



It's turtles all the way down

RECURSION

Lecture 9

CS2110 – Summer 2019

Since Michael is leaving for a week...

I will take over the next three lectures (Mon, Tue, Fri)

2

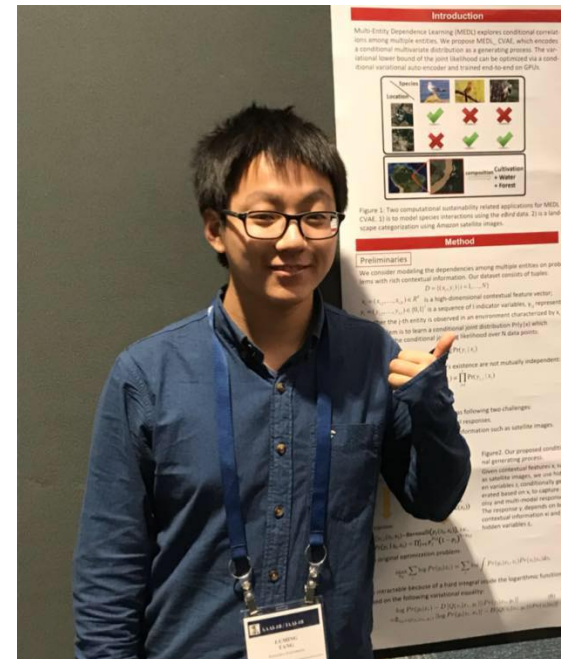
About Me: Luming Tang

Current First-year CS PhD student@Cornell

Former Mathematics and Physics
undergrad@Tsinghua, Beijing, China

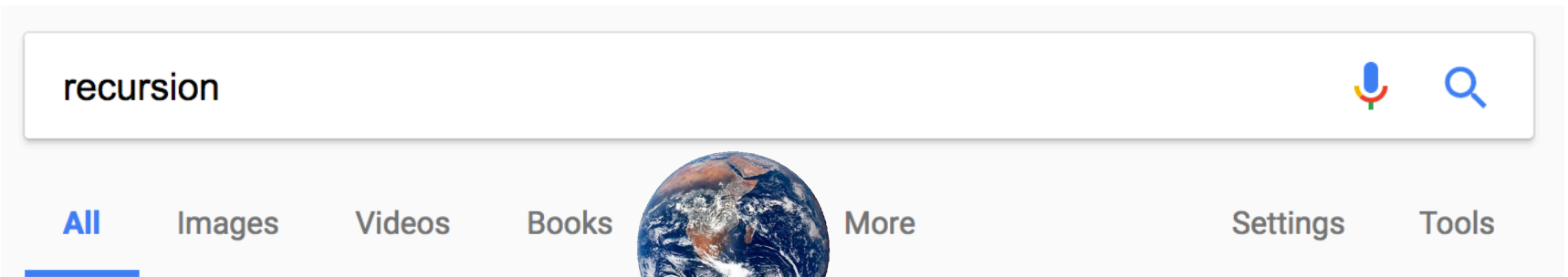
Research interests:

Machine Learning, Computer Vision, trying to figure out the
mathematical foundation giving rise to (Artificial) Intelligence (and
Everything else in the world hopefully...)



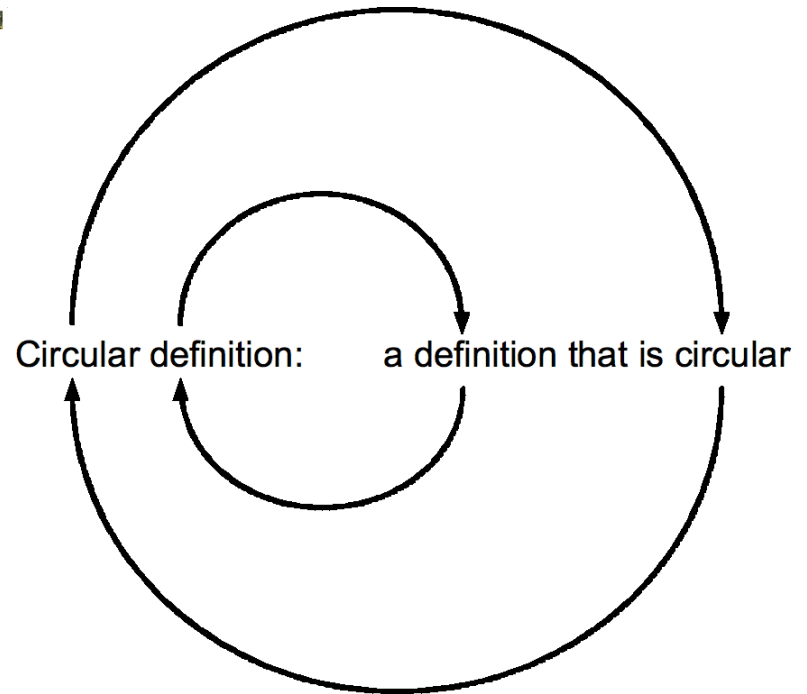
To Understand Recursion...

