INFO1103: Introduction to Programming

School of Information Technologies, University of Sydney

Week 11: Recursion, Collections

We will cover: Alternative problem solving strategy, Defining and using collections

You should read: §§11.1, 11.2, 12.1

Lecture 21: Recursion

Solving problems in terms of solutions to simpler problems

Control flow

Revision: what is control flow?

What happens when you call method A(), which calls method B()

```java
public void B() {
    System.out.print("B");
}
public void A() {
    B();
    System.out.print("A");
}
public void main() {
    A();
}
```

What is the output?

Draw a diagram to describe the method calls

Control flow

```java
public void C() {
    System.out.print("C");
}
public void B() {
    System.out.print("B");
}
public void A() {
    B();
    System.out.print("A");
}
public void main() {
    A();
    C();
}
```

Is the output ABC?

Can you draw this?

Control flow

```java
public void C() {
    System.out.print("C");
    A(); // added this
}
public void B() {
    System.out.print("B");
}
public void A() {
    B();
    System.out.print("A");
}
public void main() {
    A();
    C();
}
```

What does this look like?

Control flow

```java
public void G() { System.out.print("G"); }
public void F() { System.out.print("F"); }
public void E() { System.out.print("E"); }
public void C() { System.out.print("C"); }
public void D() {
    E();
    System.out.print("D");
}
public void B() {
    C();
    D()
    F();
    System.out.print("B");
}
public void A() {
    B();
    G();
    System.out.print("A");
}
```

Draw this...What is the output?

Recursion

What is it?
- A technique to solve a problem.

Identifying when a problem can be solved with recursion
- An arbitrary sized problem can be solved by first solving a smaller version of that problem

What is the relationship between the arbitrary sized problem and the smallest problem?
- they are both solved with the **same method** (code)

```
public ?? solver( ??? ) {
    ...
    ?? = solver( ??? );
    ...
    return ??;
}
```

Recursion for a person

Suppose there is the following situation:

- Every person has an age
- Every person can have exactly zero or one child
- Every person can report the total age in years of themselves and their child

Task: Find the total age of one person and all of their descendants

Ingredients of Recursion

Recursion has two essential ingredients:

1. a base case (somewhere to stop)
2. the recurrence relation (a way to continue)

The *base case* is also known as the “trivial” case and is the case where we don’t need to use the recurrence relation to get the answer.

The recurrence relation tells you how to get from a more complex case to a simpler one.
Without the

- *recurrence*
- *base case*

Ingredients of Recursion

Recursion has two essential ingredients:

1. a base case (somewhere to stop)
2. the recurrence relation (a way to continue)

The *base case* is also known as the “trivial” case and is the case where we don’t need to use the recurrence relation to get the answer.

The recurrence relation tells you how to get from a more complex case to a simpler one.
Without the

- *recurrence* you can’t proceed
- *base case*

Ingredients of Recursion

Recursion has two essential ingredients:

1. a base case (somewhere to stop)
2. the recurrence relation (a way to continue)

The *base case* is also known as the “trivial” case and is the case where we don’t need to use the recurrence relation to get the answer.

The recurrence relation tells you how to get from a more complex case to a simpler one.
Without the

- *recurrence* you can’t proceed
- *base case* you can’t stop

Recursion for a person continued

Each person now has a first name

Task 2: For a given Person, print each name of the descendants starting from oldest to youngest

Task 3: For a given Person, print the names of the next 3 generations only

Task 4: For a given Person, print the generation number where the child contains the name of the parent. e.g. George is father of Thomas who is father of Thomas Junior print "Generation 1"

Recursion with Matryoshka Doll

Familiar example

Define a class MatryoshkaDoll (Russian Doll)
A Matryoshka Doll contains zero or one Matryoshka Doll
A Matryoshka Doll has a size, 0 being the smallest

Task 1: For a given MatryoshkaDoll: count the number of dolls inside

Task 2: For a given MatryoshkaDoll: add up all the sizes of all dolls inside

Task 3: For a given MatryoshkaDoll: increase the size of the inner most doll by one, then increase the outer doll size by one (inner must be first!)

Task 4: For a given MatryoshkaDoll: increase the size of the inner most doll by `n`, then increase the outer doll size to at most `n + 1`

Recursion with method parameters

More useful is when the same thing eventually ends

```
public void foo(int n) {
    if (n == 0)
        return
    System.out.print("Hello");
    foo(n - 1);
    System.out.print("Bye");
}
```

What is the output for foo(5)?

Every time this happens a new copy of the function is loaded into memory, with its own copies of local variables and arguments.

Recursion more examples

Something that allows you to define something in terms of itself, e.g.,

- *factorial*: $n! = n \cdot (n-1)!$ (a function)
- *Fibonacci sequence*: $F_n = F_{n-1} + F_{n-2}$ (a sequence of numbers)
- *list*: a list either *empty*, or is an *item* followed by a *list* (a definition)
- *rooted binary tree* (a definition)
  - T = *root* {+ left subtree TL} {+ right subtree TR}
  - where TL and TR are rooted binary trees

Examples of recursion

What do these recursive functions do?

```
F ($n$)
if ($n = 1$) then
    return(1)
else
    return(F($n-2$))
```

$$f(n,k) = \begin{cases} n \text{ if } n < k; \\ f(n-k,k) \text{ otherwise} \end{cases}$$

Examples of recursion

What do these recursive functions do?

```
F $(n)$
if $(n = 1)$ then
    return(1)
else
    return(F$(n-2)$)
```

$$f(n,k) = \begin{cases} n \text{ if } n < k; \\ f(n-k,k) \text{ otherwise} \end{cases}$$

return 1 if n is odd
& die horribly otherwise: it's an infinite loop!

