CPSC 221

Basic Algorithms and Data Structures

May 22, 2015

Administrative stuff

Hw1 posted. Hw2 to come.

Office hours:

Monday - 9:30am to 10:30am | Shu Yang | Demco Table 3 (11:00 to noon later)
Tuesday - 9:00am to 10:00am | Kamil Khan | Demco Table 2
Tuesday - 10:30am to 12:00pm | Nasa Rouf | Demco Table 3 (really 10:45-11:45)
Tuesday - 12:00pm to 2:00pm | Henry Li | Demco Table 3
Tuesday - 3:00pm to 4:00pm | Kaitlyn Melton | Demco Table 3
Tuesday - 5:00pm to 6:00pm | Angad Kalra | Demco Table 1
Wednesday - 9:00am to 10:00am | Issam Laradji | Room ICCSX337
Thursday - 10:00am to 12:00pm | Devon | Room ICCSX141
Thursday, - 12:00pm to 2:00pm | Kurt Eiselt | Room ICCSX151
Thursday - 2:00pm to 3:00pm | Jason | Demco Table 3

Latest info always available at https://my.cs.ubc.ca/students/ta-hours

Recursion

Having survived previous courses, you've had some experience with recursion.

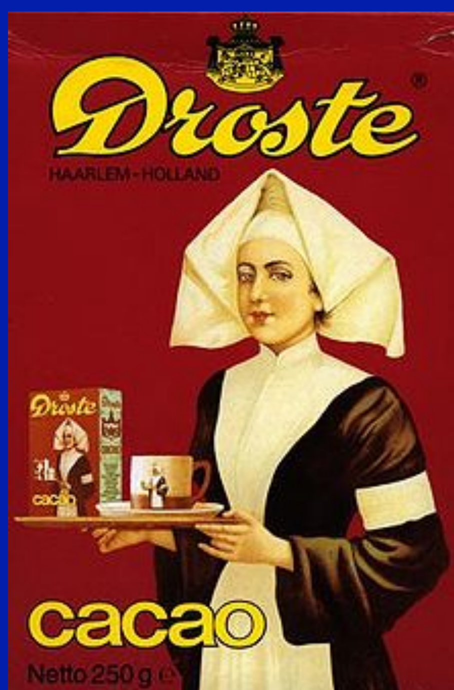
Students aren't always well-versed in how recursion works, or what the resource implications may be.

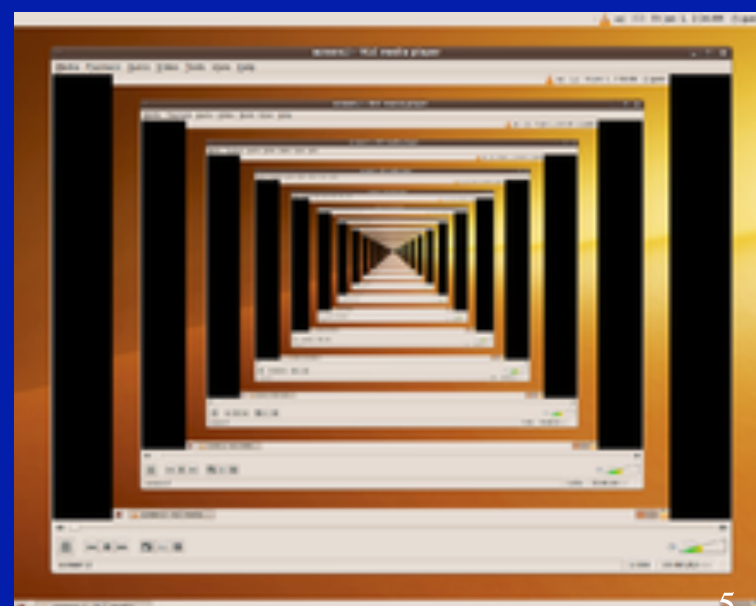
Recursion

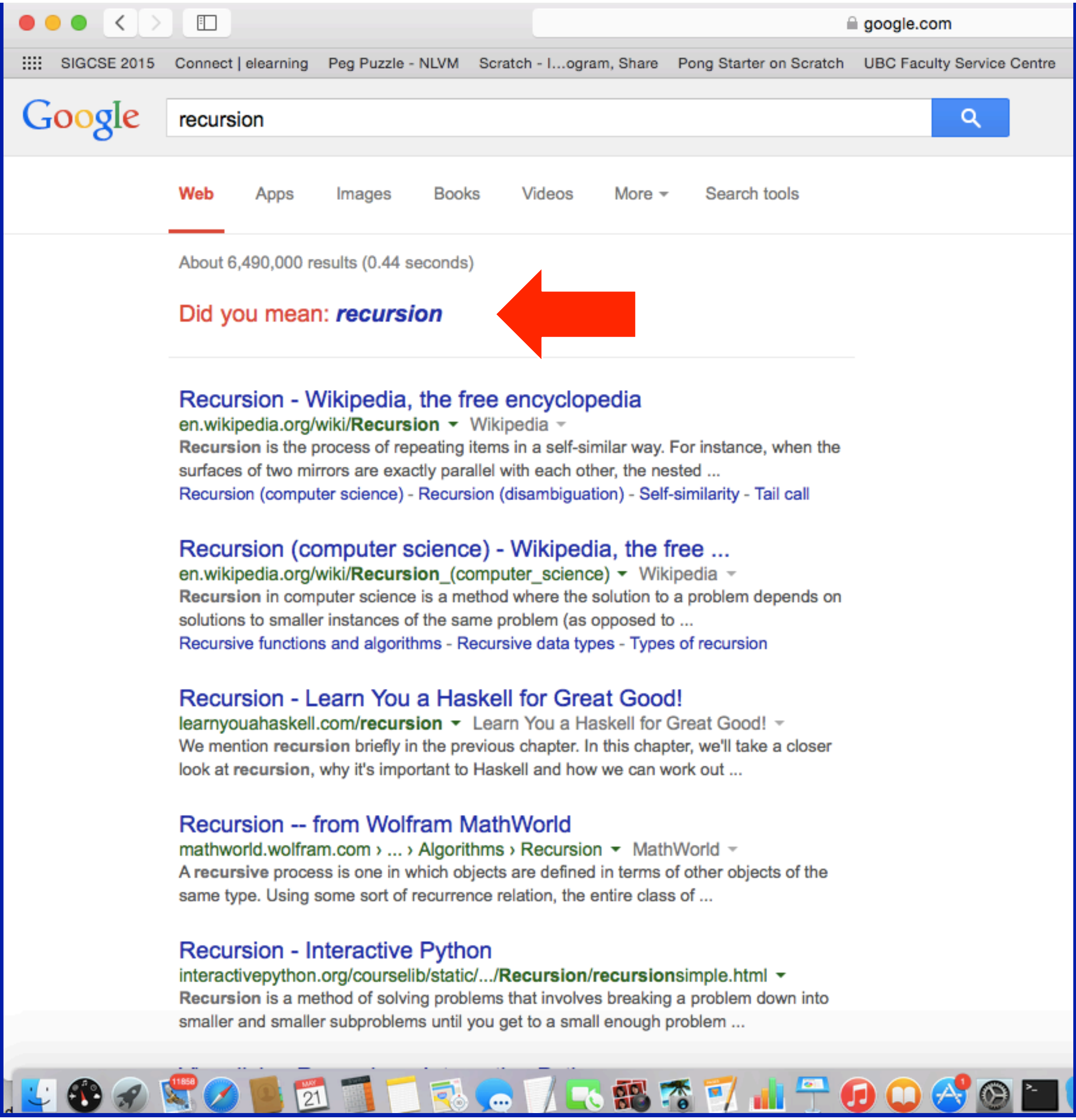
Recursion is a means of obtaining repetitive behaviour from a procedure that is defined in terms of itself. The procedure is self-referential. (That last part should sound familiar to many of you.)

Recursion

Recursion is a means of obtaining repetitive behaviour from a procedure that is defined in terms of itself. The procedure is self-referential. (That last part should sound familiar to many of you.)
Recursion exists outside of computing.





