CSE 8B Today

Recursion or magic— you decide!

Beginner’s Programming Competition

When: Saturday, June 1st 3 P.M to 7:30 P.M.
Where: CSE Building, Room B230

wic.ucsd.edu/competition.html

Register soon—event will fill!
All beginners welcome (no upper div CS)

Exam 3 returned today
1. What is a recursive method?

A. A method that contains no implementation.

B. A method that calls other methods, but never itself.

C. A method that directly or indirectly calls itself.

D. A method that doesn’t call any other methods.
2. What do you call the simplest case in a recursive method?

A. The base case.

B. The halting problem.

C. The break statement.

D. The breakpoint.

A. The base case.
3. Which two numbers does the Fibonacci sequence start with? In other words, what are the two base cases of the Fibonacci sequence?

A. -1, 0
B. 0, 1
C. 1, 2
D. 2, 3
public class RecursionDemo {
    public int g(int x) {
        return -1 * x;
    }
}

// In main:
RecursionDemo r = new RecursionDemo();
r.g(-4);

Fill in the memory model for the code at left
while g is executing
public class RecursionDemo {
    public int g( int x ) {
        return -1 * x;
    }
    public int f( int x ) {
        return g(x/2);
    }
}

//.. In main:
RecursionDemo r = new RecursionDemo();
r.f(-4);

What does the memory model for this code look like?
Draw the model at the point when g is executing its body.
(draw now, then match next slide)
A

B

C

D
public class RecursionDemo
{
    public int g( int x ) {
        return -1 * x;
    }
    public int f( int x ) {
        return g(x/2);
    }
}
//.. In main:
RecursionDemo r = new RecursionDemo();
r.f(-4);
Function data is stored in memory on “the stack” (objects are stored in a different part of memory called “the heap”)

Every time a function is called, Java makes a new “stack frame” which stores all of the data for that function (local variables)

(Disclaimer: this isn’t quite true, but close enough for us)

For the rest of this lecture (and the whole section on recursion) we will ignore the RecursionDemo object and main, and focus only on the methods the user called explicitly.
public int demo(x) {
    return x + f(x);
}

public int f(x) {
    return 11*g(x) + g(x/2);
}

public int g(x) {
    return -1 * x;
}

How many stack frames (total) are created when we call \texttt{r.demo(-4)}?

A. 0
B. 1
C. 2
D. 3
E. 4

\( \text{ever created} \)
public int demo(x) {
    return x + f(x);
}

public int f(x) {
    return 11*g(x) + g(x/2);
}

public int g(x) {
    return -1 * x;
}

What is \texttt{demo(-4)}?
How functions work…

public int demo(x) {
    return x + f(x);
}
public int f(x) {
    return 11*g(x) + g(x/2);
}
public int g(x) {
    return -1 * x;
}

What is $\text{demo}(-4)$?
How functions work...

```
public int demo(x) {
    return x + f(x);
}
public int f(x) {
    return 11*g(x) + g(x/2);
}
public int g(x) {
    return -1 * x;
}
```

What is `demo(-4)`?

These are different x's!
How functions work...

public int demo(x) {
    return x + f(x);
}

public int f(x) {
    return 11* g(x) + g(x/2);
}

public int g(x) {
    return -1 * x;
}

What is \texttt{demo(-4)} ?
public int demo(x) {
    return x + f(x);
}

public int f(x) {
    return 11*g(x) + g(x/2);
}

public int g(x) {
    return -1 * x;
}

What is \texttt{demo(-4)} ?
How functions work...

```java
public int demo(x) {
    return x + f(x);
}
public int f(x) {
    return 11*g(x) + g(x/2);
}
public int g(x) {
    return -1 * x;
}
```

What is \( demo(-4) \) ?
public int demo(x) {
    return x + f(x);
}

public int f(x) {
    return 11*g(x) + g(x/2);
}

public int g(x) {
    return -1 * x;
}

What is `demo(-4)`?
How functions work…

public int demo(x) {
    return x + f(x);
}

public int f(x) {
    return 11*g(x) + g(x/2);
}

public int g(x) {
    return -1 * x;
}

What is \texttt{demo(-4)} ?
How functions work...

```java
public int demo(x) {
    return x + f(x);
}
public int f(x) {
    return 11*g(x) + g(x/2);
}
public int g(x) {
    return -1 * x;
}
```

What is `demo(-4)`?
How functions work…

public int demo(x) {
    return x + f(x);
}

public int f(x) {
    return 11*g(x) + g(x/2);
}

public int g(x) {
    return -1 * x;
}

What is \texttt{demo(-4)} ?

demo
\[ x = -4 \]
\[ \text{return } -4 + 46 \]

\[ 42 \]
What does the above code do when we call foo(5)
A) Prints out 5
B) Prints out the numbers from 5 down to 1
C) Prints out the numbers from 1 up to 5
D) I have NO IDEA! How can you call foo from inside foo!?!?!
Recursion and Stack frames

```java
public void foo(int x) {
    if (x > 1)
        foo(x - 1);
    System.out.println(x);
}
// call foo(5)
```

1
Recursion and Stack frames

```java
class Example {
    public static void main(String[] args) {
        // call foo(5)
        foo(5);
    }

    public static void foo(int x) {
        if (x > 1) {
            foo(x - 1);
            System.out.println(x);
        }
    }
}
```