3



About 10,400,000 results (0.60 seconds)

Did you mean: ***recursion***



Recursion – Real Life Examples

4

<noun phrase> is <noun>, or

<adjective> <noun phrase>, or

<adverb> <noun phrase>

Example:

terrible horrible no-good very bad day

Recursion – Real Life Examples

5

<noun phrase> is <noun>, or

<adjective> <noun phrase>, or

<adverb> <noun phrase>

ancestor(p) is parent(p), or

parent(ancestor(p))

great great great great great great great great great great
great great grandmother.

$0! = 1$

$n! = n * (n-1)!$

1, 1, 2, 6, 24, 120, 720, 5050, 40320, 362880, 3628800, 39916800,
479001600...

Recursion – A Chinese Story...

6

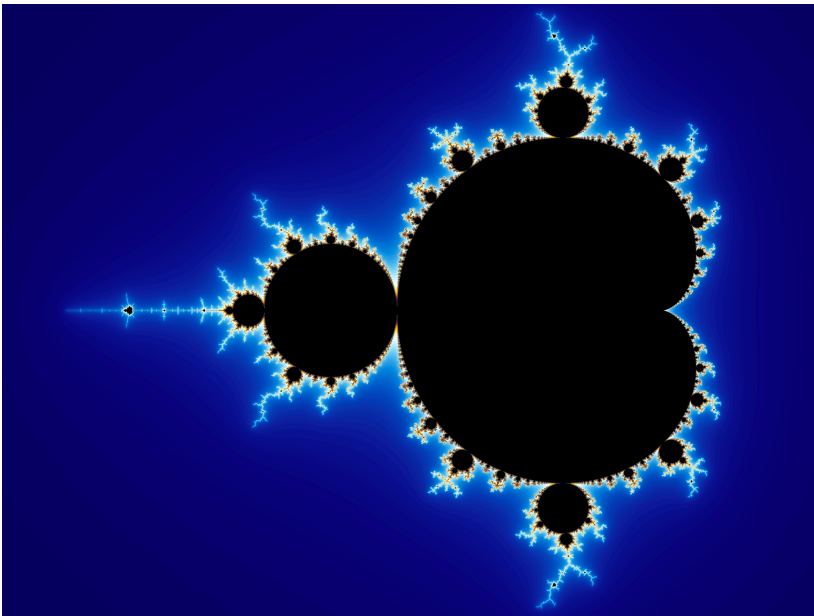
Example:

Once upon a time there was a mountain. There was a temple in the mountain. There was an old monk in the temple who was telling a story to a little monk. What did he say? Once upon a time there was a mountain, there was a temple in the mountain, and an old monk in the temple was telling a story to a little monk...



Recursion – real-world example...