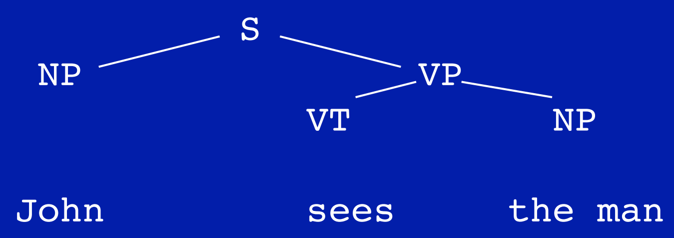

google.com
SIGCSE 2015
Connect | elearning
Peg Puzzle - NLVM
Scratch - I...ogram, Share
Pong Starter on Scratch
UBC Faculty Service Centre
Google
recursion
Web
Apps
Images
Books
Videos
More
Search tools
About 6,490,000 results (0.44 seconds)
Did you mean: recursion
Recursion - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Recursion Wikipedia
Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other, the nested ...
Recursion (computer science) - Recursion (disambiguation) - Self-similarity - Tail call
Recursion (computer science) - Wikipedia, the free ...
en.wikipedia.org/wiki/Recursion_(computer_science) Wikipedia
Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to ...
Recursive functions and algorithms - Recursive data types - Types of recursion
Recursion - Learn You a Haskell for Great Good!
learnyouahaskell.com/recursion Learn You a Haskell for Great Good!
We mention recursion briefly in the previous chapter. In this chapter, we'll take a closer look at recursion, why it's important to Haskell and how we can work out ...
Recursion -- from Wolfram MathWorld
mathworld.wolfram.com › ... › Algorithms › Recursion MathWorld
A recursive process is one in which objects are defined in terms of other objects of the same type. Using some sort of recurrence relation, the entire class of ...
Recursion - Interactive Python
interactivepython.org/courselib/static/.../Recursion/recursionsimple.html
Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem ...

Recursion

```
PHRASAL RULES:
S  -> NP VP
NP -> DET N
NP -> PN
NP -> NP PP
VP -> VI
VP -> VT NP
VP -> VP PP
PP -> PREP NP

LEXICAL RULES:
PN   -> John | Mary
DET  -> the
N    -> man | dog | lake | house
VI   -> runs
VT   -> likes | sees
PREP -> with | at | on | by

John sees the man.
John sees the man with the dog.
John sees the man with the dog at the house.
John sees the man with the dog at the house by the lake.
...
```

Recursion

```
PHRASAL RULES:
S   -> NP VP
NP  -> DET N
NP  -> PN
NP  -> NP PP
VP  -> VI
VP  -> VT NP
VP  -> VP PP
PP  -> PREP NP
```

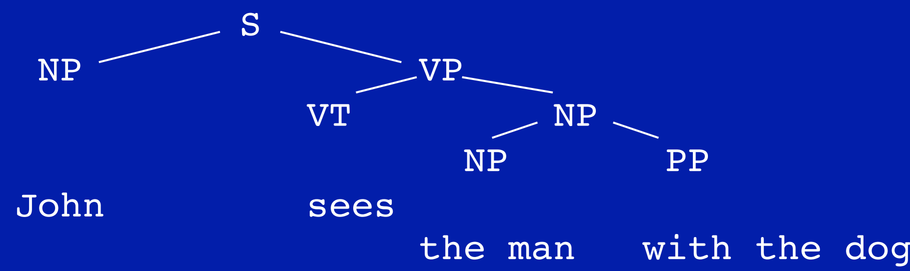



```
LEXICAL RULES:
PN   -> John | Mary
DET  -> the
N    -> man | dog | lake | house
VI   -> runs
VT   -> likes | sees
PREP -> with | at | on | by

John sees the man.
John sees the man with the dog.
John sees the man with the dog at the house.
John sees the man with the dog at the house by the lake.
...
```

Recursion

PHRASAL RULES:

```
S  -> NP VP
NP -> DET N
NP -> PN
NP -> NP PP
VP -> VI
VP -> VT NP
VP -> VP PP
PP -> PREP NP
```



LEXICAL RULES:

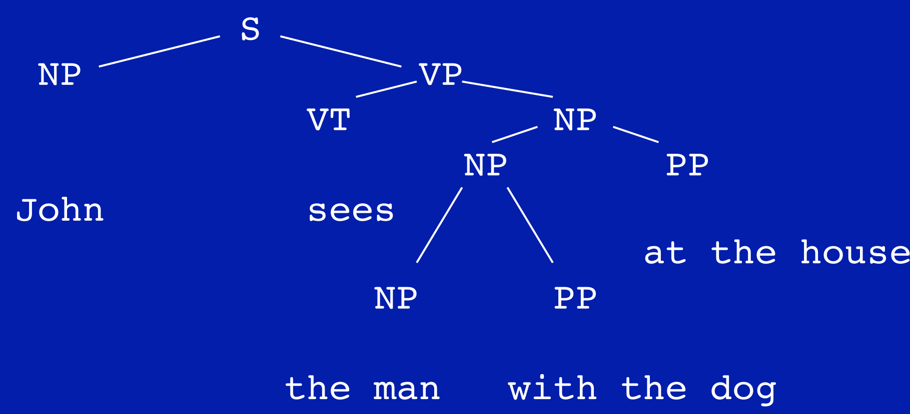
```
PN   -> John | Mary
DET  -> the
N    -> man | dog | lake | house
VI   -> runs
VT   -> likes | sees
PREP -> with | at | on | by
```

```
John sees the man.
John sees the man with the dog.
John sees the man with the dog at the house.
John sees the man with the dog at the house by the lake.
...
```

Recursion

PHRASAL RULES:

```
S  -> NP VP
NP -> DET N
NP -> PN
NP -> NP PP
VP -> VI
VP -> VT NP
VP -> VP PP
PP -> PREP NP
```



LEXICAL RULES:

```
PN   -> John | Mary
DET  -> the
N    -> man | dog | lake | house
VI   -> runs
VT   -> likes | sees
PREP -> with | at | on | by
```

```
John sees the man.
John sees the man with the dog.
John sees the man with the dog at the house.
John sees the man with the dog at the house by the lake.
...
```

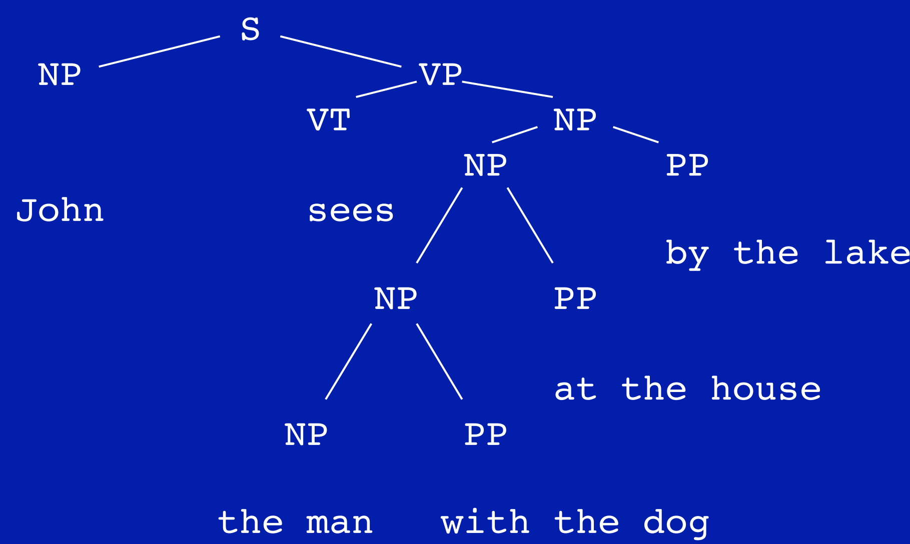
Recursion

PHRASAL RULES:

```
S  -> NP VP
NP -> DET N
NP -> PN
NP -> NP PP
VP -> VI
VP -> VT NP
VP -> VP PP
PP -> PREP NP
```

LEXICAL RULES:

```
PN   -> John | Mary
DET  -> the
N    -> man | dog | lake | house
VI   -> runs
VT   -> likes | sees
PREP -> with | at | on | by
```




```
John sees the man.
John sees the man with the dog.
John sees the man with the dog at the house.
John sees the man with the dog at the house by the lake.
...
```

Recursion

Recursion is also an approach to problem solving where a problem is split into smaller instances of itself. The splitting continues until a trivial, easy-to-solve instance is reached, which can then be used to build up a solution to the original problem.