Examples of recursion

What do these recursive functions do?

```
F ($n$)
if ($n = 1$) then
    return(1)
else
    return(F($n-2$))
```

return 1 if n is odd
& die horribly otherwise: it's an infinite loop!

$$f(n,k) = \begin{cases} n \text{ if } n < k; \\ f(n-k,k) \text{ otherwise} \end{cases}$$

return the remainder of n/k

What happens inside a recursive method

In the next part we'll look at how a function calls itself (with different arguments) to calculate a recursive function.

Tracing a factorial method

Suppose you have a function factorial, called `fac`.
The factorial is a function that is defined like this:

$$f(n) = n! = n \times (n-1)! = n \times f(n-1), n > 0; f(0) = 1$$

Tracing a factorial method

Suppose you have a function factorial, called `fac`.
The factorial is a function that is defined like this:

$$f(n) = n! = n \times (n-1)! = n \times f(n-1), n > 0; f(0) = 1$$

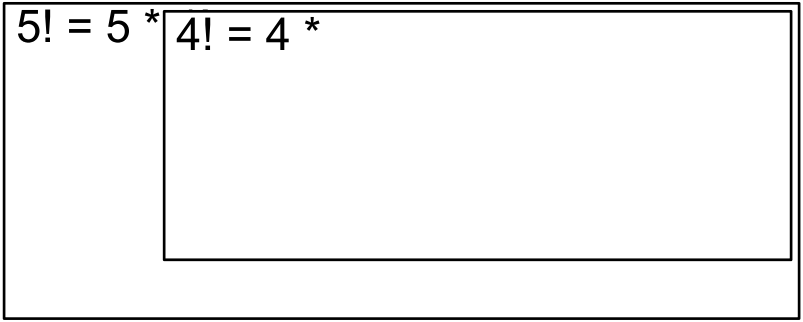
5! = 5 * 4!

one copy of the method is in memory: we next need to call it with argument 4

Tracing a factorial method

Suppose you have a function factorial, called `fac`.
The factorial is a function that is defined like this:

$$f(n) = n! = n \times (n-1)! = n \times f(n-1), n > 0; f(0) = 1$$

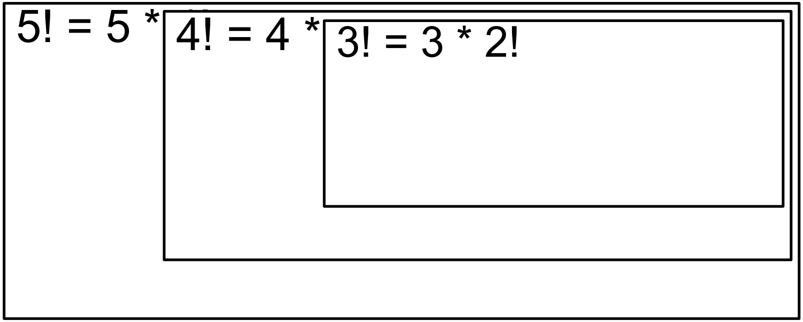


two copies of the method are in memory, each with its own argument

Tracing a factorial method

Suppose you have a function factorial, called `fac`.
The factorial is a function that is defined like this:

$$f(n) = n! = n \times (n-1)! = n \times f(n-1), n > 0; f(0) = 1$$

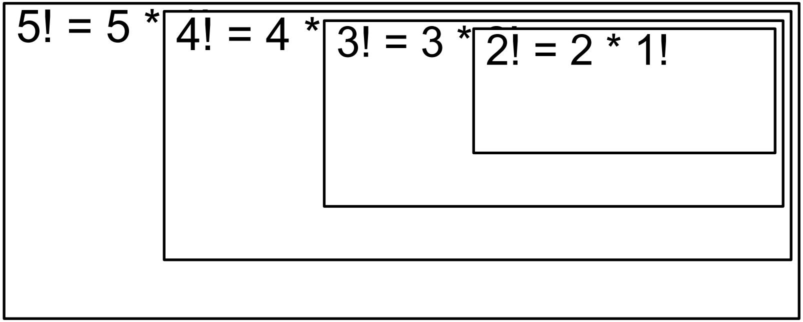


three copies of the method are in memory, with arguments 5, 4 and 3: next call `fac`(2)