7



Fractal

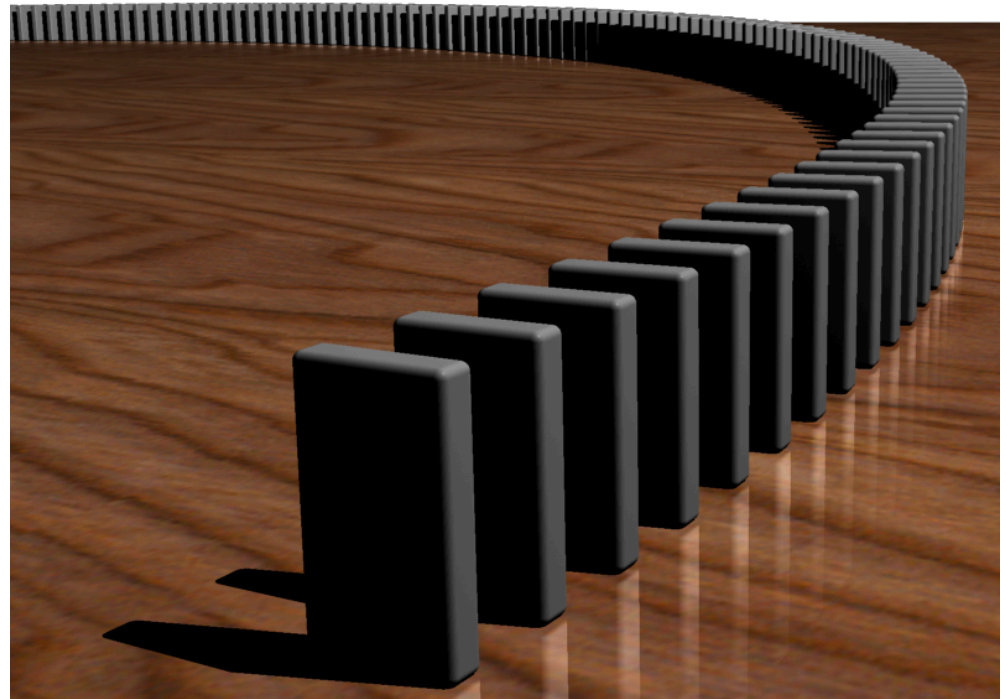


Mirror in mirror

Recursion – Mathematical Induction

8

Mathematical induction proves that we can climb as high as we like on a ladder, by proving that we can climb onto the bottom rung (the **basis**) and that from each rung we can climb up to the next one (the **step**).



Sum the digits in a non-negative integer

9

```
/** = sum of digits in n.  
 * Precondition: n >= 0 */  
public static int sum(int n) {  
    if (n < 10) return n;  
  
    // { n has at least two digits }  
    // return first digit + sum of rest  
    return n%10 + sum(n/10);  
}
```

sum calls itself!

$$\text{sum}(7) = 7$$

$$\begin{aligned}\text{sum}(8703) &= 3 + \text{sum}(870) \\ &= 3 + 8 + \text{sum}(70) \\ &= 3 + 8 + 7 + \text{sum}(0)\end{aligned}$$

Two different questions, two different answers

10

1. How is it **executed**?

(or, why does this even work?)

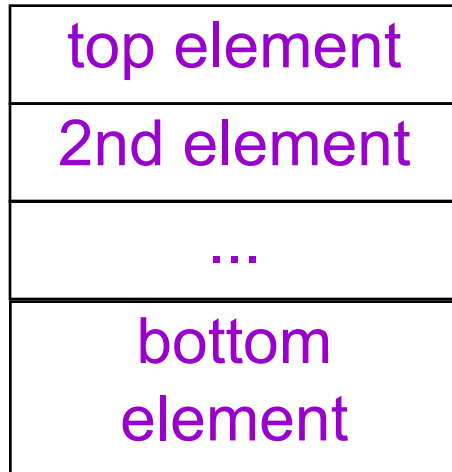
2. How do we **understand** recursive methods?

(or, how do we **write/develop** recursive methods?)

Stacks and Queues

11

↑
stack grows



first	second	...	last
-------	--------	-----	------

Americans wait in a line. The Brits wait in a queue !

Stack: list with (at least) two basic ops:

- * Push an element onto its top
- * Pop (remove) top element

Last-In-First-Out (LIFO)

Like a stack of trays in a cafeteria

Queue: list with (at least) two basic ops:

- * Append an element
- * Remove first element

First-In-First-Out (FIFO)

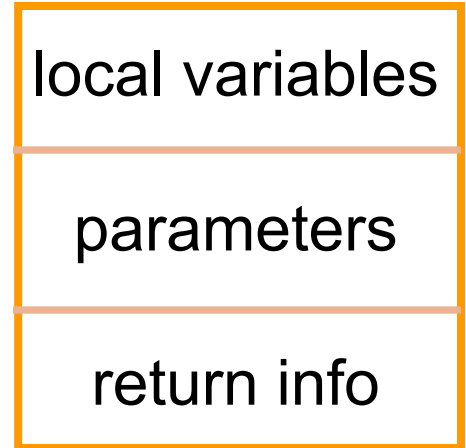
Stack Frame

12

A “frame” contains information about a method call:

At runtime Java maintains a **stack** that contains frames for all method calls that are being executed but have not completed.

a frame



Method call: push a frame for call on **stack**. Assign argument values to parameters. Execute method body. Use the frame for the call to reference local variables and parameters.

End of method call: pop its frame from the **stack**; if it is a function leave the return value on top of **stack**.

Memorize method call execution!

13

A frame for a call contains parameters, local variables, and other information needed to properly execute a method call.

To execute a method call:

1. push a frame for the call on the stack,
2. assign argument values to parameters,
3. execute method body,
4. pop frame for call from stack, and (for a function) push returned value on stack

When executing method body look in frame for call for parameters and local variables.

