

INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney



COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Week 11: Recursion, Further classes and methods

We will cover: Alternative problem solving strategy

You should read: §§2.3 pg290-299

Lecture 21: Recursion

Solving problems in terms of solutions to simpler problems

Revision: what is control flow?

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

```
1 def B():  
2     print("B")  
3  
4 def A():  
5     B()  
6     print("A")  
7  
8 def main():  
9     A()
```

What is the output?

Draw a diagram to describe the function calls

Control flow

```
1 def C():  
2     print("C")  
3  
4 def B():  
5     print("B")  
6  
7 def A():  
8     B()  
9     print("A")  
10  
11 def main():  
12     A()  
13     C()
```

Is the output ABC?

Can you draw this?

```
1 def C():
2     print("C")
3     A() # added this
4
5 def B():
6     print("B")
7
8 def A():
9     B()
10    print("A")
11
12 def main():
13     A()
14     C()
```

What does this look like?

Control flow

```
1 def G(): print("G")
2 def F(): print("F")
3 def E(): print("E")
4 def C(): print("C")
5 def D():
6     E()
7     print("D")
8
9 def B():
10    C()
11    D()
12    F()
13    print("B")
14
15 def A():
16    B()
17    G()
18    print("A")
```

Draw this...What is the output?

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 function** (code)

```
1 def solver( ??? ) :  
2     ...  
3     ?? = solver( ??? )  
4     ...  
5     return ??
```

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

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 function parameters

More useful is when the same thing eventually ends

```
1 def foo(n):  
2     if n == 0:  
3         return  
4     print("Hello")  
5     foo(n - 1)  
6     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.

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)
 - $T = \text{root} \{+ \text{left subtree TL}\} \{+ \text{right subtree TR}\}$
 - where TL and TR are rooted binary trees

Examples of recursion

What do these recursive functions do?

F (*n*)

```
1  if (n = 1) then  
2    · return(1)  
3  else  
4    · 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?

$F(n)$

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

return 1 if n is odd AND die horribly otherwise: it is infinite recursion!

Examples of recursion

What do these recursive functions do?

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

return the remainder of n/k

What happens inside a recursive function

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

Tracing a factorial function

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 function

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 function is in memory: we next need to call it with argument 4

Tracing a factorial function

Suppose you have a function factorial, called `fac`.

The factorial is a function that is defined like this:

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


$$5! = 5 * 4! = 4 *$$

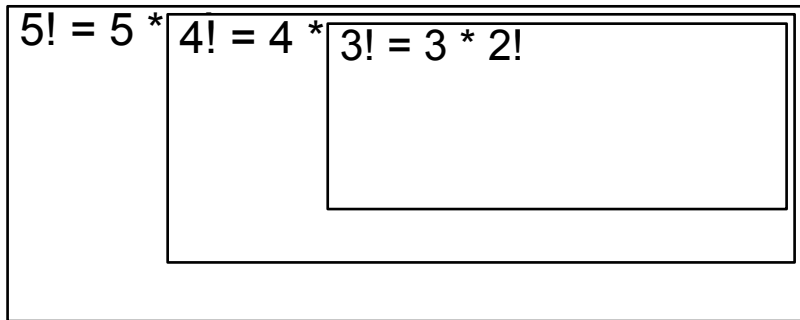
two copies of the function are in memory, each with its own argument

Tracing a factorial function

Suppose you have a function factorial, called `fac`.