Tracing a factorial method

Suppose you have a function factorial, called `fac`.
The factorial is a function that is defined like this:

$$f(n) = n! = n \times (n-1)! = n \times f(n-1), n > 0; f(0) = 1$$

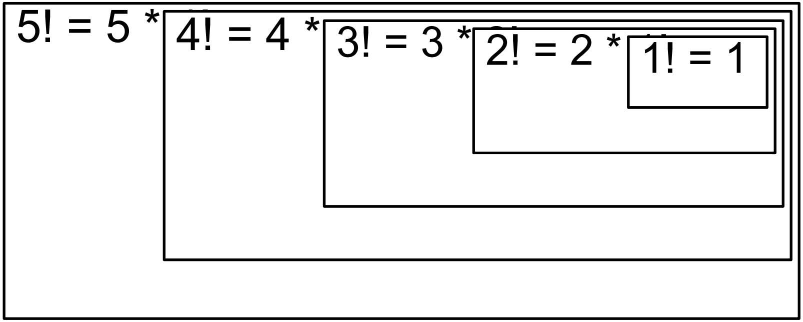


four copies…

Tracing a factorial method

Suppose you have a function factorial, called `fac`.
The factorial is a function that is defined like this:

$$f(n) = n! = n \times (n-1)! = n \times f(n-1), n > 0; f(0) = 1$$

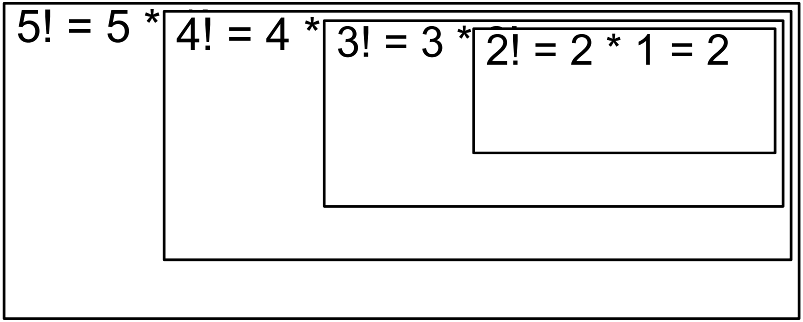


five copies. We've reached the *base case* and we're done calling `fac` on different arguments.

Tracing a factorial method

Suppose you have a function factorial, called `fac`.
The factorial is a function that is defined like this:

$$f(n) = n! = n \times (n-1)! = n \times f(n-1), n > 0; f(0) = 1$$

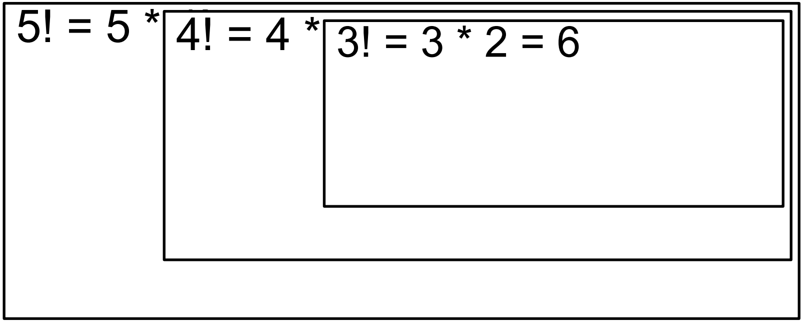


now we can work out `fac(2)` and we're back to 4 copies of the method in memory

Tracing a factorial method

Suppose you have a function factorial, called `fac`.
The factorial is a function that is defined like this:

$$f(n) = n! = n \times (n-1)! = n \times f(n-1), n > 0; f(0) = 1$$

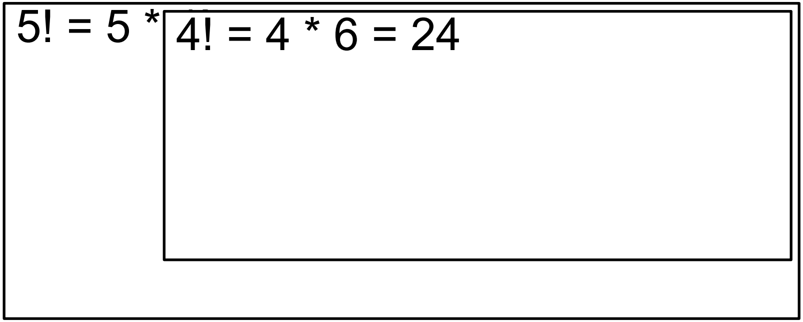


`fac(3)` is now calculable and we have released the memory required for calling `fac(2)`

Tracing a factorial method

Suppose you have a function factorial, called `fac`.
The factorial is a function that is defined like this:

$$f(n) = n! = n \times (n-1)! = n \times f(n-1), n > 0; f(0) = 1$$



Two copies of `fac` in memory and we can calculate 4! = 24

Tracing a factorial method

Suppose you have a function factorial, called `fac`.
The factorial is a function that is defined like this:

$$f(n) = n! = n \times (n-1)! = n \times f(n-1), n > 0; f(0) = 1$$

5! = 5 * 24 =120

Finally we have just one copy of `fac` in memory and can return $5! = 120$.

Writing recursive code

Ok so how do we write a recursive method?

This is MUCH simpler than it sounds, and often turns out to be much simpler than writing it “the other way”, i.e., *iteratively*.

