Recursion

We teach recursion as the first topic, instead of new object-oriented ideas, so that those who are new to Java can have a chance to catch up on the object-oriented ideas from CS100.

Recursive definition: A definition that is defined in terms of itself.

Recursive method: a method that calls itself (directly or indirectly).

Recursion is often a good alternative to iteration (loops). Its an important programming tool. Functional languages have no loops -- only recursion.

Readings:
Weiss, Chapter 7, page 231-249.
CS211 power point slides for recursion

Homework: See handout.

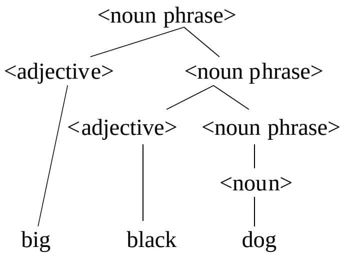
Recursion

Recursive definition: A definition that is defined in terms of itself.

A noun phrase is either

- a noun, or
- an adjective followed by a noun phrase

<noun phrase> ::= <noun>
| <adjective> <noun phrase>



<noun phrase>

<adjective> <noun phrase>

<adjective> <noun phrase>

<noun>

big black dog

Recursive definitions in mathematics

Factorial:

!0 = 1 **base case**
!n = n * !(n-1) for n > 0 **recursive case**

Thus, !3 = 3 * !2
= 3 * 2 * !1
= 3 * 2 * 1 * !0
= 3 * 2 * 1 * 1 (= 6)

Fibonacci sequence:

$Fib_0 = 0$ **base case**
$Fib_1 = 1$ **base case**
$Fib_n = Fib_{n-1} + Fib_{n-2}$ for n > 1 **recursive case**

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Turn recursive definition into recursive function

Factorial:

!0 = 1 **base case**
!n = n * !(n-1) for n > 0 **recursive case**

Thus, !3 = 3 * !2
= 3 * 2 * !1
= 3 * 2 * 1 * !0
= 3 * 2 * 1 * 1 (= 6)

note the precise specification

```
// = !n  (for n>=0)
public static int fact(int n) {
   if (n == 0) {
         return 1;                  base case
   }
   // {n > 0}                       an assertion
   return n * fact(n-1);            recursive case
}                                   (a recursive call)
```

Later, we explain why this works.

Turn recursive definition into recursive function

Fibonacci sequence:

$Fib_0 = 0$ **base case**
$Fib_1 = 1$ **base case**
$Fib_n = Fib_{n-1} + Fib_{n-2}$ for $n > 1$ **recursive case**

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

note the precise specification

```
// = Fibonacci number n (for n >= 0)
public static int Fib(int n) {
   if (n <= 1) {                can handle both
         return n;              base cases together
   }
   // {n > 0}                   an assertion
   return Fib(n-1) + Fib(n-2); recursive case
}                               (two recursive calls)
```

Later, we explain why this works.

Two issues in coming to grips with recursion

1. How are recursive calls executed?

2. How do we understand a recursive method and how do we write-create a recursive method?

We will handle both issues carefully. But for proper use of recursion they must be kept separate.

We DON’T try to understand a recursive method by executing its recursive calls!

Understanding a recursive method

MEMORIZE THE FOLLOWING

Step 0: HAVE A PRECISE SPECIFICATION.

Step 1: Check correctness of the base case.

Step 2: Check that recursive-call arguments are in some way smaller than the parameters, so that recursive calls make progress toward termination (the base case).

Step 3: Check correctness of the recursive case. When analyzing recursive calls, use the specification of the method to understand them.

Weiss doesn't have step 0 and adds point 4, which has nothing to do with "understanding"

4: Don't duplicate work by solving some instance in two places.

Understanding a recursive method

Factorial:

!0 = 1		**base case**
!n = n * !(n-1)	for n > 0	**recursive case**

Step 1: HAVE A PRECISE SPECIFICATION

```
// = !n  (for n>=0)
public static int fact(int n) {
   if (n == 0) {
         return 1;                  base case
   }
   // {n > 0}
   return n * fact(n-1);           recursive case
}                                 (a recursive call)
```

Step 2: Check the base case.

Here's when n = 0, 1 is returned, which is 0!. So the base case is handled correctly.

Understanding a recursive method

Factorial:

!0 = 1	**base case**
!n = n * !(n-1) for n > 0	**recursive case**

Step 3: Recursive calls make progress toward termination.

argument n-1 is smaller than parameter n, so there is progress toward reaching base case 0

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

recursive case

parameter n

argument n-1

Understanding a recursive method

Factorial:

!0 = 1	**base case**
!n = n * !(n-1) for n > 0	**recursive case**

Step 4: Check correctness of recursive case; use the method specification to understand recursive calls.

> **In the recursive case, the value returned is**
>
> **n * fact(n -1).**
>
> **Using the specification for method fact, we see this is equivalent to**
>
> **n * !(n -1).**
>
> **That's the definition of !n, so the recursive case is correct.**

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

recursive case

Creating recursive methods

Use the same steps that were involved in understanding a recursive method.