The factorial is a function that is defined like this:

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



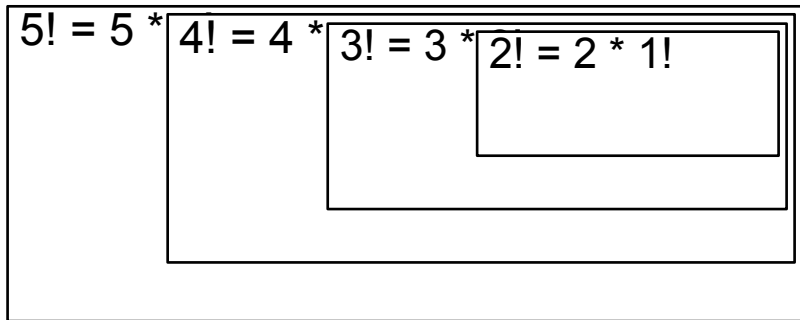
three copies of the function are in memory, with arguments 5, 4 and 3: next call `fac(2)`

Tracing a factorial function

Suppose you have a function factorial, called `fac`.

The factorial is a function that is defined like this:

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



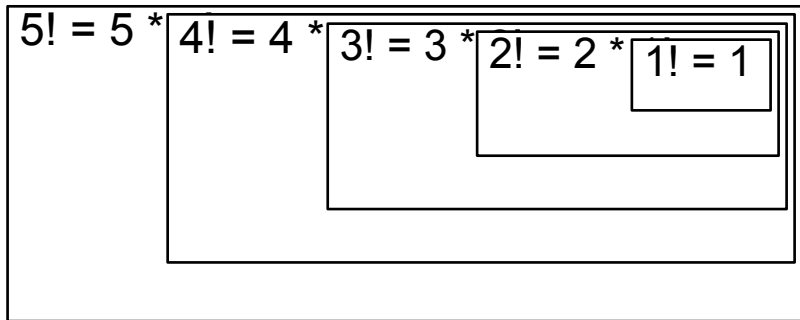
four copies...

Tracing a factorial function

Suppose you have a function factorial, called `fac`.

The factorial is a function that is defined like this:

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



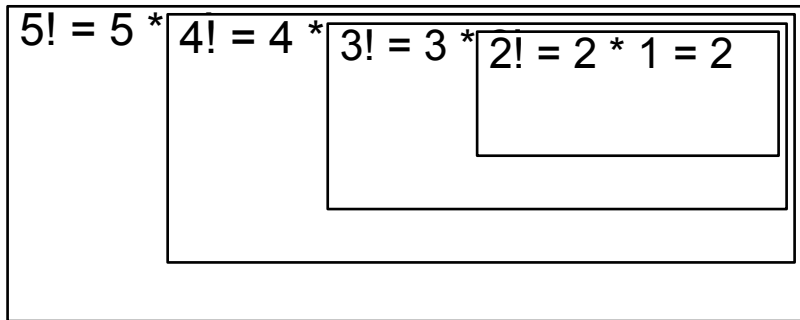
five copies. We've reached the *base case* and we're done calling `fac` on different arguments.

Tracing a factorial function

Suppose you have a function factorial, called `fac`.

The factorial is a function that is defined like this:

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



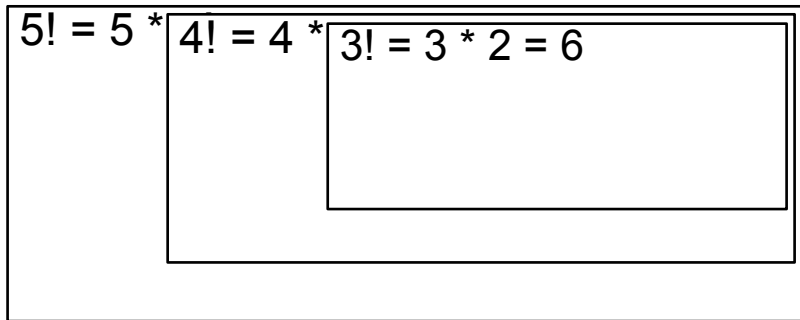
now we can work out `fac(2)` and we're back to 4 copies of the function in memory

Tracing a factorial function

Suppose you have a function factorial, called `fac`.