Writing a recursive method, you must

- know what the recurrence relation and base cases are
- check whether the base case has been reached
- if it hasn’t been reached, call the same function with different arguments

Factorial recursion

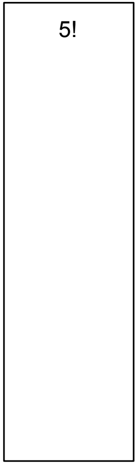
```java
package recursion;

public class Factorial {

    public int fac(int n) {
        if (n <= 0) {
            System.out.println("base case: 0! = 1");
            return 1;
        }
        System.out.println("recursive case: n=" + n);
        return n*fac(n-1);
    }

    public static void main(String[] args) {
        Factorial f = new Factorial();
        int n = 6;
        System.out.println(n + "! = " + f.fac(n));
    }
}
```

running Factorial

```
recursive case: n=6
recursive case: n=5
recursive case: n=4
recursive case: n=3
recursive case: n=2
recursive case: n=1
base case: 0! = 1
6! = 720
```

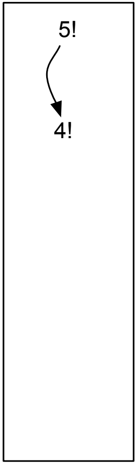
Recursion Tree

The way we call recursive methods can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:



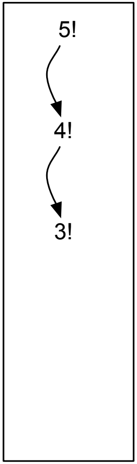
Recursion Tree

The way we call recursive methods can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:



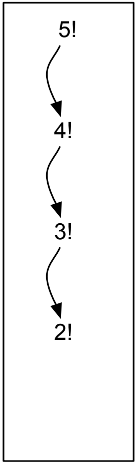
Recursion Tree

The way we call recursive methods can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:



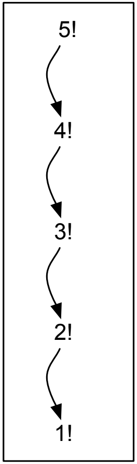
Recursion Tree

The way we call recursive methods can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:



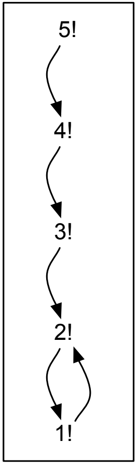
Recursion Tree

The way we call recursive methods can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:



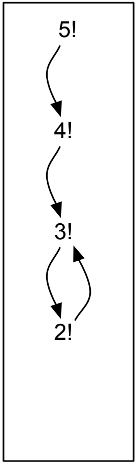
Recursion Tree

The way we call recursive methods can be represented as a *tree*.
Here’s a very simple one for the case of a recursive factorial:



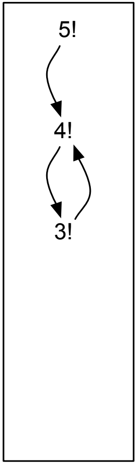
Recursion Tree

The way we call recursive methods can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:



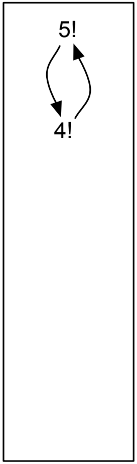
Recursion Tree

The way we call recursive methods can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:



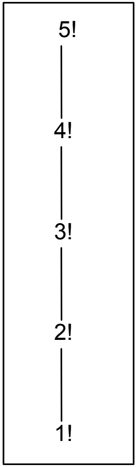
Recursion Tree

The way we call recursive methods can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:

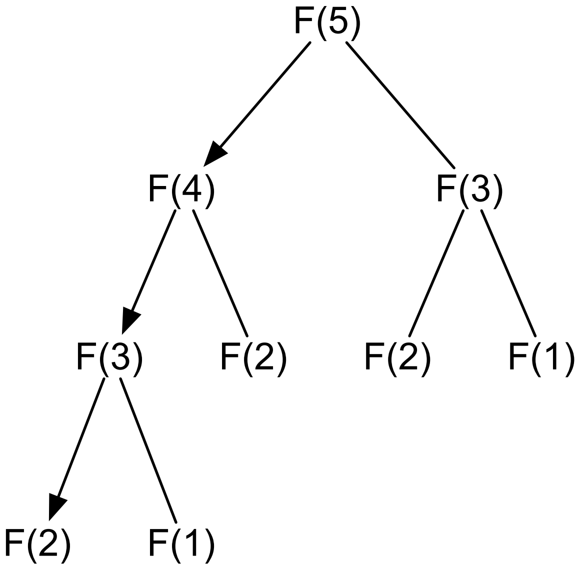


Recursion Tree

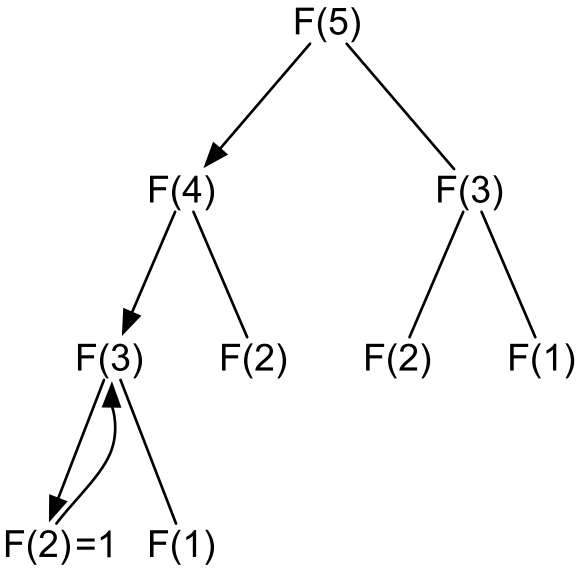
The way we call recursive methods can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:



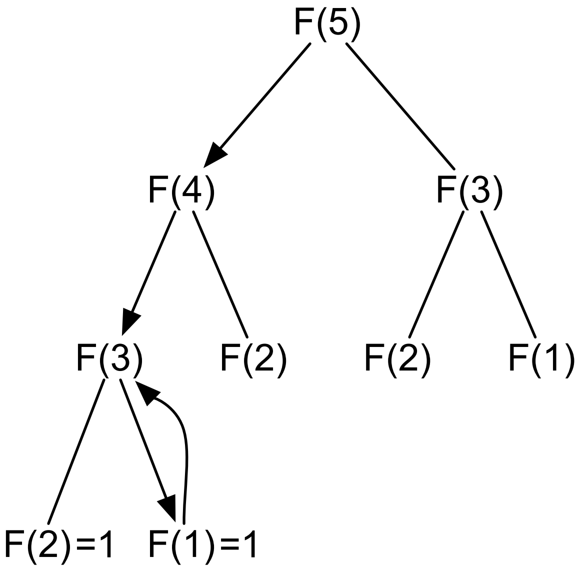
Recursion tree for Fibonacci



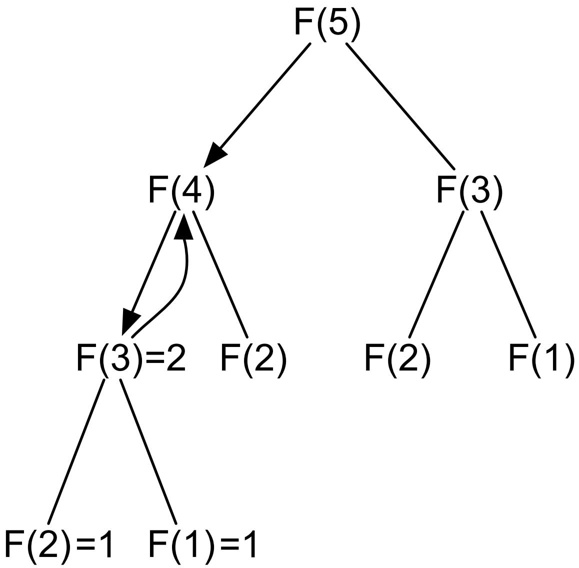
Recursion tree for Fibonacci



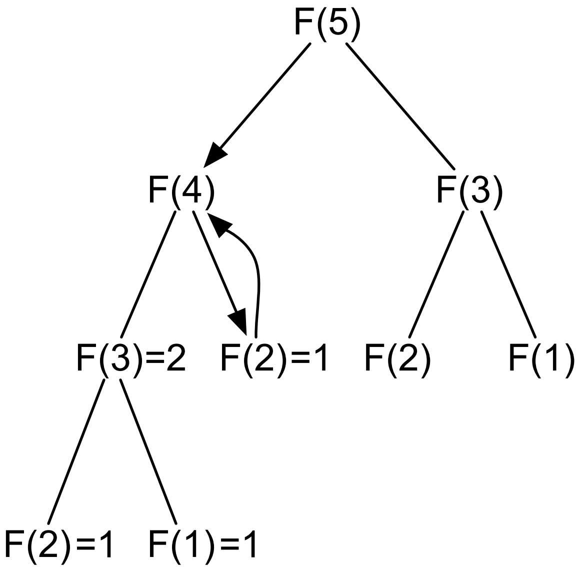
Recursion tree for Fibonacci



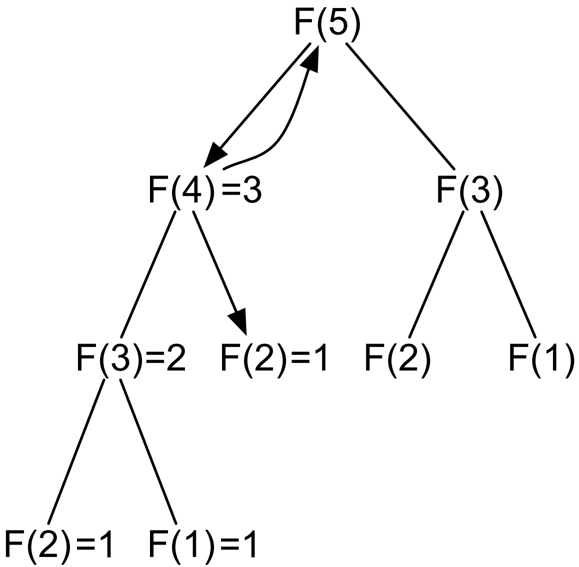
Recursion tree for Fibonacci



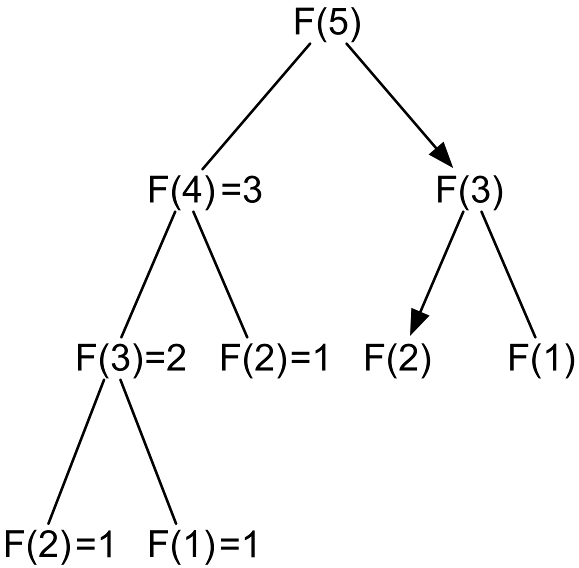
Recursion tree for Fibonacci



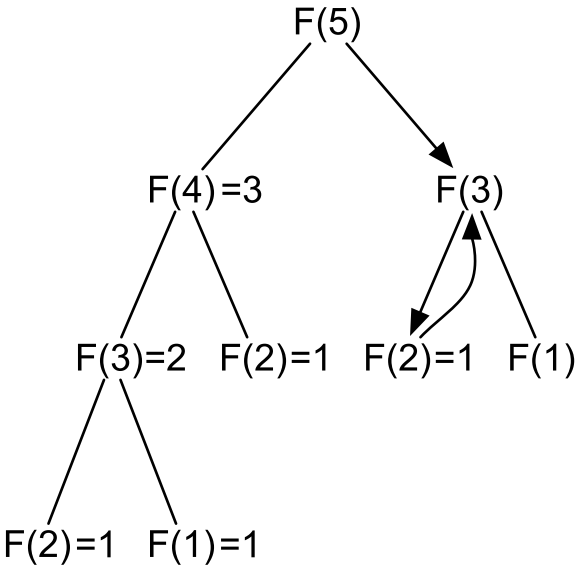
Recursion tree for Fibonacci



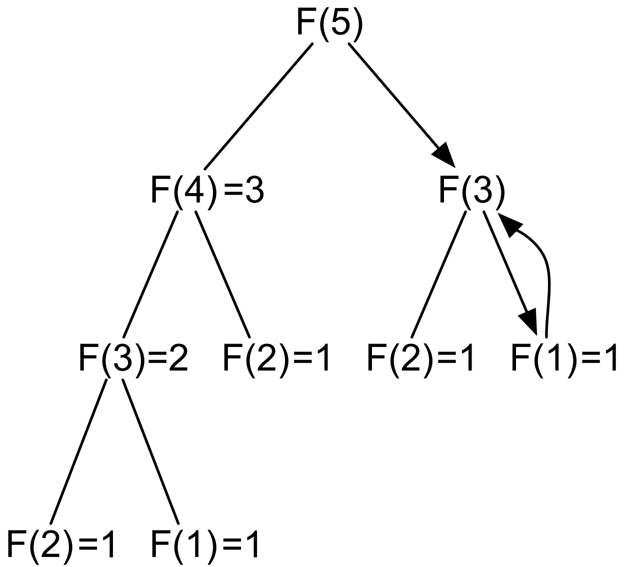
Recursion tree for Fibonacci



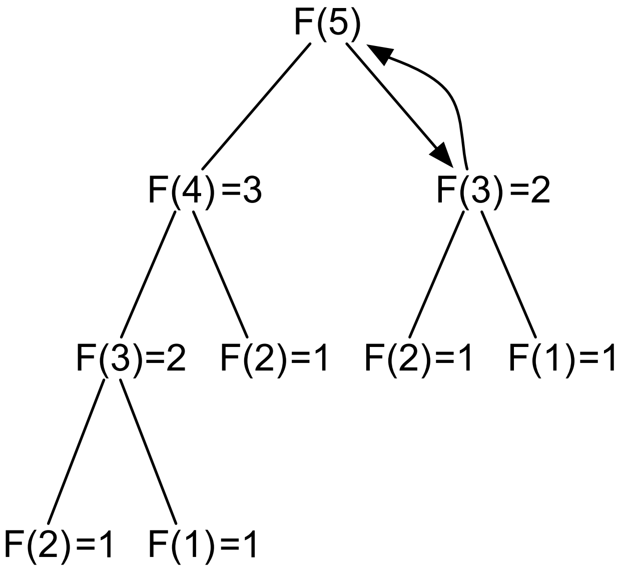
Recursion tree for Fibonacci



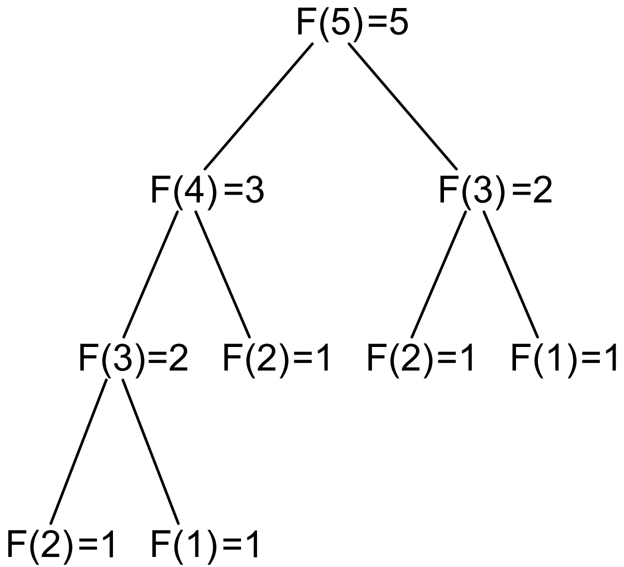
Recursion tree for Fibonacci



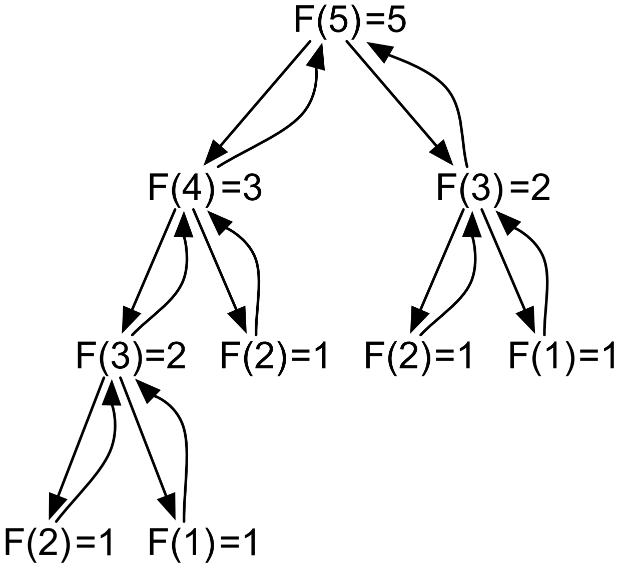
Recursion tree for Fibonacci



Recursion tree for Fibonacci



Recursion tree for Fibonacci



Recursion tree properties

Each node in the tree is a method call.

The number of nodes in the tree is the number of *method calls*, so the amount of time taken to work through the whole tree is proportional to *the number of nodes in the tree*.

The *height* of the tree is the maximum number of copies of the method in memory at any time, so this corresponds to the maximum memory requirement of the recursion.

Important point!

Recursion can be very useful:

- code is usually clean
- easy to write/read/understand,
- some problems are much harder to solve in other ways.