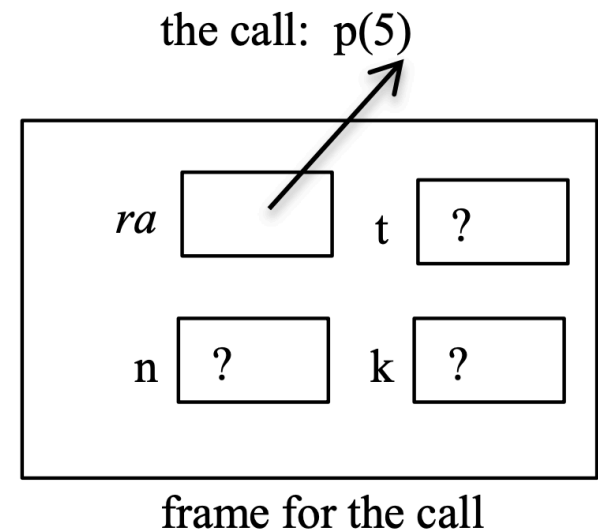
An easy example

14

Consider this procedure `p`, which prints the integers in the range.

Now consider the call `p(5)`. When this call is executed, a box is created, called the frame for the call, to contain all information needed to execute the call. This information includes:

```
/** Print the integers in 1..n. */  
public static void p(int n) {  
    for (int k= 1; k <= n; k= k+1) {  
        int t= k;  
        System.out.println(t);  
    }  
}
```

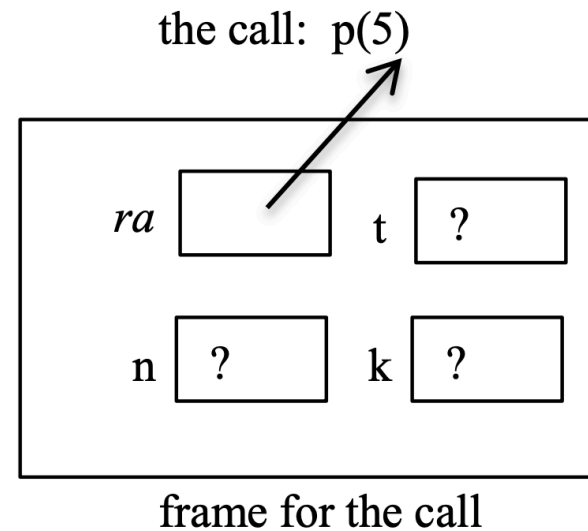


An easy example

This information includes:

1. The parameters of the call (in this case, n),
2. The local variables declared in the method body (k and t),
3. A return address —something to indicate where the call occurred in the program, so it is known where to continue after the call is finished.

```
/** Print the integers in 1..n. */  
public static void p(int n) {  
    for (int k= 1; k <= n; k= k+1) {  
        int t= k;  
        System.out.println(t);  
    }  
}
```



Frames for methods sum main method in the system

16

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return n%10 + sum(n/10);  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Frame for method in the system
that calls method main

frame:

n ____
return info

frame:

r ____ args ____
return info

frame:

?
return info

Example: Sum the digits in a non-negative integer

17

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return n%10 + sum(n/10);  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Frame for method in the system that calls method main: main is then called

main

r ____ args ____
return info

system

?
return info

Memorize method call execution!

18

To execute a method call:

1. push a frame for the call on the stack,
2. assign argument values to parameters,
3. execute method body,
4. pop frame for call from stack, and (for a function) push returned value on stack

The following slides step through execution of a recursive call to demo execution of a method call.

Here, we demo using: www.pythontutor.com/visualize.html

Caution: the frame shows not ALL local variables but only those whose scope has been entered and not left.

Example: Sum the digits in a non-negative integer

19

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return n%10 + sum(n/10);  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Method main calls sum:

main

n 824
return info

r ____ args ____
return info

system

?
return info

Example: Sum the digits in a non-negative integer

20

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return n%10 + sum(n/10);  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

$n \geq 10$ sum calls sum:

main

system

n 82
return info

n 824
return info

r ____ args ____
return info

?
return info

Example: Sum the digits in a non-negative integer

21

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return n%10 + sum(n/10);  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