As we explore data structures, we'll see that recursion is a very useful and powerful tool.

Thinking recursively

Do not start with code. Write the *story* of the problem, including the data definition!

Define the problem: What should be done given a specific input?

Solve some example cases by hand.

Identify and solve the (usually simple) base case(s).

Figure out how to break the complex cases down in terms of any smaller case(s). For the smaller cases, call the function recursively and assume it works. Do not think about how!

Implementing recursion

Once you have all that, write out your solution in comments (i.e., make your own “template”). Then fill out the code and test.

Should be easy. If it’s hard, maybe you’re not assuming your recursive call works (i.e., thinking too much, not abstracting enough).

Recursion

A recursive procedure consists of three parts:

1 The base case or termination condition. Usually the first thing done upon entering a recursive procedure

2 The reduction step -- the operation that moves the computation closer to the termination condition

3 The recursive procedure call itself

Recursion

Finding the length of a list:

Data definition is just your basic list.
Return the number of elements in a list.
What's the length of the empty list?
What's the length of '(a b c)?
What's the base case?
How does it break down into smaller parts?
- What's the reduction?
- Where does the recursive call go?

Recursion

base case/termination

```
(define (count list)
  (if (null? list)
      0
      (+ 1 (count (rest list)))))
```

recursive call

reduction step

Recursion

Finding if some item is an element of a list:

Data definition is just your basic list.
Return the list that begins with the found element
    (other versions just return true)
What do you return if the list is empty?
What do you return for (element `b `(a b c))?
What do you return for (element `x `(a b c))?
What's the base case?
How does it break down into smaller parts?
    What's the reduction?
    Where does the recursive call go?

Recursion

base case/termination

```
(define (element i list)
  (if (null? list)
      '()
      (if (equal? i (first list))
          list
          (element i (rest list)))))
```

recursive call

reduction step

Recursion

Sometimes the solution is handed to you on a platter.

The mathematical definition for factorial:

$$n! = \begin{cases} 1 \text{ for } n = 0 \\ n * (n - 1)! \text{ for } n > 0 \end{cases}$$

Recursion

base case/termination

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

recursive call

reduction step

Visualizing recursion

How does it work? The activation stack model:

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

Activation stack

Each function call (whether recursive or not) generates a stack frame (also known as an activation frame or activation record) holding local variables and the program point to return to, which is pushed on a stack (the activation stack or call stack) that tracks the current chain of function calls.

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;  // using 'fact' for brevity
```

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```

```
fact(4)
```

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```

```
fact(4)
4 * fact(3)
```

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```

fact(3)
fact(4) 4 * fact(3)

Visualizing recursion

How does it work? The activation stack model:

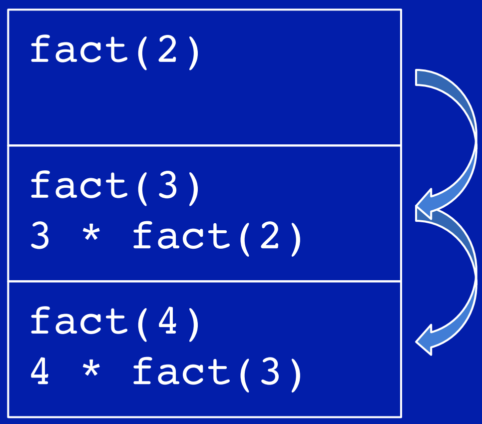
```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}


cout << fact(4) << endl;
```

fact(3) 3 * fact(2)
fact(4) 4 * fact(3)

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```



Visualizing recursion

How does it work? The activation stack model:

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

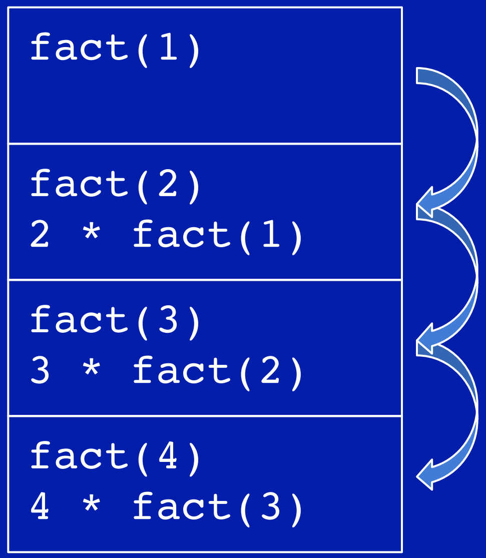
```
cout << fact(4) << endl;
```

fact(2) 2 * fact(1)
fact(3) 3 * fact(2)
fact(4) 4 * fact(3)

Visualizing recursion

How does it work? The activation stack model:

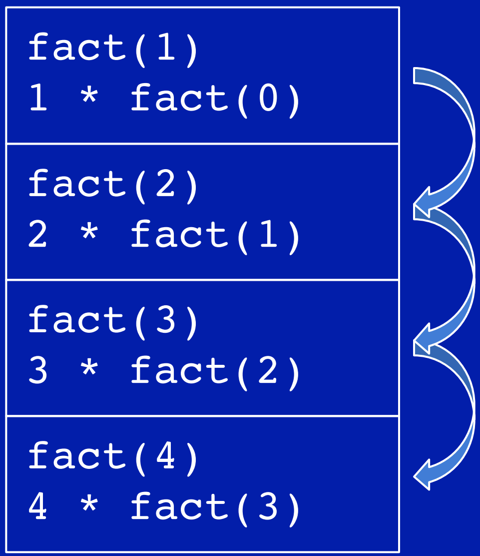
```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}


cout << fact(4) << endl;
```



Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```



Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```

`fact(0)`
`fact(1)` `1 * fact(0)`
`fact(2)` `2 * fact(1)`
`fact(3)` `3 * fact(2)`
`fact(4)` `4 * fact(3)`

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```

```
fact(0)
1

fact(1)
1 * fact(0)

fact(2)
2 * fact(1)

fact(3)
3 * fact(2)

fact(4)
4 * fact(3)
```

Visualizing recursion

How does it work? The activation stack model:

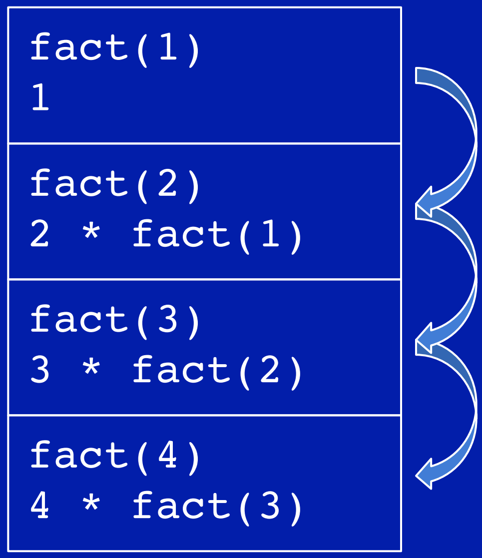
```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}


cout << fact(4) << endl;
```

fact(1) 1 * 1
fact(2) 2 * fact(1)
fact(3) 3 * fact(2)
fact(4) 4 * fact(3)

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```



Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```

`fact(2)` `2 * 1`
`fact(3)` `3 * fact(2)`
`fact(4)` `4 * fact(3)`

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```

fact(2) 2
fact(3) 3 * fact(2)
fact(4) 4 * fact(3)

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```

`fact(3)` `3 * 2`
`fact(4)` `4 * fact(3)`

Visualizing recursion

How does it work? The activation stack model:

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

```
cout << fact(4) << endl;
```

```
fact(3)
6
```

```
fact(4)
4 * fact(3)
```

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```

```
fact(4)
4 * 6
```

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
```

```
fact(4)
24
```

Visualizing recursion

How does it work? The activation stack model:

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


cout << fact(4) << endl;
24
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
  4 * fact(3)
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
  4 * fact(3)
  4 * 3 * fact(2)
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
  4 * fact(3)
  4 * 3 * fact(2)
  4 * 3 * 2 * fact(1)
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
  4 * fact(3)
  4 * 3 * fact(2)
  4 * 3 * 2 * fact(1)
  4 * 3 * 2 * 1 * fact(0)
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
  4 * fact(3)
  4 * 3 * fact(2)
  4 * 3 * 2 * fact(1)
  4 * 3 * 2 * 1 * fact(0)
  4 * 3 * 2 * 1 * 1
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
  4 * fact(3)
  4 * 3 * fact(2)
  4 * 3 * 2 * fact(1)
  4 * 3 * 2 * 1 * fact(0)
  4 * 3 * 2 * 1 * 1
  4 * 3 * 2 * 1
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
  4 * fact(3)
  4 * 3 * fact(2)
  4 * 3 * 2 * fact(1)
  4 * 3 * 2 * 1 * fact(0)
  4 * 3 * 2 * 1 * 1
  4 * 3 * 2 * 1
  4 * 3 * 2
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
  4 * fact(3)
  4 * 3 * fact(2)
  4 * 3 * 2 * fact(1)
  4 * 3 * 2 * 1 * fact(0)
  4 * 3 * 2 * 1 * 1
  4 * 3 * 2 * 1
  4 * 3 * 2
  4 * 6
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
  4 * fact(3)
  4 * 3 * fact(2)
  4 * 3 * 2 * fact(1)
  4 * 3 * 2 * 1 * fact(0)
  4 * 3 * 2 * 1 * 1
  4 * 3 * 2 * 1
  4 * 3 * 2
  4 * 6
  24
```

Visualizing recursion

How does it work? The substitution model:

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


cout << fact(4) << endl;
24
```

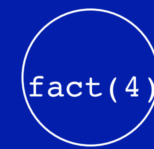
Visualizing recursion

How does it work? The recursion tree model:

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


cout << fact(4) << endl;
```

Visualizing recursion

How does it work? The recursion tree model:

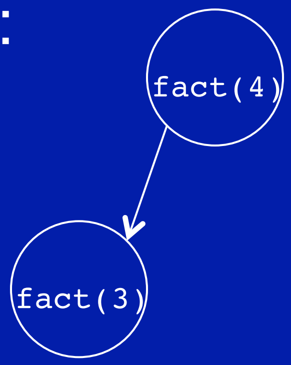



```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}


cout << fact(4) << endl;
```

Visualizing recursion

How does it work? The recursion tree model:

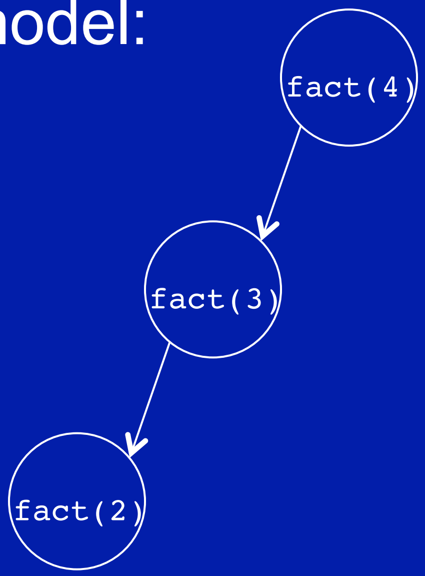
```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}


cout << fact(4) << endl;
```



Visualizing recursion

How does it work? The recursion tree model:

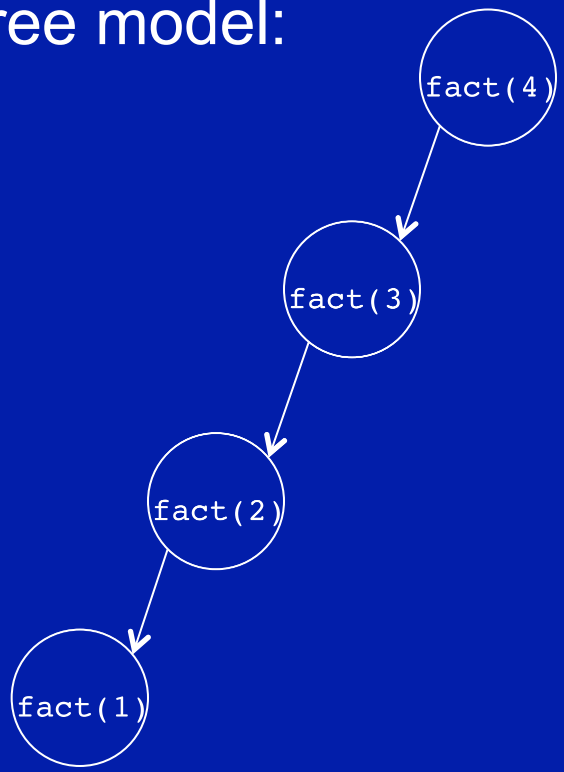
```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}


cout << fact(4) << endl;
```



Visualizing recursion

How does it work? The recursion tree model:

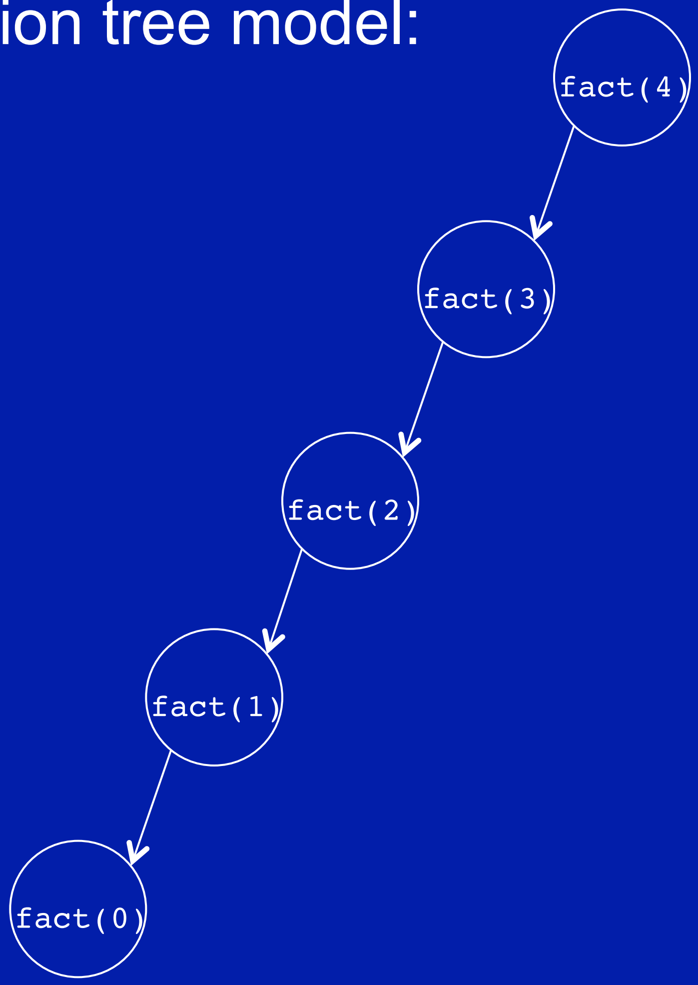
```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}


