original problem

EXAMPLES

Recursive Algorithm for Finding the Length of a String

if the string is empty (has no characters)

The length is 0

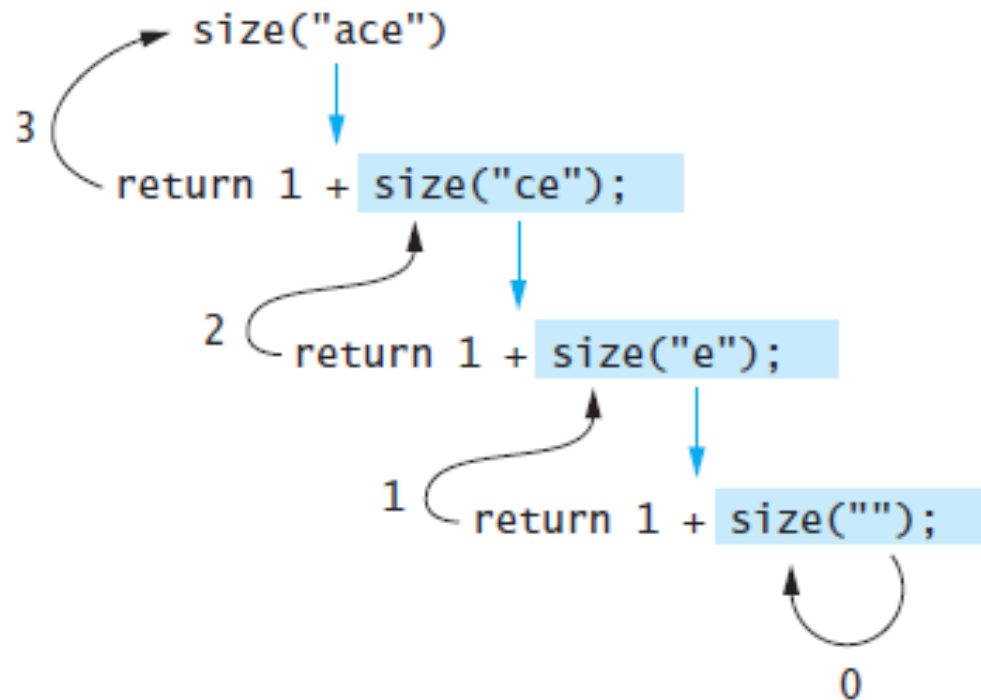
else

3. The length is **1** + the length of the string
that excludes the first character

Recursive Algorithm for Finding the Length of a String (cont.)

```
public static int size(String str) {  
    if ( str == null || str.equals(""))  
        return 0;  
    else {  
        int output = 1 + size(str.substring(1));  
        return output;  
    }  
}
```

Tracing a Recursive function



Recursive Algorithm for Printing String Characters

```
public static void print_chars(String str) {  
    if (str == null || str.equals("")) {  
        return;  
    } else {  
        System.out.println(str.charAt(0));  
        print_chars(str.substring(1));  
    }  
}
```


Recursive Algorithm for Printing String Characters in Reverse Order

```
public static void print_chars_rev(String str) {  
    if (str == null || str.equals("")) {  
        return;  
    } else {  
        print_chars_rev(str.substring(1));  
        System.out.println(str.charAt(0));  
    }  
}
```

Recursive Design Example: mystery

- What does this do?

```
int mystery (int n) {  
    if (n == 0)  
        return 0;  
    else  
        return n * mystery(n-1);  
}
```

Proving a Recursive Method Correct

- Recall **Proof by Induction**
- Prove the theorem for the base case(s): $n=0$
- Show that:
 - If the theorem is assumed true for n ,
 - Then it must be true for $n+1$
- Result: Theorem true for all $n \geq 0$

The Stack and Activation Frames

- Java maintains a stack on which it saves new information in the form of an *activation frame*
- The activation frame contains storage for
 - function arguments
 - local variables (if any)
 - the return address of the instruction that called the function
- Whenever a new function is called (recursive or otherwise), Java pushes a new activation frame onto the stack

Run-Time Stack and Activation Frames (cont.)