$n \geq 10$. sum calls sum:

main

system

n 8
return info

n 82
return info

n 824
return info

r _____ args _____
return info

?
return info

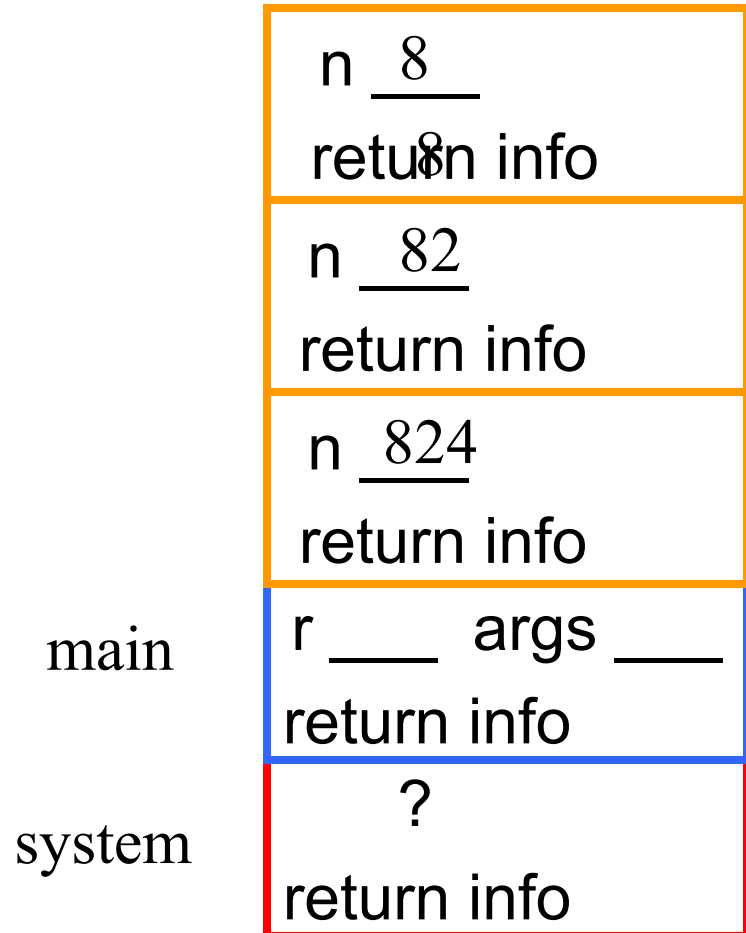
Example: Sum the digits in a non-negative integer

22

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return n%10 + sum(n/10);  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

$n < 10$ sum stops: frame is popped
and n is put on stack:



Example: Sum the digits in a non-negative integer

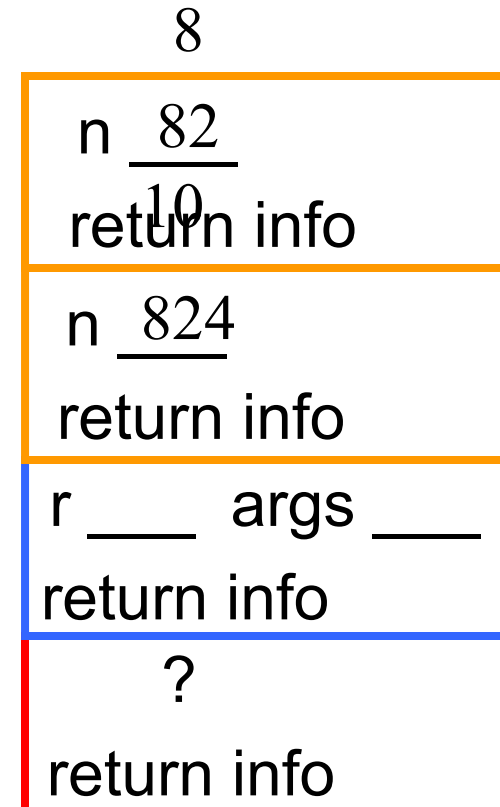
23

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return n%10 + sum(n/10);  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Using return value 8 stack computes
 $2 + 8 = 10$ pops frame from stack puts
return value 10 on stack

main



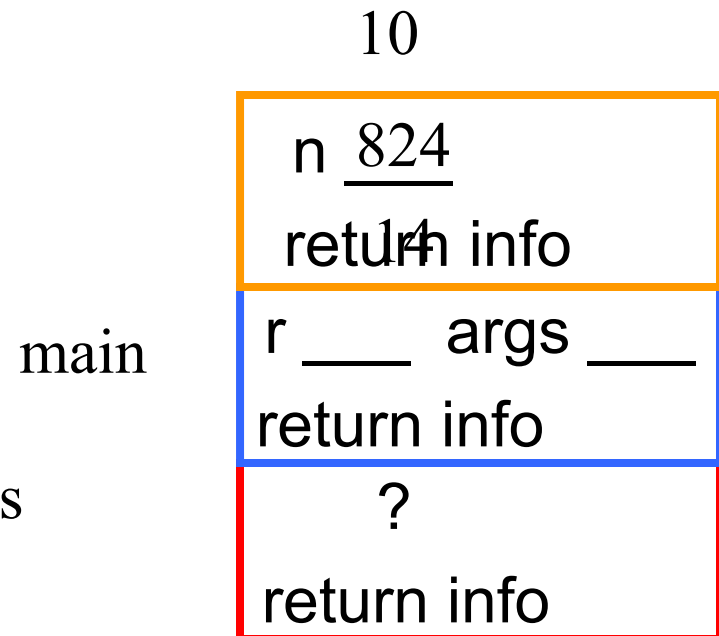
Example: Sum the digits in a non-negative integer