The factorial is a function that is defined like this:

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



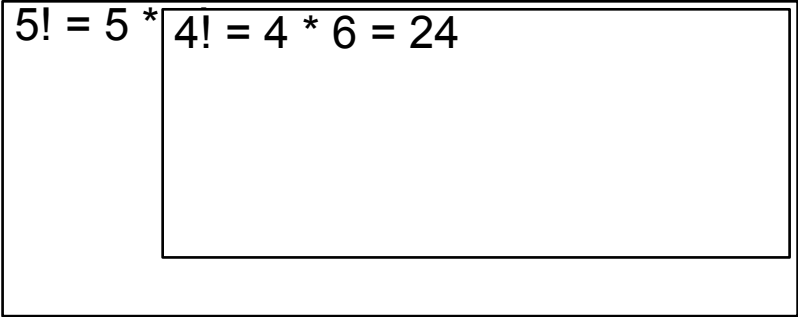
`fac(3)` is now calculable and we have released the memory required for calling `fac(2)`

Tracing a factorial function

Suppose you have a function factorial, called `fac`.

The factorial is a function that is defined like this:

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


$$5! = 5 * 4! = 4 * 6 = 24$$

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

Tracing a factorial function

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 $5! = 120$.

Ok so how do we write a recursive function?

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 function, 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

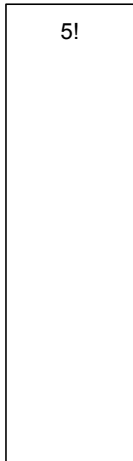
```
1 def fac(n):
2     if n <= 0:
3         print("base case: 0! = 1")
4         return 1
5
6     print("recursive case: n=" + str(n))
7     return n * fac(n - 1)
8
9 n = 6
10 result = fac(n)
11 print("{}! = {}".format(n, result))
```