But recursion is *not always ideal*:

Recursive code may be very inefficient (and this can be hard to spot).

Recursion is *never essential*: with enough effort, it can be replaced by *iteration*.

Recursion can have multiple variables, e.g., the binomial

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Binomial recurrence

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

The binomial coefficients appear in the expansion of $(a+b)^n$:

$$(a+b)^n = a^n + \binom{n}{1} a^{n-1} b + \binom{n}{2} a^{n-2} b^2 + \ldots \binom{n}{k} a^{n-k} b^k + \ldots + \binom{n}{n-1} a^1 b^{n-1} + b^n$$

…and are often seen in Pascal’s Triangle.

Pascal's triangle

```
                                    1
                                1       1
                            1       2       1
                        1       3       3       1
                    1       4       6       4       1
                1       5       10      10      5       1
            1       6       15      20      15      6       1
        1       7       21      35      35      21      7       1
    1       8       28      56      70      56      28      8       1
1       9       36      84      126     126     84      36      9       1
```

This is an extension exercise in the labs this week: have a go at making your own beautifully laid out Pascal's triangle ☺

The non-recursive way to recursive methods

I noted before that it's never necessary to use recursive code, but how would write something like methods for factorial or Fibonacci, *without* using recursive function calls?

Let's look at these cases separately:
Case 1: Factorial:

$$n! = n(n-1)! = n(n-1)(n-2)! = \ldots = n(n-1)(n-2), \ldots (2)(1)$$

This should show you how to code factorial with an *iterative* method…

The non-recursive way to recursive methods (cont.)

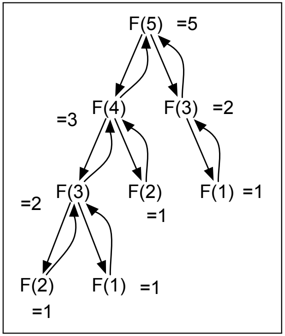
```java
package iterative;

public class NonRecursiveFactorial {
    public int fac(int n) {
        int prod = 1;
        for (int i = n; i > 1; i--) {
            prod *= i;
        }
        return prod;
    }

    public static void main(String[] args) {
        NonRecursiveFactorial nrf = new NonRecursiveFactorial();
        System.out.println(nrf.fac(10));
    }
}
```

Non-recursive Fibonacci

Remember the Fibonacci sequence is defined by

$$F(n) = F(n-1) + F(n-2); F(1) = F(2) = 1$$

Think about the recursion tree for F(5):



You should see that some values are used multiple times: this method is therefore calling `F()` with the same arguments, e.g., 1, 2, 3, more times than it needs to. This effect gets really bad really quickly: try a recursive Fibonacci method when n = 40 and you'll see what I mean!