24

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return n%10 + sum(n/10);  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Using return value 10 stack computes
 $4 + 10 = 14$ pops frame from stack
puts return value 14 on stack



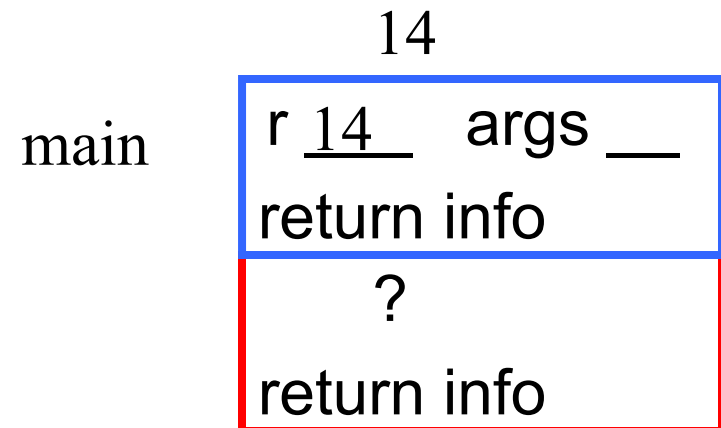
Example: Sum the digits in a non-negative integer

25

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return n%10 + sum(n/10);  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Using return value 14 main stores
14 in r and removes 14 from stack



Questions about local variables

26

```
public static void m(...) {  
    ...  
    while (...) {  
        int d= 5;  
        ...  
    }  
}
```

```
public static void m(...) {  
    int d;  
    ...  
    while (...) {  
        d= 5;  
        ...  
    }  
}
```

In a call `m(...)`
when is local variable `d` created and when is it destroyed?
Which version of procedure `m` do you like better? Why?

Two different questions, two different answers

27

1. How is it **executed**?

(or, why does this even work?)

It's **not** magic! Trace the code's execution using the method call algorithm, drawing the stack frames as you go.

Use only to gain understanding / assurance that recursion works.

2. How do we **understand** recursive methods?

(or, how do we **write/develop** recursive methods?)

This requires a totally different approach.

Back to Real Life Examples

28

Factorial function:

$$0! = 1$$

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

$$\text{(e.g.: } 4! = 4*3*2*1=24\text{)}$$

Exponentiation:

$$b^0 = 1$$

$$b^c = b * b^{c-1} \text{ for } c > 0$$

Easy to make math definition into a Java function!

```
public static int fact(int n) {  
    if (n == 0) return 1;  
  
    return n * fact(n-1);  
}
```

```
public static int exp(int b, int c) {  
    if (c == 0) return 1;  
  
    return b * exp(b, c-1);  
}
```

How to understand what a call does

29

Make a copy of the method spec,
replacing the parameters of the
method by the arguments

spec says that the
value of a call
equals the sum of
the digits of n

sumDigs(654)

sum of digits of **n**

sum of digits of **654**

```
/** = sum of the digits of n.  
 * Precondition: n >= 0 */  
public static int sumDigs(int n) {  
    if (n < 10) return n;  
    // n has at least two digits  
    return n%10 + sumDigs(n/10);  
}
```

Understanding a recursive method

30

Step 1. Have a **precise spec!**

Step 2. Check that the method works in **the base case(s)**: That is, cases where the parameter is **small** enough that the result can be computed simply and without recursive calls.

If $n < 10$ then n consists of a single digit.

Looking at the spec we see that that digit is the required sum.

```
/** = sum of the digits of n.  
 * Precondition: n >= 0 */  
public static int sumDigs(int n) {  
    if (n < 10) return n;  
    // n has at least two digits  
    return n%10 + sumDigs(n/10);  
}
```

Understanding a recursive method

31

Step 1. Have a precise spec!

Step 2. Check that the method works in **the base case(s)**.