Frame for
size("")

str: ""
return address in size("e")

Frame for
size("e")

str: "e"
return address in size("ce")

Frame for
size("ce")

str: "ce"
return address in size("ace")

Frame for
size("ace")

str: "ace"
return address in caller

Run-time stack after all calls

Frame for
size("e")

str: "e"
return address in size("ce")

Frame for
size("ce")

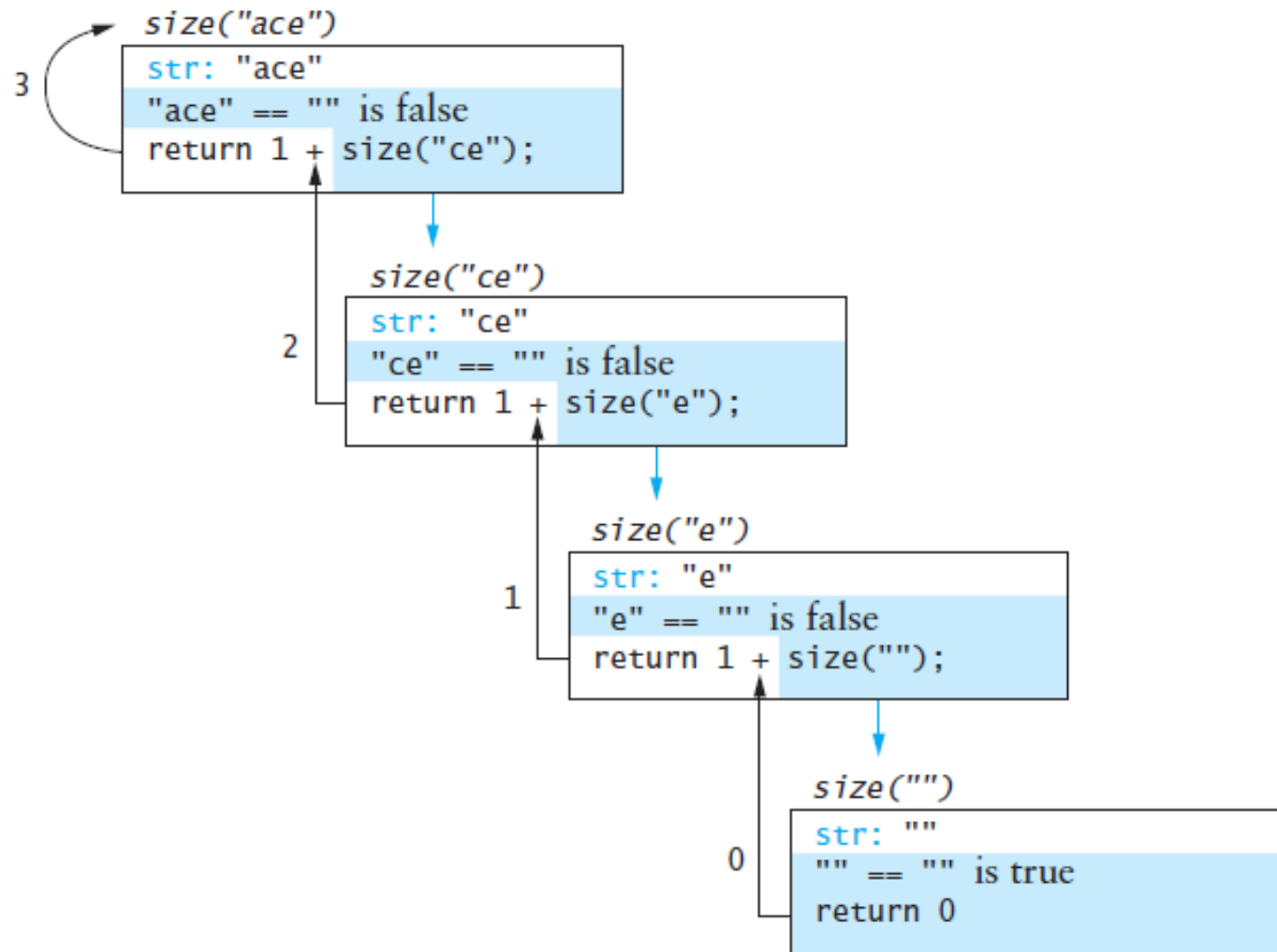
str: "ce"
return address in size("ace")