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
```

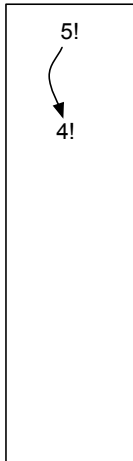
Recursion Tree

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

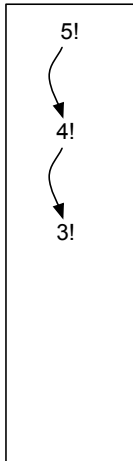


Recursion Tree

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

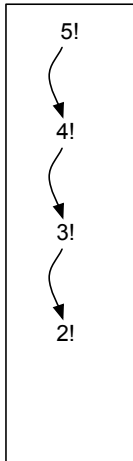


The way we call recursive functions can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:



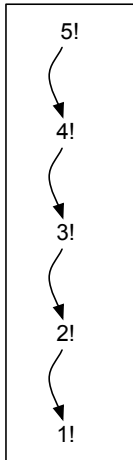
Recursion Tree

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



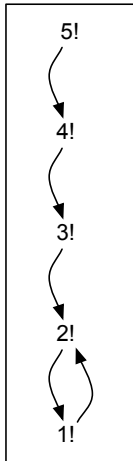
Recursion Tree

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



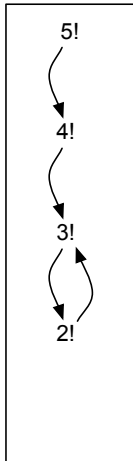
Recursion Tree

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



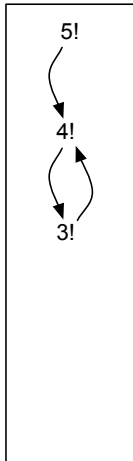
Recursion Tree

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



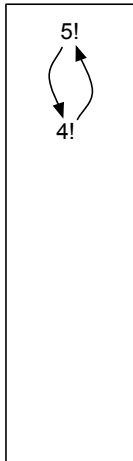
Recursion Tree

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