Step 3. Look at the **recursive case(s)**. In your mind replace each recursive call by what it

does according to the method spec and verify that the correct result is then obtained.

```
return n%10 + sum(n/10);
```

```
return n%10 + (sum of digits of n/10); // e.g. n = 843
```

```
/** = sum of the digits of n.  
 * Precondition: n >= 0 */  
public static int sumDigs(int n) {  
    if (n < 10) return n;  
    // n has at least two digits  
    return n%10 + sumDigs(n/10);  
}
```

Understanding a recursive method

32

Step 1. Have a precise spec!

Step 2. Check that the method works in **the base case(s)**.

Step 3. Look at the **recursive case(s)**. In your mind replace each recursive call by what it does acc. to the spec and verify correctness.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the pars of the method.

$n/10 < n$, so it will get smaller until it has one digit

```
/** = sum of the digits of n.  
 * Precondition: n >= 0 */  
public static int sumDigs(int n) {  
    if (n < 10) return n;  
    // n has at least two digits  
    return n%10 + sumDigs(n/10);  
}
```


Understanding a recursive method

33

Step 1. Have a precise spec!

Important! Can't do step 3 without precise spec.

Step 2. Check that the method works in **the base case(s)**.

Step 3. Look at the **recursive case(s)**. In your mind replace each recursive call by what it does according to the spec and verify correctness.

Once you get the hang of it this is what makes recursion easy! This way of thinking is based on math induction which we don't cover in this course.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method

Writing a recursive method

34

Step 1. Have a precise spec!

Step 2. Write the **base case(s)**: Cases in which no recursive calls are needed. Generally for “small” values of the parameters.

Step 3. Look at all other cases. See how to define these cases in terms **of smaller problems of the same kind**. Then implement those definitions using recursive calls for those **smaller problems of the same kind**. Done suitably, point 4 (about termination) is automatically satisfied.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method

Two different questions, two different answers

35

2. How do we **understand** recursive methods?
(or, how do we **write/develop** recursive methods?)

Step 1. Have a precise **spec**!

Step 2. Check that the method works in **the base case(s)**.

Step 3. Look at the **recursive case(s)**. In your mind replace each recursive call by what it does according to the spec and verify correctness.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the parameters of the method

Examples of writing recursive functions

36

For the rest of the class we demo writing recursive functions using the approach outlined below. The java file we develop will be placed on the course webpage some time after the lecture.

Step 1. Have a precise **spec!**

Step 2. Write the **base case(s)**.

Step 3. Look at all other cases. See how to define these cases in terms **of smaller problems of the same kind**. Then implement those definitions using recursive calls for those **smaller problems of the same kind**.

Step 4. Make sure recursive calls are “smaller” (no infinite recursion).

Check palindrome-hood

37

A String palindrome is a String that reads the same backward and forward:

`isPal("racecar")` → true `isPal("pumpkin")` → false

A String with at least two characters is a palindrome if

- (0) its first and last characters are equal and
- (1) chars between first & last form a palindrome:

e.g. AMANAPLANACANALPANAMA

have to be the same

have to be a palindrome

A recursive definition!

Check palindrome-hood

38

A String palindrome is a String that reads the same backward and forward:

`isPal("racecar")` → true `isPal("pumpkin")` → false

A String with at least two characters is a palindrome if

- (0) its first and last characters are equal and
- (1) chars between first & last form a palindrome:
- Also, a Chinese version lol:

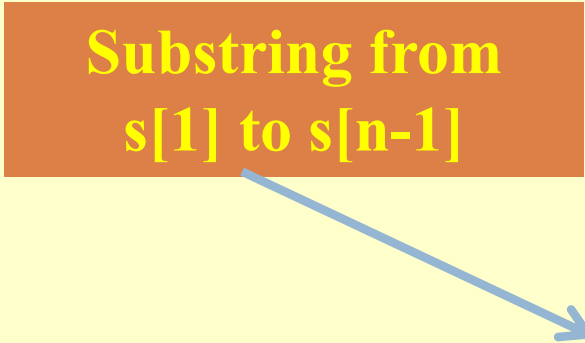
上海自来水来自海上...