Frame for
size("ace")

str: "ace"
return address in caller

Run-time stack after return from last call

Run-Time Stack and Activation Frames



Section 7.2

RECURSIVE DEFINITIONS OF MATHEMATICAL FORMULAS

Recursive Definitions of Mathematical Formulas

- Mathematicians often use **recursive definitions** of formulas that lead naturally to recursive algorithms
- Examples include:
 - factorials
 - powers
 - greatest common divisors (gcd)

Factorial of n : $n!$

- The factorial of n , or $n!$ is defined as follows:

$$0! = 1$$

$$n! = n \times (n - 1)! \text{ (for } n > 0\text{)}$$

- The base case: n is equal to 0
- The second formula is a recursive definition

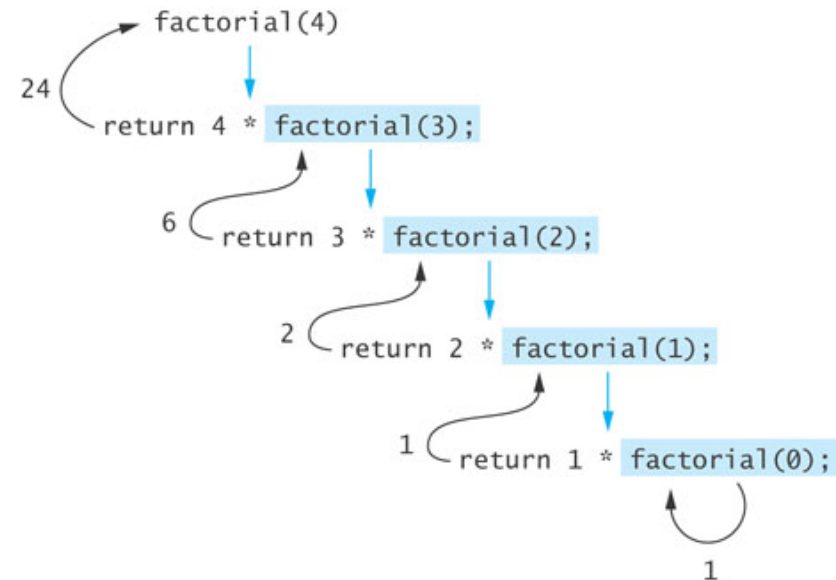
Factorial of n : $n!$ (cont.)

The recursive definition can be expressed by the following algorithm:

```
if  $n$  equals 0  
   $n!$  is 1  
else  
   $n! = n * (n - 1)!$ 
```