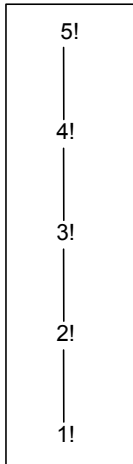
Recursion Tree

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

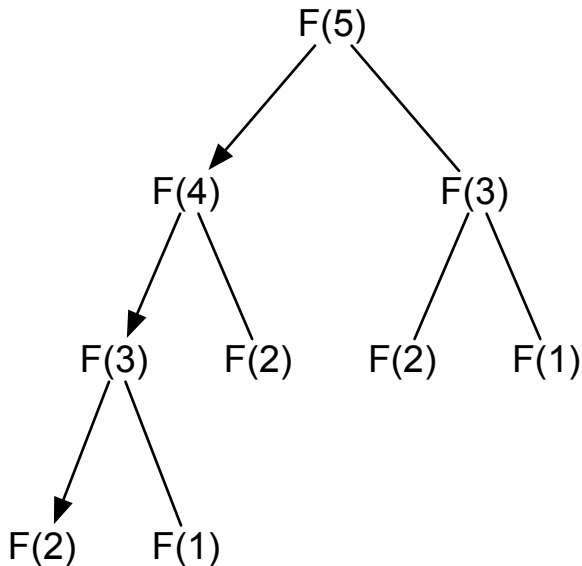


Recursion Tree

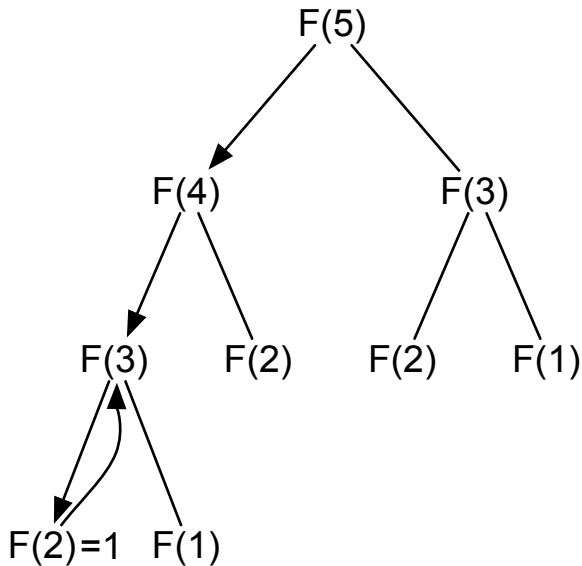
The way we call recursive functions can be represented as a *tree*.
Here's a very simple one for the case of a recursive factorial:



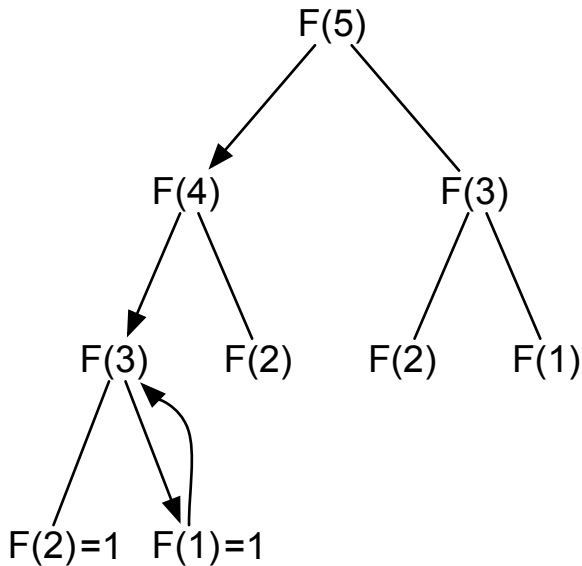
Recursion tree for Fibonacci



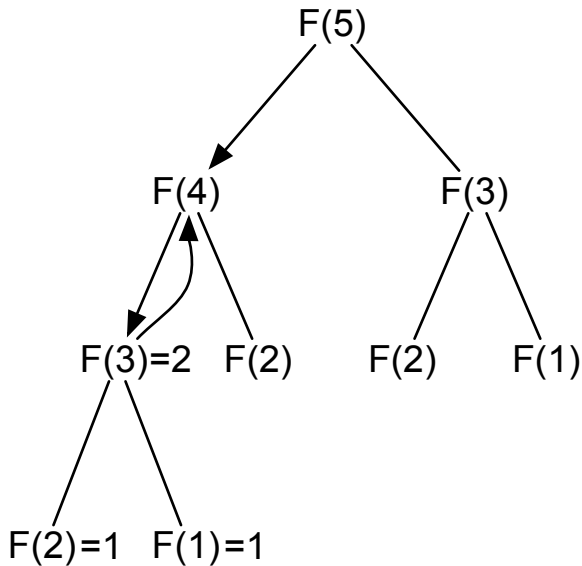
Recursion tree for Fibonacci



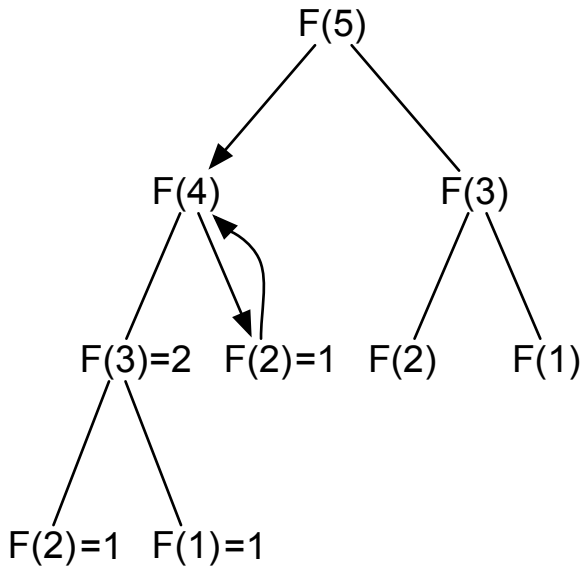
Recursion tree for Fibonacci



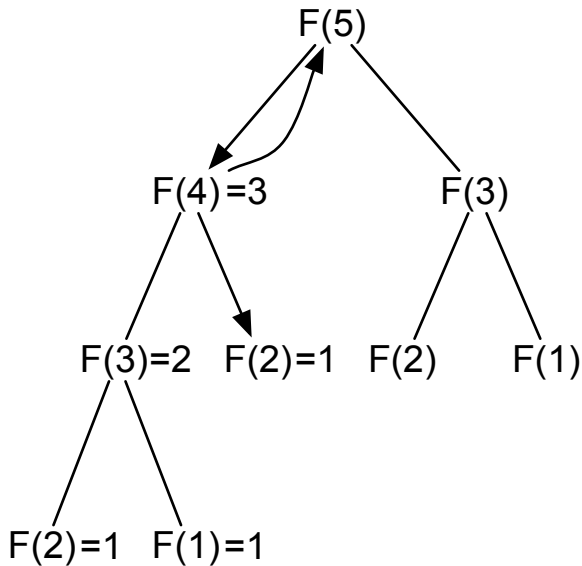
Recursion tree for Fibonacci



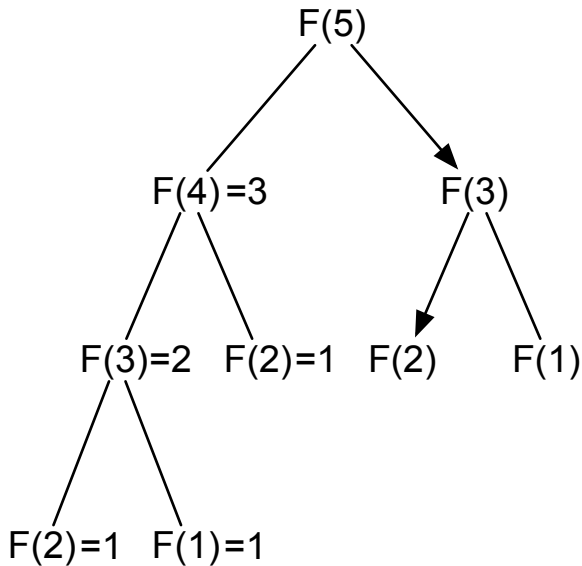
Recursion tree for Fibonacci



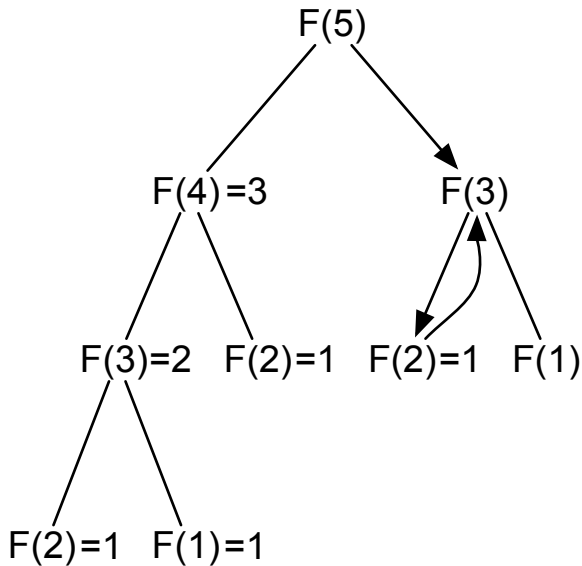
Recursion tree for Fibonacci



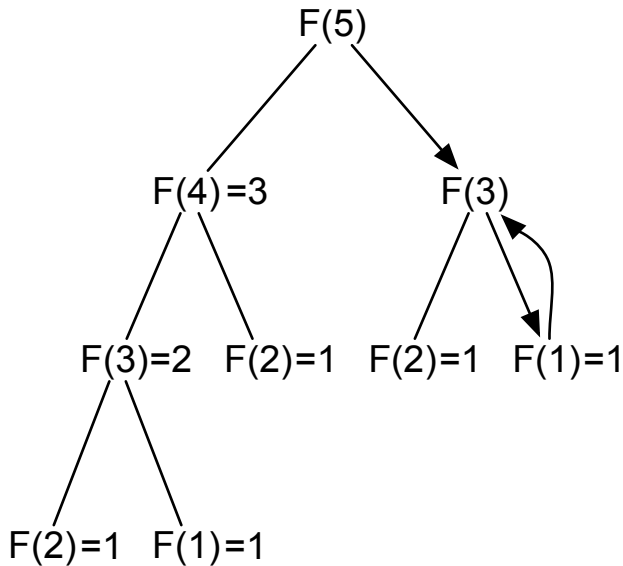
Recursion tree for Fibonacci



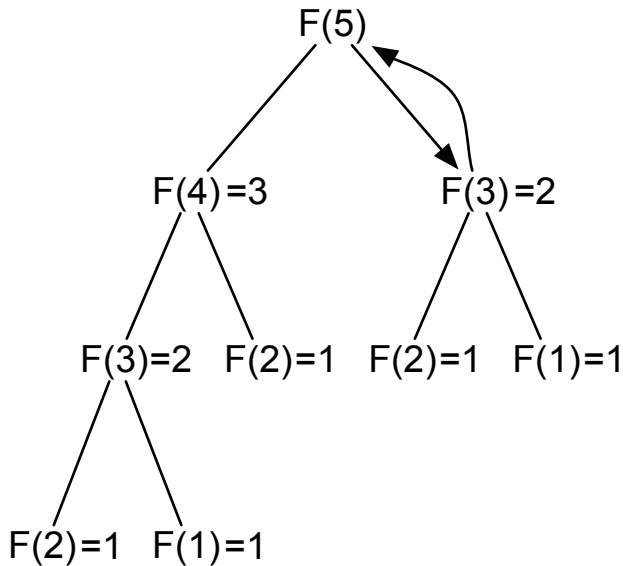
Recursion tree for Fibonacci



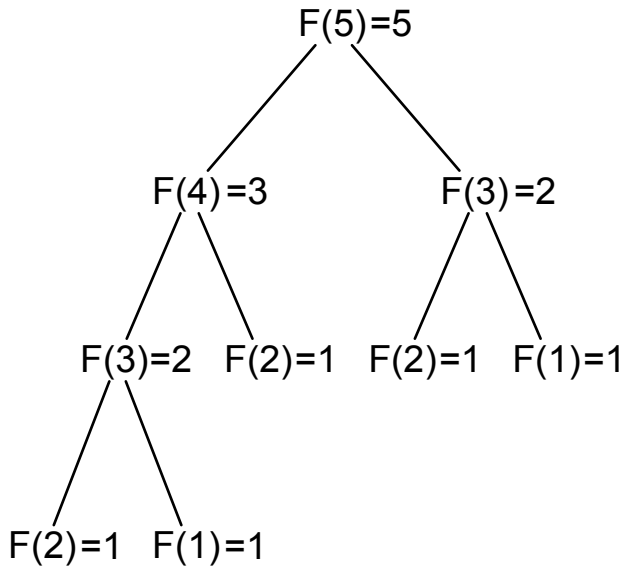