- A man a plan a caret a ban a myriad a sum a lac a liar a hoop a pint a catalpa a gas an oil a bird a yell a vat a caw a pax a wag a tax a nay a ram a cap a yam a gay a tsar a wall a car a luger a ward a bin a woman a vassal a wolf a tuna a nit a pall a fret a watt a bay a daub a tan a cab a datum a gall a hat a fag a zap a say a jaw a lay a wet a gallop a tug a trot a trap a tram a torr a caper a top a tonk a toll a ball a fair a sax a minim a tenor a bass a passer a capital a rut an amen a ted a cabal a tang a sun an ass a maw a sag a jam a dam a sub a salt an axon a sail an ad a wadi a radian a room a rood a rip a tad a pariah a revel a reel a reed a pool a plug a pin a peek a parabola a dog a pat a cud a nu a fan a pal a rum a nod an eta a lag an eel a batik a mug a mot a nap a maxim a mood a leek a grub a gob a gel a drab a citadel a total a cedar a tap a gag a rat a manor a bar a gal a cola a pap a yaw a tab a raj a gab a nag a pagan a bag a jar a bat a way a papa a local a gar a baron a mat a rag a gap a tar a decal a tot a led a tic a bard a leg a bog a burg a keel a doom a mix a map an atom a gum a kit a baleen a gala a ten a don a mural a pan a faun a ducat a pagoda a lob a rap a keep a nip a gulp a loop a deer a leer a lever a hair a pad a tapir a door a moor an aid a raid a wad an alias an ox an atlas a bus a madam a jag a saw a mass an anus a gnat a lab a cadet an em a natural a tip a caress a pass a baronet a minimax a sari a fall a ballot a knot a pot a rep a carrot a mart a part a tort a gut a poll a gateway a law a jay a sap a zag a fat a hall a gamut a dab a can a tabu a day a batt a waterfall a patina a nut a flow a lass a van a mow a nib a draw a regular a call a war a stay a gam a yap a cam a ray an ax a tag a wax a paw a cat a valley a drib a lion a saga a plat a catnip a pooh a rail a calamus a dairyman a bater a canal Panama

Example: Is a string a palindrome?

40

```
/** = "s is a palindrome" */  
public static boolean isPal(String s) {  
    if (s.length() <= 1)  
        return true;  
  
    // { s has at least 2 chars }  
    int n= s.length()-1;  
    return s.charAt(0) == s.charAt(n) && isPal(s.substring(1,n));  
}
```



The Fibonacci Function

41

Mathematical definition:

$$\begin{aligned} \text{fib}(0) &= 0 \\ \text{fib}(1) &= 1 \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \quad n \geq 2 \end{aligned}$$

two base cases!

Fibonacci sequence: 0 1 1 2 3 5 8 13 ...

```
/** = fibonacci(n). Pre: n >= 0 */  
static int fib(int n) {  
    if (n <= 1) return n;  
    // { 1 < n }  
    return fib(n-1) + fib(n-2);  
}
```



Fibonacci (Leonardo
Pisano) 1170-1240?

Statue in Pisa Italy
Giovanni Paganucci
1863

Example: Count the e's in a string

42

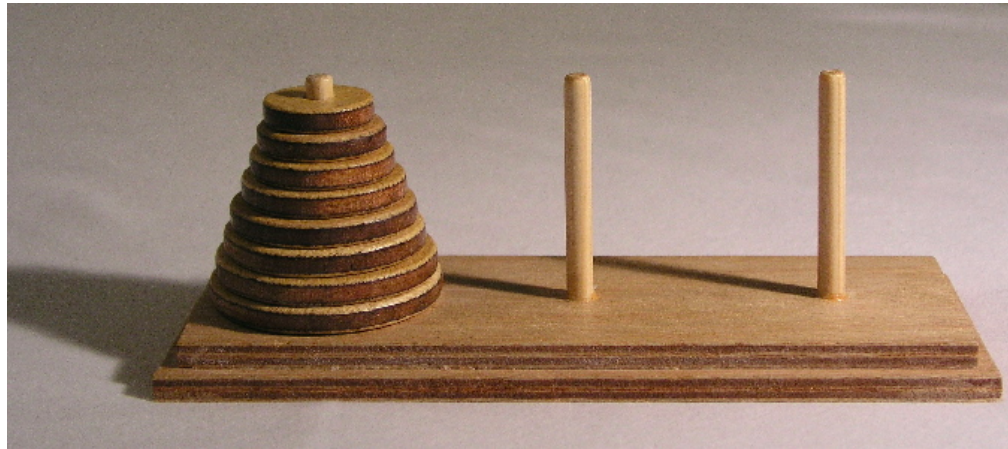
```
/** = number of times c occurs in s */  
public static int countEm(char c, String s) {  
    if (s.length() == 0) return 0;  
  
    // { s has at least 1 character }  
    if (s.charAt(0) != c)  
        return countEm(c, s.substring(1));  
  
    // { first character of s is c }  
    return 1 + countEm(c, s.substring(1));  
}
```

substring s[1..]
i.e. s[1] ...
s(s.length()-1)

- `countEm('e', "it is easy to see that this has many e's") = 4`
- `countEm('e', "Mississippi") = 0`

Plus...Tower of Hanoi

43



The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No larger disk may be placed on top of a smaller disk.