cout << fact(4) << endl;
```



Visualizing recursion

How does it work? The recursion tree model:

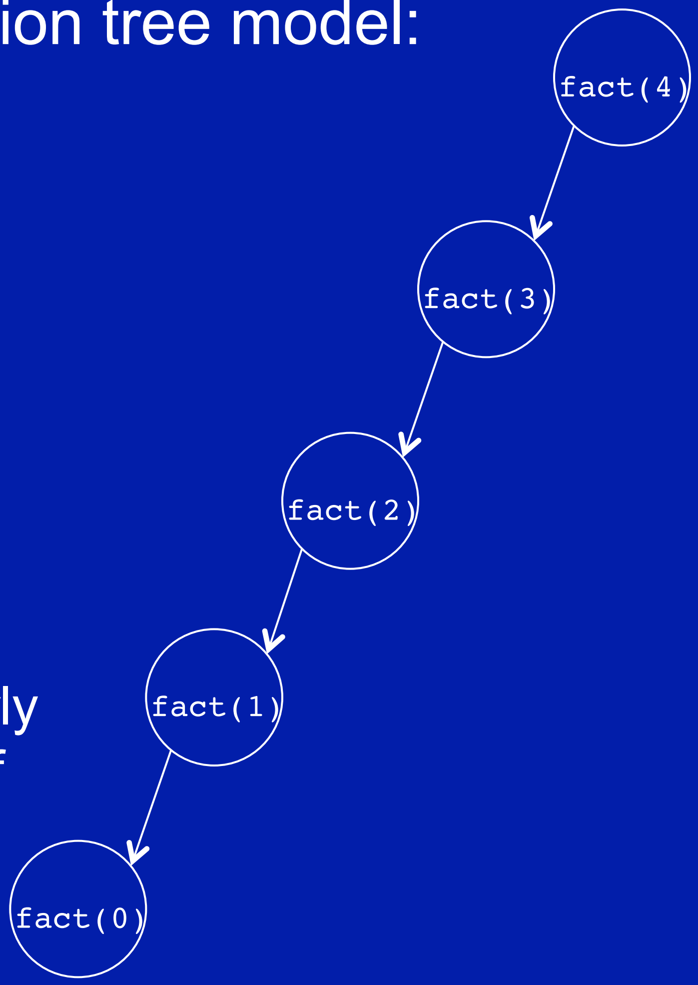
```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}


cout << fact(4) << endl;
```



Visualizing recursion

How does it work? The recursion tree model:

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


cout << fact(4) << endl;
```

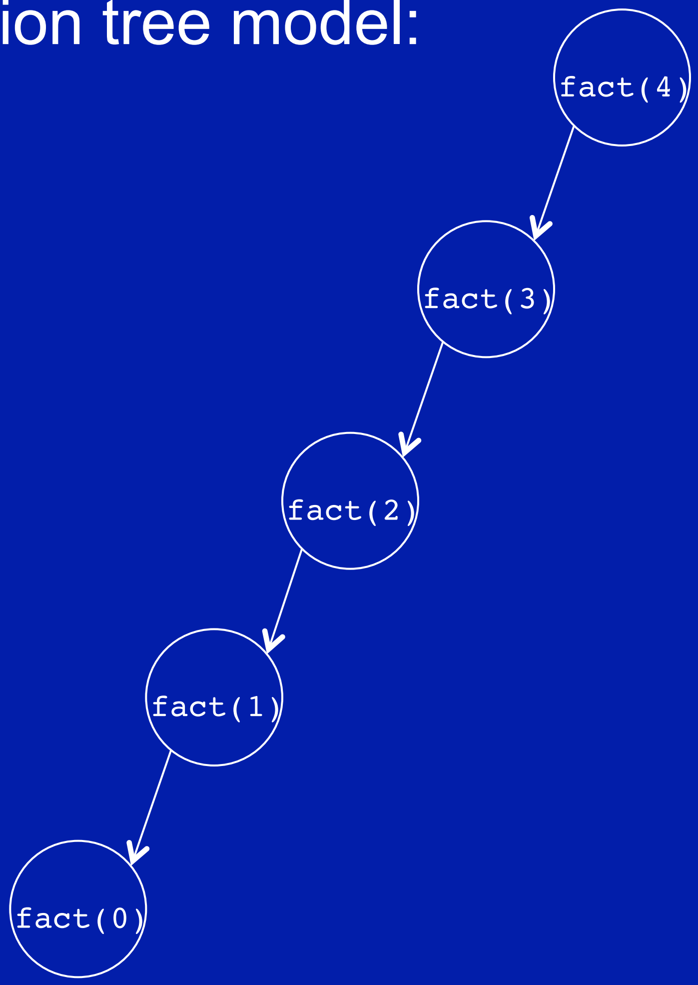


Seriously, this visualization isn't nearly as useful as the others in the case of factorial, but it will give a good idea of how much work has to be done with the next example of recursion.

Visualizing recursion

How does it work? The recursion tree model:

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


cout << fact(4) << endl;
```



What’s the time complexity for this approach to computing factorials?

Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

Visualizing recursion

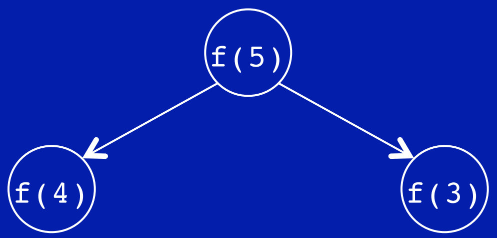
How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

Visualizing recursion

How does it work? The recursion tree model:

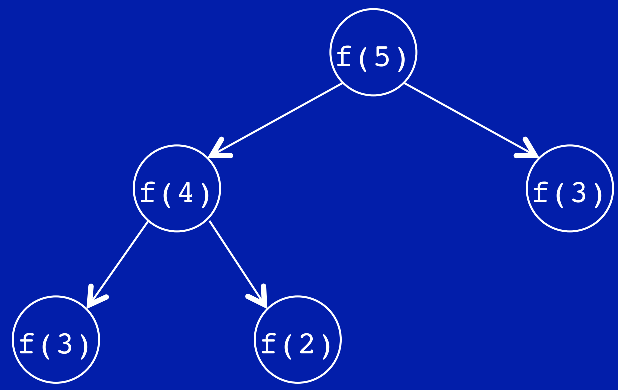
```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



Visualizing recursion

How does it work? The recursion tree model:

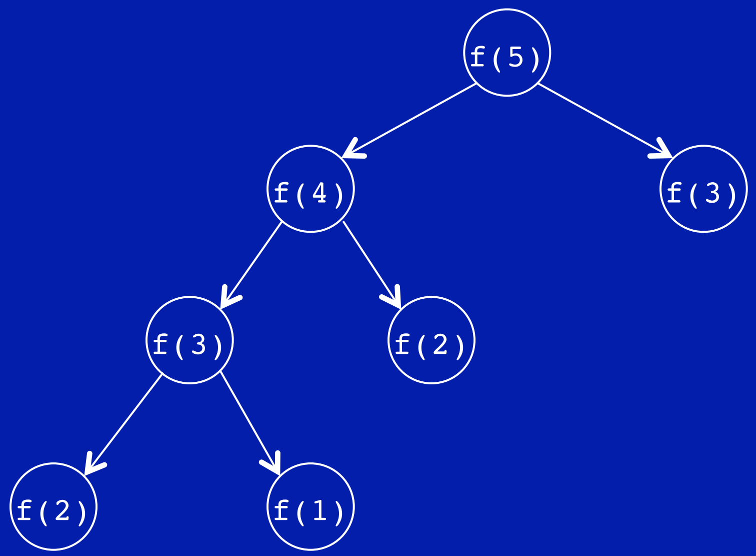
```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



Visualizing recursion

How does it work? The recursion tree model:

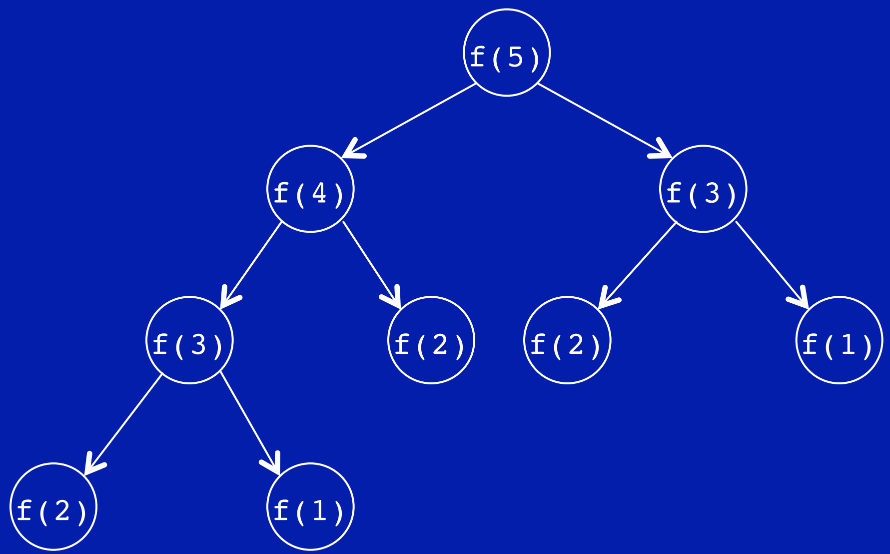
```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