Fibonacci

First, let's look at the recursive method for the Fibonacci sequence:

$$F(n) = F(n-1) + F(n-2); F(1) = F(2) = 1$$

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

Fibonacci (cont.)

And just for fun I'm going to see how much time each recursive code takes to call.

```java
    public static void main(String[] args) {
        Fibonacci f = new Fibonacci();
        Date start = new Date();
        for (int i = 0; i <= 45; ++i) {
            Date now = new Date();
            System.out.print("F(" + i + ") = " + f.fib(i));
            System.out.println("\t(" +
                    (now.getTime() - start.getTime()) + "ms)");
                    // getTime() returns the number of milliseconds
                    // that have passed since the end of the 60s
        }
```

Running Fibonacci.java

```
F(0) = 1    (0ms)
F(1) = 1    (0ms)
F(2) = 1    (0ms)
F(3) = 2    (0ms)
F(4) = 3    (0ms)
F(5) = 5    (0ms)
F(6) = 8    (0ms)
F(7) = 13    (0ms)
F(8) = 21    (0ms)
F(9) = 34    (0ms)
F(10) = 55    (0ms)
F(11) = 89    (1ms)
F(12) = 144    (1ms)
F(13) = 233    (1ms)
F(14) = 377    (1ms)
F(15) = 610    (1ms)
F(16) = 987    (1ms)
F(17) = 1597    (1ms)
F(18) = 2584    (1ms)
```

Running Fibonacci.java (cont.)

```
F(19) = 4181   (2ms)
F(20) = 6765   (2ms)
F(21) = 10946   (2ms)
F(22) = 17711   (3ms)
F(23) = 28657   (5ms)
F(24) = 46368   (7ms)
F(25) = 75025   (11ms)
F(26) = 121393   (12ms)
F(27) = 196418   (13ms)
F(28) = 317811   (13ms)
F(29) = 514229   (14ms)
F(30) = 832040   (16ms)
F(31) = 1346269   (18ms)
F(32) = 2178309   (23ms)
F(33) = 3524578   (30ms)
F(34) = 5702887   (40ms)
F(35) = 9227465   (58ms)
F(36) = 14930352   (84ms)
F(37) = 24157817   (128ms)
```

Running `Fibonacci.java` (cont.)

```
      F(38) = 39088169   (196ms)
      F(39) = 63245986   (309ms)
      F(40) = 102334155   (481ms)
      F(41) = 165580141   (762ms)
      F(42) = 267914296   (1222ms)
      F(43) = 433494437   (1950ms)
      F(44) = 701408733   (3095ms)
      F(45) = 1134903170   (4956ms)
```

Youch.

NRFibonacci

```
    public int fib(int n) {
        if (n < 3) {
            return 1;
        }
        int[] F = new int[n+1];
        F[0] = F[1] = F[2] = 1; // yes, this works
        for (int i = 3; i <= n; ++i) {
            F[i] = F[i-1] + F[i-2];
        }
        return F[n];
    }
```

Running NRFibonacci

With approximately the same `main` method as before, we have the following output:

```
F(0) = 1   (0ms)
F(1) = 1   (0ms)
F(2) = 1   (0ms)
F(3) = 2   (0ms)
F(4) = 3   (0ms)
F(5) = 5   (0ms)
F(6) = 8   (1ms)
F(7) = 13   (1ms)
F(8) = 21   (1ms)
F(9) = 34   (1ms)
F(10) = 55   (1ms)
F(11) = 89   (1ms)
F(12) = 144   (1ms)
F(13) = 233   (1ms)
F(14) = 377   (1ms)
F(15) = 610   (1ms)
```

Running `NRFibonacci` (cont.)

```
F(16) = 987   (1ms)
F(17) = 1597   (1ms)
F(18) = 2584   (1ms)
F(19) = 4181   (1ms)
F(20) = 6765   (1ms)
F(21) = 10946   (1ms)
F(22) = 17711   (1ms)
F(23) = 28657   (1ms)
F(24) = 46368   (1ms)
F(25) = 75025   (1ms)
F(26) = 121393   (1ms)
F(27) = 196418   (1ms)
F(28) = 317811   (1ms)
F(29) = 514229   (1ms)
F(30) = 832040   (1ms)
F(31) = 1346269   (1ms)
F(32) = 2178309   (2ms)
F(33) = 3524578   (2ms)
F(34) = 5702887   (2ms)
```

Running NRFibonacci (cont.)

```
F(35) = 9227465   (2ms)
F(36) = 14930352   (2ms)
F(37) = 24157817   (2ms)
F(38) = 39088169   (2ms)
F(39) = 63245986   (2ms)
F(40) = 102334155   (2ms)
F(41) = 165580141   (2ms)
F(42) = 267914296   (2ms)
F(43) = 433494437   (2ms)
F(44) = 701408733   (2ms)
F(45) = 1134903170   (2ms)
```

Recursion: a summary

- recursive methods have a base case and a recurrence relation;
- recursion is never necessary: the alternative is iteration
- recursion can be very inefficient
- the recursion tree indicates approximately how much time is taken and how much space is required for a recursive method