Recursion tree for Fibonacci



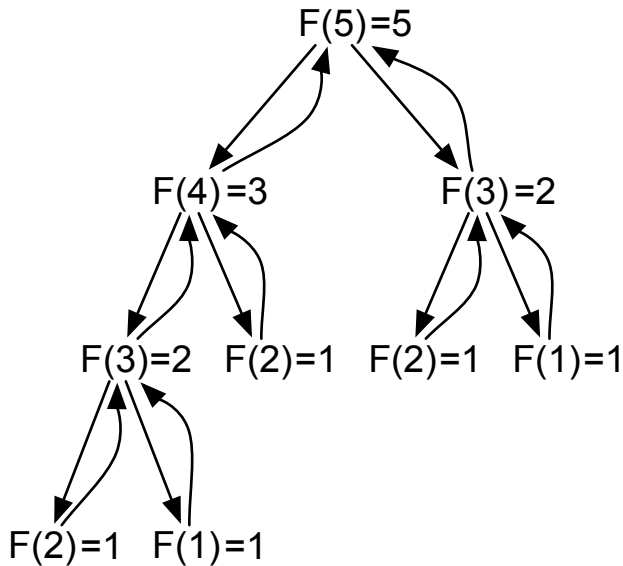
Recursion tree for Fibonacci



Recursion tree for Fibonacci



Recursion tree for Fibonacci



Recursion tree properties

Each node in the tree is a function call.

The number of nodes in the tree is the number of *function 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 function 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*.

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

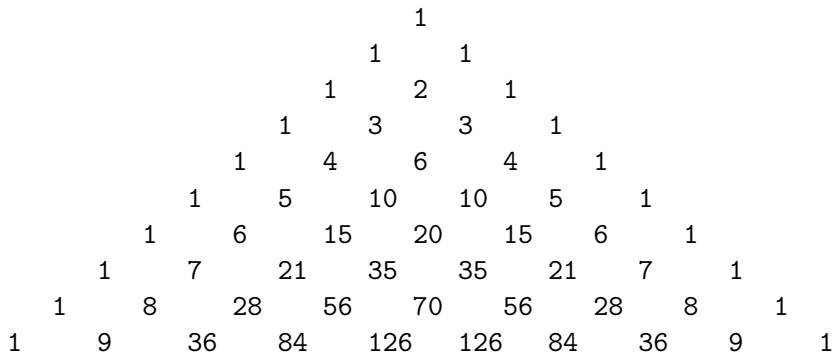
$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

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

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

...and are often seen in Pascal's Triangle.

Pascal's triangle



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 functions

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

Let's look at these cases separately:

Case 1: Factorial:

$$n! = n(n-1)! = n(n-1)(n-2)! = \dots = n(n-1)(n-2)\dots(2)(1)$$

This should show you how to code factorial with an *iterative* function...

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

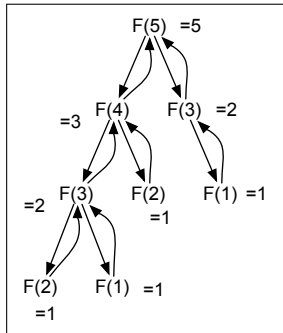
```
1 def fac(n):
2     prod = 1
3     i = n
4     while (i > 1):
5         prod *= i
6         i -= 1
7
8     return prod
9
10 if __name__ == '__main__':
11     import timeit
12     i = 0
13     while i <= 1000:
14         statement = "fac({})".format(i)
15         tt = timeit.Timer(statement, setup="from __main__ import
16             fac")
17         stats = tt.timeit(number=1)
18         print("statement {0} took {1:.4f} time".format(statement,
19             stats))
20         i += 1
```

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 function 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 function when $n = 40$ and you'll see what I mean!

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

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