The output is:
```
1
2
3
4
5
```
public void foo(int x) {
    if (x > 1) {
        foo(x - 1);
        System.out.println(x);
    }
} // call foo(5)

// call foo(5)

public void foo(int x) {
    if (x > 1) {
        foo(x - 1);
        System.out.println(x);
    }
}

foo
  x = 5
  foo(4)

foo
  x = 4
  foo(3)
  System.out.println(3)

foo
  x = 3
  foo(2)

foo
  x = 2
  foo(1)

foo
  x = 1

System.out.println(1)

foo
  x = 0

// call foo(5)

1

2

3
Recursion and Stack frames

public void foo(int x) {
    if (x>1) {
        foo(x-1);
        System.out.println(x);
    }
} // call foo(5)

foo
x = 5
foo(4)

foo
x = 4
foo(3)
System.out.println(4)
Recursion and Stack frames

```java
public void foo(int x) {
    if (x>1) {
        foo(x-1);
        System.out.println(x);
    }
} // call foo(5)
```

```
foo
    x = 5
    foo(4)
    System.out.println(5)
```
Thinking *sequentially* 

**factorial**

\[
5! = 120 \\
5! = 5 \times 4 \times 3 \times 2 \times 1
\]

\[
N! = N \times (N-1) \times (N-2) \times \ldots \times 3 \times 2 \times 1
\]
Thinking *recursively*

**factorial**

\[ 5! = 120 \]

\[ 5! = 5 \times 4 \times 3 \times 2 \times 1 \]

\[ N! = N \times (N-1) \times (N-2) \times \ldots \times 3 \times 2 \times 1 \]

\[ N! = N \times (N-1)! \]

Recursive step

\[ N! = N! \]
public int fac(int n) {
    return n * fac(n-1);
}

A. A function cannot call itself in Java
B. It will run forever because it is missing a base case
C. It will not compile
D. It will not compute the correct value of factorial because it is missing the recursive step
Thinking *recursively*

**factorial**

\[ N! = N \times (N-1) \times (N-2) \times \ldots \times 3 \times 2 \times 1 \]

\[ N! = N \times (N-1)! \quad \text{(and 1! = 1)} \]

Recursive step, do a tiny bit of work, and then assume you have code that solves a smaller version of the same problem.

Base case (what to do when the problem can’t be simplified any further)
What should go in blank (1)?
A. \( n > 1 \)
B. \( n \leq 1 \)
C. \( \text{fac}(n) == 1 \)
D. \( \text{fac}(n) > 1 \)
public int fac( int n ) {
    if ( n <= 1 )
        return 1;  \[\text{B. return 0}\]
    else
        return fac( n-1 );
}

What should go in blank (2)?
A. return 1  \[\text{A. return 1}\]
B. return 0
C. return fac(1)
D. return fac(n-1)
E. return n-1
Recursion practice

Write a method to change all the Strings in a String array to lower case, recursively

[“Hello”, “CSE8B”] → [“hello”, “cse8b”]

```java
public void allToLower( String[] myList, int index )
{
  ???
}
```

What is the base case? (i.e., the simplest version of the problem)
A. When the myList already contains only lower case strings
B. When index is (greater than or) equal to the myList.length
C. When index is (greater than or) equal to the myList.length-1
D. When myList is empty
Recursion practice

Write a method to change all the strings in a String array to lower case, recursively
[“Hello”, “CSE8B”] → [“hello”, “cse8b”]

```java
public void allToLower( String[] myList, int index )
{
    if ( index >= myList.length )
        return
    else
        ???
}
```

What line or lines complete the method correctly?

A. `myList[index] = myList[index].toLowerCase();
   allToLower( myList, index );`

B. `allToLower( myList, index-1 );`

C. `myList[index] = myList[index].toLowerCase();
   allToLower( myList, index+1 );`

D. `myList[index] = myList[index].toLowerCase();
   allToLower( myList, index-1 );`
NOTE: All recursive code can be implemented with loops

• Simple: if, else recursion is usually “easy” to turn into a loop

• Next we’ll look at a slightly more complex case (which is still pretty easy to do with a loop)

• Monday: we’ll look at some more complex recursion techniques that are harder to write with loops
  – It’s “natural” to do them recursively
Write a method to find an element in an array and return its index

```java
public int find( String[] myList, String toFind, int currIndex )
{
    ???
}
```

What is the base case?
A. The element at currIndex is equal to toFind
B. The array is empty
C. currIndex is (greater than or) equal to myList.length
D. toFind is not in myList
E. Something else
Recursion: slightly harder practice

Write a method to find an element in an array and return its index

```java
public int find( String[] myList, String toFind, int currIndex )
{
    if ( currIndex >= myList.length )
        return ???
    if ( myList[currIndex].equals( toFind ) )
        return currIndex;

    // recursive step here
}
```
Worksheet: Complete the recursive step

Write a method to find an element in an array and return its index

```java
public int find( String[] myList, String toFind, int currIndex )
{
    if ( currIndex >= myList.length )
        return -1 // It wasn’t there!
    if ( myList[currIndex].equals( toFind ) )
        return currIndex; // That was easy!
}
```