CSE1030 – Introduction to Computer Science II

Lecture #19

Recursion I

CSE1030 – Lecture #19

- Introduction to Recursion
- Execution Stack
- Example: Reversing a String
- Example: Mathematical Bisection
- We're Done!

What is the Skill we Learn, when we Learn Programming?

- We have studied lots of little tricks
- And we have learned that programming is just the act of breaking down big problems that we can't solve, into little problems that we can solve using the tricks we have learned

```
public void func()
{
   doSomething();
   doSomethingElse();
   doOneLastThing();
}
```

Take a "big" problem…

…and break it down into "little problems" that we can easily solve

Here are some of our Tricks

- To compare two things – use "if"
- To do something a bunch of times – "loop"
- Have a concept that has several parts? – make a class and use instance data ("has-a")
- To remember a relatively unchanging number of objects – use an Array
- To calculate a logarithm – use "Math.log"
- These all represent "little problems" that we can solve

Problem Decomposition: Turning a "big" problem into "little" ones

- Example Problem: Write a function that returns the factorial of an integer:

$$x! = x * (x-1) * (x-2) * \ldots * 3 * 2 * 1$$

```
int factorial(int x)
{
   int answer = 1;

   for(int i = 1; i <= x; i++)
      answer *= i;

   return answer;
}
```

We naturally break the problem down, in this case to:
1 multiply operation, inside 1 loop

New Idea for Today!

- We don't have to decompose a "big" problem down only into little problems that we can solve
- Some problems can be decomposed into a smaller version of the same problem
- In this case, we don't have to solve the "big" problem or even the "smaller" problem, instead we can get away with solving a very very small version of the problem

Think about that Factorial again…

- We can rewrite the factorial equation,

$$x! = x * \underbrace{(x-1) * (x-2) * \ldots * 3 * 2 * 1}_{(x-1)!}$$

- like this instead:

$$x! = x * \underbrace{(x-1)!}$$

This formulation expresses factorial in terms of a smaller factorial…

… in other words, if we had a way to solve a smaller factorial, then it would be easy to solve this bigger one.

What would that look like in code?

- New factorial formulation:

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

- "Naive Translation" into code:

```
int factorial(int x)
{
   return x * factorial(x-1);
}
```

factorial()

So far…

- The programs we have seen all look like this…

```
public void func()
{
   doSomething();
   doSomethingElse();
   doOneLastThing();
}
```

The function "func()"
isn't called in here

… we have only seen functions that call **other** functions

- What happens if a function calls itself?

Running our "Naive Translation"

- Just to give us more information we've added some print statements. Now, let's see what happens if we run it…

```
static int factorial(int x)
{
   System.out.println("factorial(" + x + ") called!");
   return x * factorial(x-1);
}

static public void main(String[] args)
{
   int fact = factorial(10);
   System.out.println("fact = " + fact);
}
```

Output (1/2)

```
>java factorialRecursive
factorial(10) called!
factorial(9) called!
factorial(8) called!
factorial(7) called!
factorial(6) called!
factorial(5) called!
factorial(4) called!
factorial(3) called!
factorial(2) called!
factorial(1) called!
factorial(0) called!
factorial(-1) called!
factorial(-2) called!
factorial(-3) called!
factorial(-4) called!
factorial(-5) called!
factorial(-6) called!
... (more on next slide)
```

The Good News is that it works – a function can call itself!

But we didn't stop at a reasonable spot (is there anything we can do about this?)

Output (2/2)

```
factorial(-4354) called!
factorial(-4355) called!
factorial(-4356) called!
factorial(-4357) called!
factorial(-4358) called!
Exception in thread "main" java.lang.StackOverflowError
     at sun.nio.cs.SingleByteEncoder.encodeArrayLoop(Unknown Source)
     at sun.nio.cs.SingleByteEncoder.encodeLoop(Unknown Source)
     at java.nio.charset.CharsetEncoder.encode(Unknown Source)
     at sun.nio.cs.StreamEncoder.implWrite(Unknown Source)
     at sun.nio.cs.StreamEncoder.write(Unknown Source)
     at java.io.OutputStreamWriter.write(Unknown Source)
     at java.io.BufferedWriter.flushBuffer(Unknown Source)
     at java.io.PrintStream.write(Unknown Source)
     at java.io.PrintStream.print(Unknown Source)
     at java.io.PrintStream.println(Unknown Source)
     at factorialRecursive.factorial(factorialRecursive.java:6)
     at factorialRecursive.factorial(factorialRecursive.java:13)
     at factorialRecursive.factorial(factorialRecursive.java:13)
     at factorialRecursive.factorial(factorialRecursive.java:13)
     at factorialRecursive.factorial(factorialRecursive.java:13)
(the last line repeats about 4000 times) ...
```

What did we see?