```
1 def fib(n):  
2     if n <= 2:  
3         return 1  
4  
5     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.

```
7  if __name__ == '__main__':  
8      import timeit  
9      i = 5  
10     while i <= 38:  
11         statement = "fib({})".format(i)  
12         tt = timeit.Timer(statement, setup="from __main__ import  
           fib")  
13         stats = tt.timeit(number=1)  
14         print("statement {0} took {1:.4f} time".format(statement,  
           stats))  
15         i += 1
```

Running Fibonacci.py

```
statement fib(5) took 0.0000 time
statement fib(6) took 0.0000 time
statement fib(7) took 0.0000 time
statement fib(8) took 0.0000 time
statement fib(9) took 0.0000 time
statement fib(10) took 0.0000 time
statement fib(11) took 0.0001 time
statement fib(12) took 0.0001 time
statement fib(13) took 0.0001 time
statement fib(14) took 0.0002 time
statement fib(15) took 0.0003 time
statement fib(16) took 0.0006 time
statement fib(17) took 0.0010 time
statement fib(18) took 0.0013 time
statement fib(19) took 0.0022 time
statement fib(20) took 0.0033 time
statement fib(21) took 0.0055 time
statement fib(22) took 0.0090 time
statement fib(23) took 0.0160 time
```

Running Fibonacci.py (cont.)

```
statement fib(24) took 0.0255 time
statement fib(25) took 0.0429 time
statement fib(26) took 0.0632 time
statement fib(27) took 0.1051 time
statement fib(28) took 0.1664 time
statement fib(29) took 0.2756 time
statement fib(30) took 0.4440 time
statement fib(31) took 0.7118 time
statement fib(32) took 1.1318 time
statement fib(33) took 1.8688 time
statement fib(34) took 3.1068 time
statement fib(35) took 4.9362 time
statement fib(36) took 7.7565 time
statement fib(37) took 12.5528 time
statement fib(38) took 20.3215 time
```

Youch.

```
1 def fib(n):
2     if n < 3:
3         return 1
4
5     F = [0] * (n+1)
6     F[0] = 1
7     F[1] = 1
8     F[2] = 1
9     i = 3
10    while i <= n:
11        F[i] = F[i-1] + F[i-2]
12        i += 1
13
14    return F[n]
```

Running NRFibonacci

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

```
statement fib(5) took 0.0000 time
statement fib(6) took 0.0000 time
statement fib(7) took 0.0000 time
...
statement fib(105) took 0.0001 time
statement fib(106) took 0.0001 time
statement fib(107) took 0.0001 time
...
statement fib(505) took 0.0003 time
statement fib(506) took 0.0005 time
statement fib(507) took 0.0003 time
...
statement fib(997) took 0.0005 time
statement fib(998) took 0.0005 time
statement fib(999) took 0.0005 time
statement fib(1000) took 0.0009 time
```

Recursion: a summary

- recursive functions 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 function