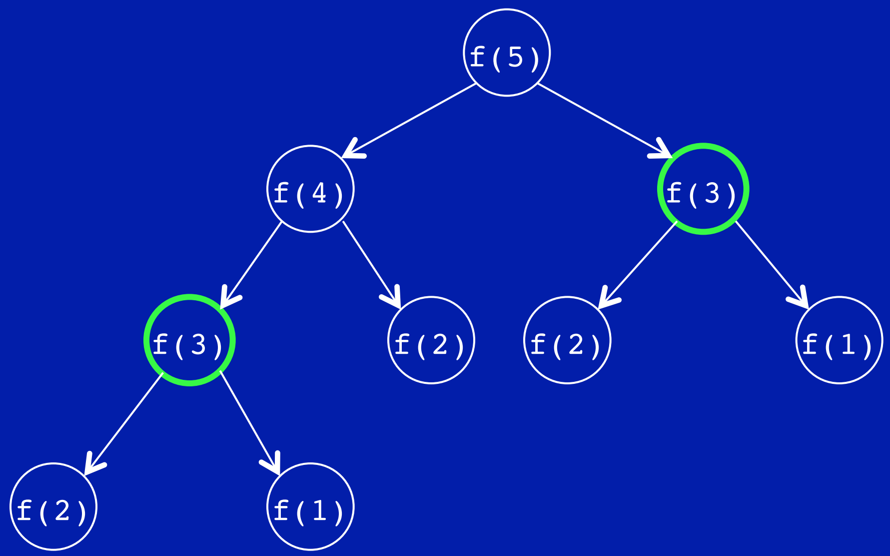


Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

fib(3) is evaluated 2 times

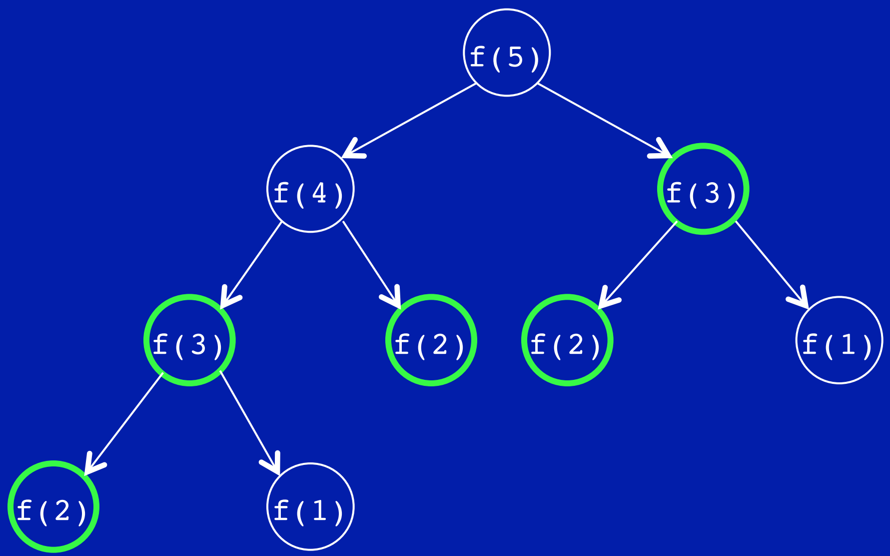


Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

fib(3) is evaluated 2 times
fib(2) is evaluated 3 times

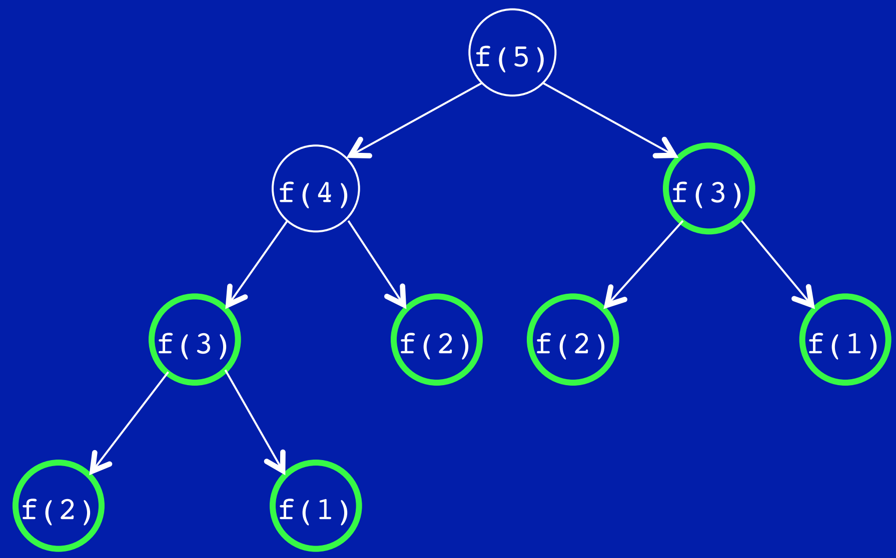


Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

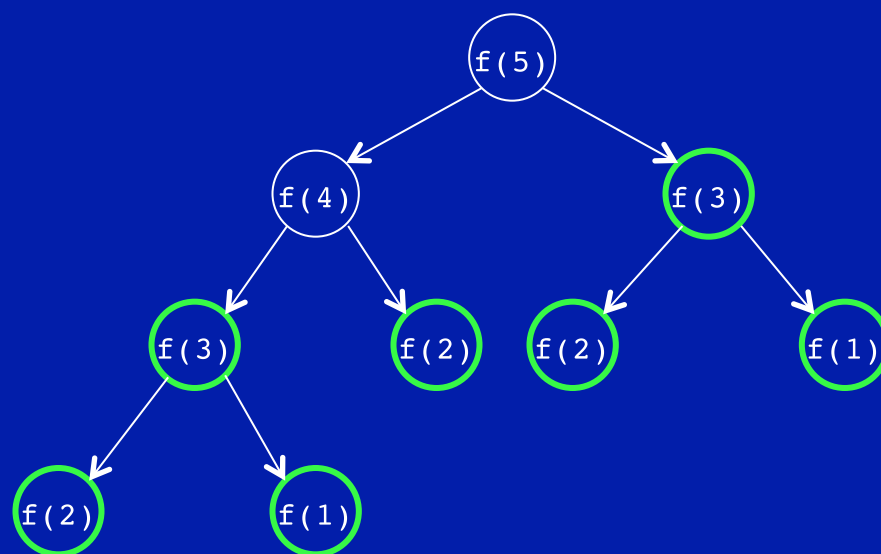
fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times



Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



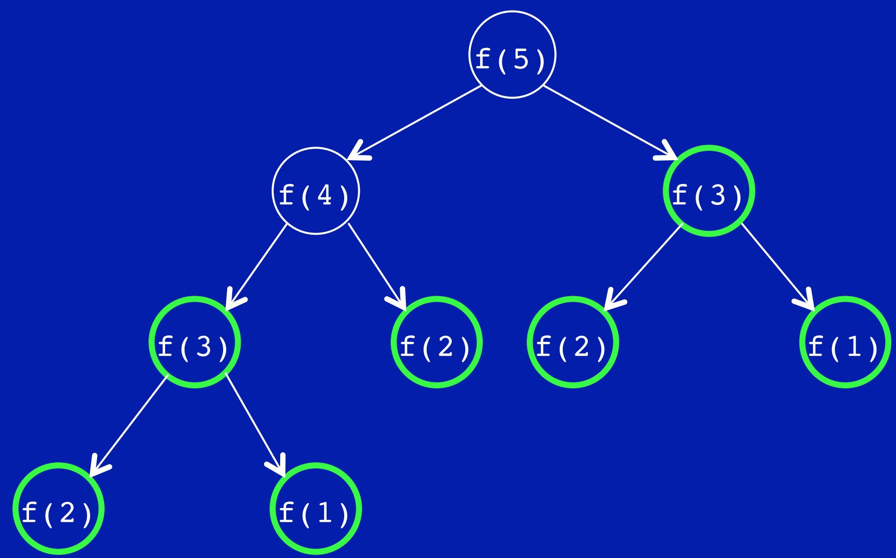
fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
Can you use the visualization to help you estimate the growth rate of the number of function calls (= activation frames to be processed)?

Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



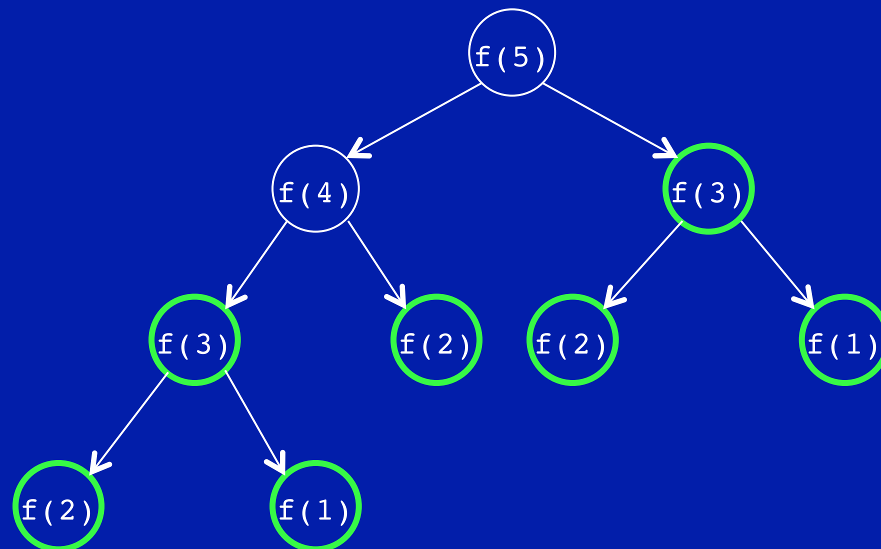
fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
Not very. Your book confirms that T(n) for fibonacci(n) increases exponentially with n, because of all the duplicated function calls.

Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



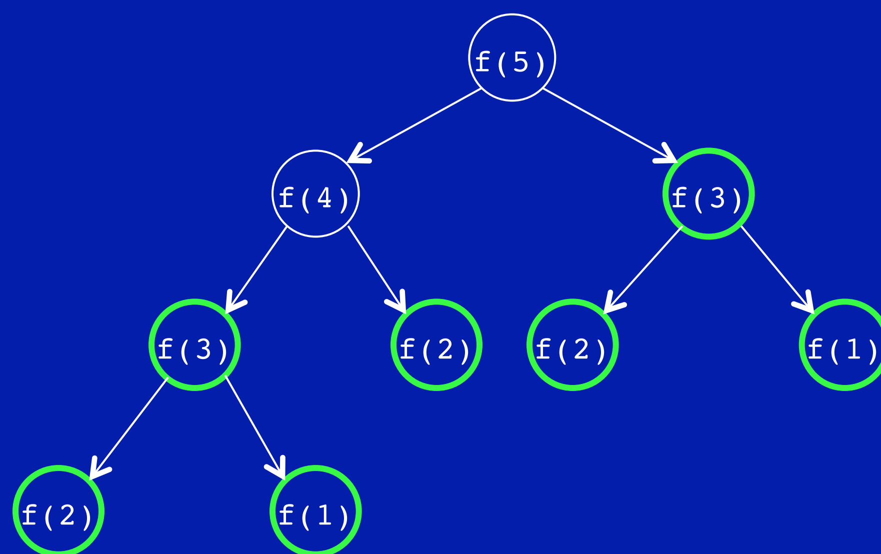
fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
Your book says that fib(100) requires about 2^{100} activation frames. If your computer can process 1,000,000 activation frames per second, it'll take 3×10^{16} years.

Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



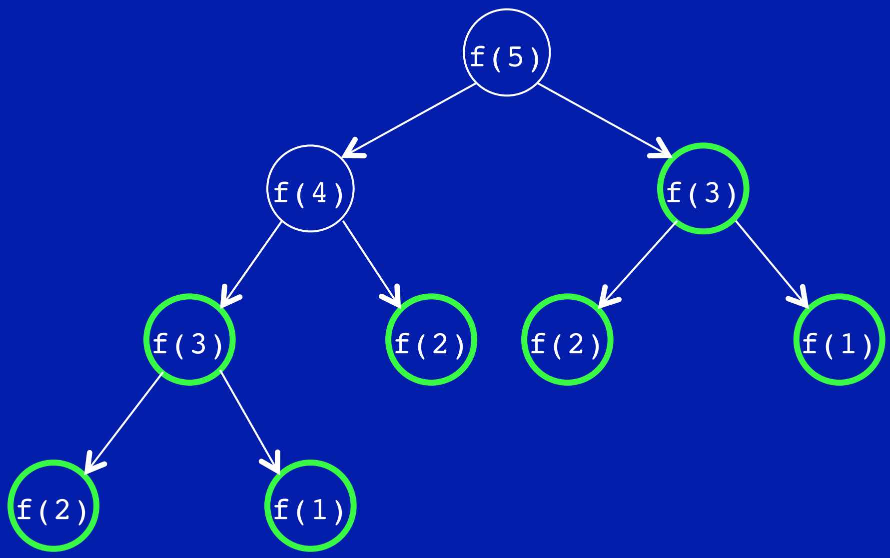
fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
But computers have become much faster since the book was written. With gigahertz speeds, we might process 1,000,000,000 frames/sec. Now we're down to a mere 3 x 10^{13} years. I feel much better now.

Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
We'll let you do the formal proof on your own.

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

main

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
     fib5
main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
          fib4
     fib5 fib5
main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
               fib3
          fib4 fib4
     fib5 fib5 fib5
main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                fib2
           fib3 fib3
      fib4 fib4 fib4
 fib5 fib5 fib5 fib5
main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                   fib2
              fib3 fib3 fib3
         fib4 fib4 fib4 fib4
    fib5 fib5 fib5 fib5 fib5
main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3
          fib4 fib4 fib4 fib4 fib4
     fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3
          fib4 fib4 fib4 fib4 fib4 fib4
     fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3
          fib4 fib4 fib4 fib4 fib4 fib4 fib4
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                  fib2      fib1
             fib3 fib3 fib3 fib3 fib3      fib2
        fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4
   fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4           fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2                fib2
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2                fib2
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2                fib2      fib1
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3 fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2                fib2      fib1
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3 fib3 fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2                fib2      fib1
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3 fib3 fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2           fib1
               fib3 fib3 fib3 fib3 fib3           fib2                fib2      fib1
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4           fib3 fib3 fib3 fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main main main main main
```

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2       fib1
               fib3 fib3 fib3 fib3 fib3      fib2                fib2      fib1
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3 fib3 fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main main main main main
```

fib(3) is evaluated 2 times

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
                fib3 fib3 fib3 fib3 fib3      fib2                fib2      fib1
           fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3 fib3 fib3 fib3
      fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main main main main main
```

fib(3) is evaluated 2 times
fib(2) is evaluated 3 times

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2                fib2      fib1
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3 fib3 fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main main main main main
```

fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times (that’s just a little consistency check)

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2                fib2      fib1
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3 fib3 fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main main main main main
```

What are your thoughts about memory usage (space complexity)? Exponential too?

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2                fib2      fib1
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3 fib3 fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main main main main main
```

The height of the recursion tree (plus 1) from several slides ago is the maximum number of activation frames on the stack at any given time – in this case it's 5. Memory for the stack is limited. If we attempt to push more frames than can be stored on the stack, a stack overflow occurs and the program will crash.