- **Be sure you SPECIFY THE METHOD PRECISELY.**

- **Handle the base case first**

- **In dealing with the non-base cases, think about how you can express the task in terms of a similar but smaller task.**

Creating a recursive method

Task: Write a method that removes blanks from a String.

0. Specification: **precise specification!**

```
// = s but with its blanks removed
public static String deblank(String s)
```

1. Base case: the smallest String is “”.

```
    if (s.length == 0)
        return s;
```

2. Other cases: String s has at least 1 character. If it’s blank, return s[1..] but with its blanks removed. If it’s not blank, return

s[0] + (s[1..] but with its blanks removed)

Notation: s[i] is shorthand for s.charAt[i]. s[i..] is shorthand for s.substring(i).

Creating a recursive method

```
// = s but with its blanks removed
public static String deblank(String s) {
   if (s.length == 0)
         return s;
   // {s is not empty}
   if  (s[0] is a blank)
         return s[1..] with its blanks removed
   // {s is not empty and s[0] is not a blank}
   return s[0] + (s[1..] with its blanks removed);
}
```

The tasks given by the two English, blue expressions are similar to the task fulfilled by this function, but on a smaller String! !!!Rewrite each as

```
   deblank(s[1..]) .
```

Notation: s[i] is shorthand for s.charAt[i].
s[i..] is shorthand for s.substring(i).

Creating a recursive method

```
// = s but with its blanks removed
public static String deblank(String s) {
   if (s.length == 0)
         return s;
   // {s is not empty}
   if  (s.charAt(0) is a blank)
         return deblank(s.substring(1));
   // {s is not empty and s[0] is not a blank}
   return s.charAt(0) +
            deblank(s.substring(1));
}
```

Check the four points:
0. Precise specification?
1. Base case: correct?
2. Recursive case: progress toward termination?
3. Recursive case: correct?

Creating a recursive method

Task: Write a method that tests whether a String is a palindrome (reads the same backwards and forward).

E.g. palindromes: noon, eve, ee, o, “”
nonpalindromes: adam, no

0. Specification:

precise specification!

```
// = “s is a palindrome”
public static boolean isPal(String s)
```

1. Base case: the smallest String is “”. A string consisting of 0 or 1 letters is a palindrome.

```
   if (s.length() <= 1)
         return true;
   // { s has at least two characters }
```

Creating a recursive method

```
// = “s is a palindrome”
public static boolean isPal(String s) {
   if (s.length() <= 1)
         return true;
   // { s has at least two characters }
```

We treat the case that s has at least two letters. How can we find a smaller but similar problem (within s)?

s is a palindrome if

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

have to be the same

e.g. **AMANAPLANACANALPANAMA**

has to be a palindrome

the task to decide whether the characters between the last and first form a palindrome is a smaller, similar problem!!

Creating a recursive method

```
// = “s is a palindrome”
public static boolean isPal(String s) {
   if (s.length() <= 1)
         return true;
   // { s has at least two characters }
```

We treat the case that s has at least two letters. How can we find a smaller but similar problem (within s)?

s is a palindrome if

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

have to be the same

e.g. **AMANAPLANACANALPANAMA**

has to be a palindrome

the task to decide whether the characters between the last and first form a palindrome is a smaller, similar problem!!

Binary search

Consider **int** array b[0..n-1] and integer x. Assume that

virtual element b[-1] contains -
virtual element b[n] contains

```
       -1  0  1   2  3  4  5  6  7
   b = -  3    5   7  7  7  9  9        n = 7
```

Find an index i such that

b[i] <= x <= b[i+1]

If x = 7, finds position of rightmost 7.
If x = 2, return 0.
If x = -5, return 0
If x = 15, return 9

```
// = index i such b[i] <= x <= b[i+1]
//    precondition b[h] <= x <= b[k]  and
//                      -1 <= h < k <= b.length
public static int bsearch(int[] b, int h, int k)
```

Search whole array using:

```
bsearch(b, 0, b.length)
```

Binary search

Consider **int** array b[0..n-1] and integer x. Assume that

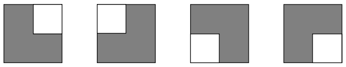
virtual element b[-1] contains -
virtual element b[n] contains

```
        -1  0  1   2  3  4  5  6  7
   b = -  3     5   7  7  7  9  9        n = 7
```

```
// = index i such b[i] <= x <= b[i+1]
//    precondition b[h] <= x <= b[k]  and
//                          -1 <= h < k <= b.length
public static int bsearch(int[] b, int h, int k) {
     int e= (h+k) % 2;
     // {-1 <= h < e < k <= b.length}
     if (b[e] <= x)
          {  i= e;  }
     else  {j= e;}
}
```

Tiling Elaine’s Kitchen

2**n by 2**n kitchen, for some n>= 0.
A 1 by 1 refrigerator sits on one of the squares of the kitchen. Tile the kitchen with L-shaped tiles:, each a 2 by 2 tile with one corner removed:



Base case: n=0, so it’s a 2**0 by 2**0 kitchen.
Nothing to do!

Recursive case: n>0. How can you find the same kind of problem, but smaller, in the big one?