- A function that calls itself works!
- But it didn't terminate… it just kept going
- Eventually the Java Virtual Machine ran out of "Stack" space
  - We'll talk about the Stack in a few slides
- Question: How can we make the program terminate?

Terminating…

- Large values of factorial can be daunting, but the smallest values are easy:

2! = 2

1! = 1

0! = 1

- Remember we said we could get away with: "solving a very very small version of the problem"? Let's use: **0!** = **1**

Terminating…

- Let's add one "if" statement to check for the factorial of zero…

```
static int factorial(int x)
{
   System.out.println("factorial(" + x + ") called!");

   if(x == 0)
      return 1;
   else
      return x * factorial(x-1);
}

static public void main(String[] args)
{
   int fact = factorial(10);
   System.out.println("fact = " + fact);
}
```

Output of Improved Version:

```
>java factorialRecursive
factorial(10) called!
factorial(9) called!
factorial(8) called!
factorial(7) called!
factorial(6) called!
factorial(5) called!
factorial(4) called!
factorial(3) called!
factorial(2) called!
factorial(1) called!
factorial(0) called!
fact = 3628800
```

It worked!

How do the Result values get Returned?

- Same functionality, more print statements…

```
static int factorial(int x)
{
   System.out.println("factorial(" + x + ") called!");

   if(x == 0)
   {
      System.out.println("factorial(0) returned: 1");
      return 1;
   }
   else
   {
      int retval =  x * factorial(x-1);
      System.out.println("factorial(" + x + ")"
                         + " returned: " + retval);
      return retval;
   }
}
```

Output of Improved Version

```
>java factorialRecursiveVerbose
factorial(10) called!
factorial(9) called!
factorial(8) called!
factorial(7) called!
factorial(6) called!
factorial(5) called!
factorial(4) called!
factorial(3) called!
factorial(2) called!
factorial(1) called!
factorial(0) called!
factorial(0) returned: 1
factorial(1) returned: 1
factorial(2) returned: 2
factorial(3) returned: 6
factorial(4) returned: 24
factorial(5) returned: 120
factorial(6) returned: 720
factorial(7) returned: 5040
factorial(8) returned: 40320
factorial(9) returned: 362880
factorial(10) returned: 3628800
fact = 3628800
```

- Same functionality, more print statements…
- Shows both the calls recursing down to the terminating case
- Then the returns recursing back out to the answer

Let's Re-examine the Code

```
static int factorial(int x)
{
   if(x == 0)
      return 1;
   else
      return x * factorial(x-1);
}


static public void main(String[] args)
{
   int fact = factorial(10);
   System.out.println("fact = " + fact);
}
```

The "Termination Condition" or "Base Case"

Formulation of the "big" problem in terms of a "smaller" version, the "Recursive Case"

Definition of Recursion

- A function is **Recursive** if it calls itself (directly or indirectly) from within its own body
- The two components of a Recursive Solution:
  1. A solution to the problem that involves a simpler instance of the problem (called the "**Recursive Case**")
  2. A **Direct Solution** to a simple version of the problem (called the "**Termination Case**", or "**Base Case**")
- Also, notice that we have seen two solutions to the factorial problem – one that is recursive, and one that is not recursive ("iterative")…

Iterative versus Recursive Solutions

```
int factorial(int x)
{
   if(x == 0)
      return 1;
   else
      return x * factorial(x-1);
}
```

Recursive Solution

```
int factorial(int x)
{
   int answer = 1;

   for(int i = 1; i <= x; i++)
      answer *= i;

   return answer;
}
```

Iterative Solution

Iterative versus Recursive Solutions

- There is nothing particularly special about Recursion versus Iteration
- Any recursive algorithm can be converted into an iterative algorithm, and vice-versa
- The decision regarding whether to apply iteration or recursion depends upon the nature of the problem
  - Most problems are best approached with iteration
  - But some problems are simpler to approach with recursion

Advantages and Disadvantages of Recursion

- Advantages
  - Some problems can be coded much more simply with recursion (we'll see more examples soon)
  - Some other languages are optimised for recursion ("functional languages", like Haskell, Erlang, Lisp)
- Disadvantages
  - Can use more memory than Iteration (in particular, can use more valuable execution stack memory)
  - Can be less efficient (poorly designed recursion can cause intermediate values to be calculated more than once)

CSE1030 – Lecture #19

- Introduction to Recursion
- Execution Stack
- Example: Reversing a String
- Example: Mathematical Bisection
- We're Done!