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
3 + fib(3)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
3 + fib(3)
3 + fib(2) + fib(1)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
3 + fib(3)
3 + fib(2) + fib(1)
3 + 1 + fib(1)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
3 + fib(3)
3 + fib(2) + fib(1)
3 + 1 + fib(1)
4 + fib(1)
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
3 + fib(3)
3 + fib(2) + fib(1)
3 + 1 + fib(1)
4 + fib(1)
4 + 1
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
3 + fib(3)
3 + fib(2) + fib(1)
3 + 1 + fib(1)
4 + fib(1)
4 + 1
5
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
3 + fib(3)
3 + fib(2) + fib(1)
3 + 1 + fib(1)
4 + fib(1)
4 + 1
5
```

fib(3) is evaluated 2 times

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
3 + fib(3)
3 + fib(2) + fib(1)
3 + 1 + fib(1)
4 + fib(1)
4 + 1
5
```

fib(3) is evaluated 2 times
fib(2) is evaluated 3 times

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
3 + fib(3)
3 + fib(2) + fib(1)
3 + 1 + fib(1)
4 + fib(1)
4 + 1
5
```

fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times (one more consistency check)

Recursion mythology

A common complaint about recursion, as we've just seen, is the resource consumption in terms of memory (activation stack space) as well as time (handling the function calls and putting frames on/taking frames off the stack).

The culprit here is the repeated postponement of computations by pushing those computations on the stack.

Consider factorial. What’s the space complexity? How many frames go on the stack for factorial(10)? factorial(100)? factorial(1000)?

Recursion mythology

But what if there were a type of recursion that worked without postponing computations? If this were so, we could have factorial using O(1) stack space instead of O(n) stack space.

Recursion mythology

But what if there were a type of recursion that worked without postponing computations? If this were so, we could have factorial using O(1) stack space instead of O(n) stack space.

This type of recursion exists, and it's inspired by the fact that if we can pass a partially-completed computation through the recursive function calls, we don't have to postpone any computations...we just do them as we go.

If we don't postpone any computations, we don't really need to push anything on the stack.

Recursion mythology

In other words, I could start with an accumulator "variable" to hold the product,

product:

Recursion mythology

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1,

product: 1

Recursion mythology

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1, multiply it by 4,

product:	4

```
f(4) = 4 * f(3)
```

Recursion mythology

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1, multiply it by 4, then multiply that value by 3,

product: 12 `f(4) = 4 * 3 * f(2)`

Recursion mythology

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1, multiply it by 4, then multiply that value by 3, and then multiply that value by 2,

product: 24 `f(4) = 4 * 3 * 2 * f(1)`

Recursion mythology

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1, multiply it by 4, then multiply that value by 3, and then multiply that value by 2, and finally multiply that value by 1 to give the result of the function call `factorial(4)`:

product: 24 `f(4) = 4 * 3 * 2 * 1`

Recursion mythology

In other words, I could start with an accumulator "variable" to hold the product, initialize it to 1, multiply it by 4, then multiply that value by 3, and then multiply that value by 2, and finally multiply that value by 1 to give the result of the function call `factorial(4)`:

How do we make this happen?

product:	24

Tail recursion

This type of recursion is called tail recursion, and it often involves the introduction of an additional parameter used as a "variable" to hold the partially-computed result instead of storing postponed computations on the stack.

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

fact_aux is an auxiliary or helper function to keep the syntax of fact(n) the same as before. The recursive function is really fact_aux, not fact. Note that fact_aux has no postponed or pending computations on return from recursive calls. All the work is done “inside” the recursive call in the tail call position. No work is done “outside” the recursive call in that tail call position.

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
```

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
fact_aux(4, 1)
```

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
fact_aux(4, 1)
fact_aux(3, 4)
```

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
fact_aux(4, 1)
fact_aux(3, 4)
fact_aux(2, 12)
```

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
fact_aux(4, 1)
fact_aux(3, 4)
fact_aux(2, 12)
fact_aux(1, 24)
```

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
fact_aux(4, 1)
fact_aux(3, 4)
fact_aux(2, 12)
fact_aux(1, 24)
24
```

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
fact_aux(4, 1)
fact_aux(3, 4)
fact_aux(2, 12)
fact_aux(1, 24)
24
```

Using the substitution model, we don’t see any growth to the right, indicating there’s no postponed computations. But does that mean there’s no stack growth? We’re still making function calls and pushing stack frames, aren’t we?

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
fact_aux(4, 1)
fact_aux(3, 4)
fact_aux(2, 12)
fact_aux(1, 24)
24
```

A function is tail recursive if for any recursive call in the function, that call is the absolute last thing the function needs to do before returning.

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
fact_aux(4, 1)
fact_aux(3, 4)
fact_aux(2, 12)
fact_aux(1, 24)
24
```

In that case, we don’t really need to push a new activation frame, do we? There’s nothing new to remember. We could just re-use the old frame. (Or we could pop the old frame and push the new frame. Either way, there’s no stack growth.)

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
fact_aux(4, 1)
fact_aux(3, 4)
fact_aux(2, 12)
fact_aux(1, 24)
24
```

Many compilers will do exactly that: reuse the frame. By definition, Scheme (and therefore Racket) must do tail call optimization. Some versions of Java will, but others won't. The gcc C++ compiler does, if you switch on optimization.

Tail recursion

For our purposes in CPSC 221, don't use tail recursion because you think it will get you a more efficient C++ program. Do use tail recursion if it makes more sense to you than other approaches.

Recursion: so misunderstood (sigh)

"Recursion isn't useful very often, but when used judiciously it produces exceptionally elegant solutions.... In general, recursion leads to small code and slow execution and chews up stack space. For a small group of problems, recursion can produce simple, elegant solutions. For a slightly larger group of problems, it can produce simple, elegant, hard-to-understand solutions. For most problems, it produces massively complicated solutions -- in those cases, simple iteration is usually more understandable. Use recursion selectively."

Steve McConnell in Code Complete

Recursion: so misunderstood (sigh)

"Recursion isn't useful very often, but when used judiciously it produces exceptionally elegant solutions.... In general, recursion leads to small code and slow execution and chews up stack space."

As you've just seen, optimizing compilers and tail recursion can make this problem go away. You need to have a better understanding or recursion than just the reflexive response of "Eek! Recursion! I don't get it."

Recursion: so misunderstood (sigh)

More from McConnell:

If a programmer who worked for me used recursion to compute a factorial, I'd hire someone else. Here's the recursive version of the factorial routine... (in Pascal)

```
Function Factorial( Number: integer ): integer;
begin
   if ( Number = 1 ) then
      Factorial := 1
   else
      Factorial := Number * Factorial( Number - 1);
end;
```

Recursion: so misunderstood (sigh)

More from McConnell:

In addition to being slow and making the use of run-time memory unpredictable, the recursive version of this routine is harder to understand than the iterative version. Here's the iterative version:

```
Function Factorial( Number: integer ): integer;
var
   IntermediateResult: integer;
   Factor:             integer;
begin
   IntermediateResult := 1;
   for Factor := 2 to Number do
      IntermediateResult := IntermediateResult * Factor;
   Factorial := IntermediateResult;
end;
```

Recursion: so misunderstood (sigh)

Things to consider:

1. You now know that the recursive version isn't necessarily slow and doesn't necessarily chew up memory.

2. There may be reasons for employing recursion that are more important than efficiency. Like ease of implementation and understandability.

3. Understandability, like beauty, is in the eye of the beholder.

Recursion: so misunderstood (sigh)

The authors of your textbook are much more accepting:

Generally, if it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive function, because the reduction in efficiency does not outweigh the advantage of readable code that is easy to debug.

Koffman and Wolfgang, p. 416

Though they may be accepting, even they don't fully understand...

Recursion: so misunderstood (sigh)

From Objects, Abstraction, Data Structures and Design Using C++ by Koffman and Wolfgang (also on p. 416):

In [tail-recursive] algorithms, there is a single recursive call and it is the last line of the function. An example...

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

Can you see what's wrong here?

Recursion: so misunderstood (sigh)

From Objects, Abstraction, Data Structures and Design Using C++ by Koffman and Wolfgang (also on p. 416):

In [tail-recursive] algorithms, there is a single recursive call and it is the last line of the function. An example...

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

Can you see what's wrong here? This is the function from our first recursion example, and it's not tail recursive.