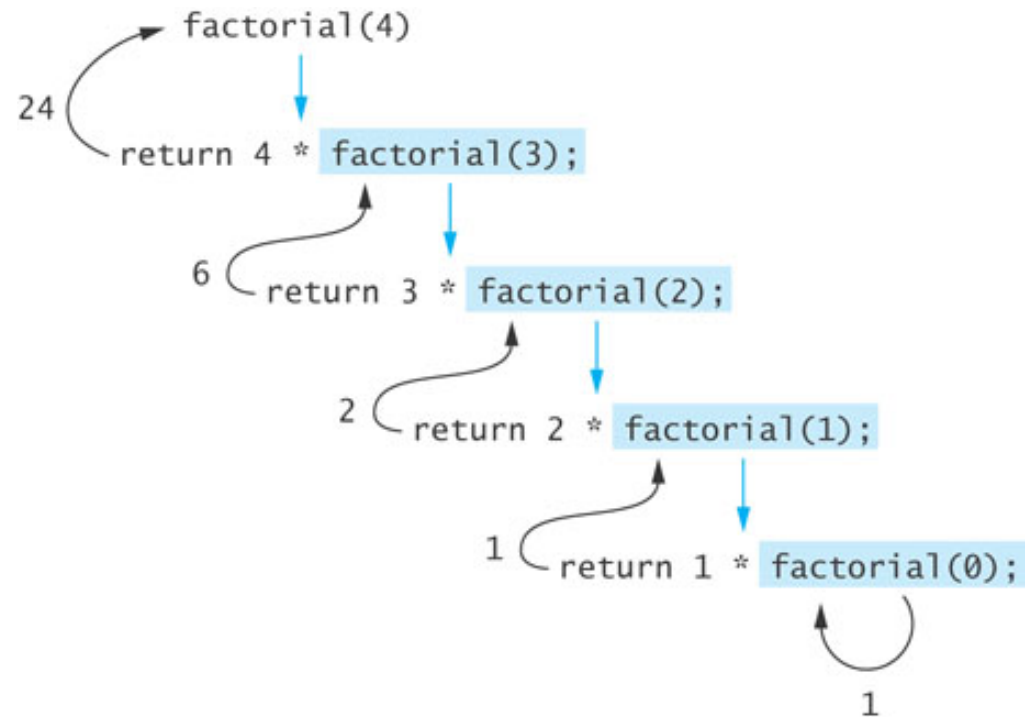
The last step can be implemented as:

```
return  $n * \text{factorial}(n - 1);$ 
```



Factorial of n : $n!$ (cont.)

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



Recursive Algorithm for Calculating x^n (cont.)

```
double power(double x, int n) {  
    if (n == 0)  
        return 1;  
    else if (n > 0)  
        return x * power(x, n - 1);  
    else  
        return 1.0 / power(x, -n);  
}
```

Recursive Algorithm for Calculating gcd (cont.)

```
int gcd(int m, int n) {  
    if (m < n)  
        return gcd(n, m); // Transpose arguments  
    else if (m % n == 0)  
        return n;  
    else  
        return gcd(n, m % n);  
}
```

Recursion Versus Iteration

Recursion and iteration are similar

Iteration

A loop repetition condition determines whether to repeat the loop body or exit from the loop

Recursion

the condition usually tests for a base case

You can always write an iterative solution to a problem that is solvable by recursion

BUT

A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

Tail Recursion or Last-Line Recursion

- When recursion involves single call that is at the end ...
- It is called **tail recursion** and it easy to make iterative

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

- It is a straightforward process to turn such a function into an iterative one

Iterative factorial function

```
int factorial_iter(int n) {  
    int result = 1;  
    for (int k = 1; k <= n; k++)  
        result = result * k;  
    return result;  
}
```


Efficiency of Recursion

Recursive method often slower than iterative;
why?

- Overhead for loop repetition smaller than

- Overhead for call and return

If easier to develop algorithm using recursion,

- Then code it as a recursive method:

- Software engineering benefit probably outweighs ...

- Reduction in efficiency

Don't "optimize" prematurely!

Efficiency of Recursion (cont.)

- Memory usage
 - A recursive version can require significantly more memory than an iterative version because of the need to save local variables and parameters on a stack

Acknowledgement

- Douglas Wilhelm Harder.
 - Thanks for making an excellent set of slides for ECE 250 *Algorithms and Data Structures* course
- Prof. Hung Q. Ngo:
 - Thanks for those beautiful slides created for CSC 250 (Data Structures) course at UB.
- Many of these slides are taken from these two sources.