How is Recursion Possible?

- Next we are going to discuss a mechanism that the Java Virtual Machine uses when executing programs
- There is a thing called an "Execution Stack" that is used to keep the variables within functions separate from one another (remember "Variable Scope"?)
- It is the Execution Stack that makes recursion possible

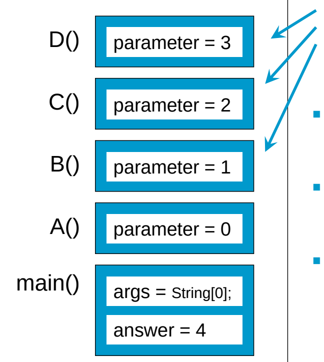
How does Java keep the "parameter" variables straight?

```
class example
{
   public static void main(String[] args)
   {
      int answer = A(0);
      System.out.println("answer = " + answer);
   }


   static int A(int parameter) { return B(parameter + 1); }

   static int B(int parameter) { return C(parameter + 1); }

   static int C(int parameter) { return D(parameter + 1); }

   static int D(int parameter) { return parameter + 1; }
}
```


Example – annotated.java

```
>java annotated
Calling: A(0)
  A(0) called
    B(1) called
      C(2) called
        D(3) called
        D returns 4
      C returns 4
    B returns 4
  A returns 4
answer = 4
```


Execution Stack



- Every time that Java starts a new function, it creates a "stack frame", a unique place to hold the local variables for that function
- When a function returns, the stack frame goes away
- This is called the "execution stack"
- Consequently, in this example each function has their own distinct variable called "parameter"

Recursive Execution Stack Example

fact() x = 0

fact() x = 1

fact() x = 2

fact() x = 3

fact() x = 4

main() args = String[0]; fact = 24

- Every time we recurse, Java creates a new stack frame, within which the variables exist.
- This is how recursion works.

```
int fact(int x)
{
   if(x == 0)
      return 1;
   else
      return x * fact(x-1);
}

public void main(String[] args)
{
   int f = fact(4);
   System.out.println("fact = " + f);
}
```


CSE1030 – Lecture #19

- Introduction to Recursion
- Execution Stack
- Example: Reversing a String
- Example: Mathematical Bisection
- We're Done!

Example: Reversing a String

- main():

```
class reverseRecursive
{
   static String reverse(String s)
   ...


   static public void main(String[] args)
   {
      String before = "ABCDEFG";
      String after = reverse(before);

      System.out.println("after = " + after);
   }
}
```

- Output:

```
>java reverseRecursive
after = GFEDCBA
```


Reversing a String

```
static String reverse(String s)
{
   if(s.length() == 0)
      return "";

   return reverse(s.substring(1))
                    + s.charAt(0);
}
```

Recursive Solution

```
static String reverse(String s)
{
   String retval = "";

   for(int i = s.length() - 1; i >= 0; i--)
      retval = retval + s.charAt(i);

   return retval;
}
```

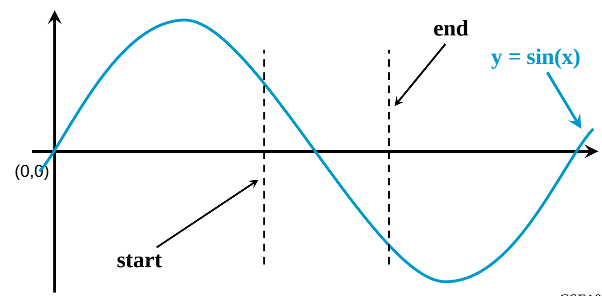
Iterative Solution

CSE1030 – Lecture #19

- Introduction to Recursion
- Execution Stack
- Example: Reversing a String
- Example: Mathematical Bisection
- We're Done!

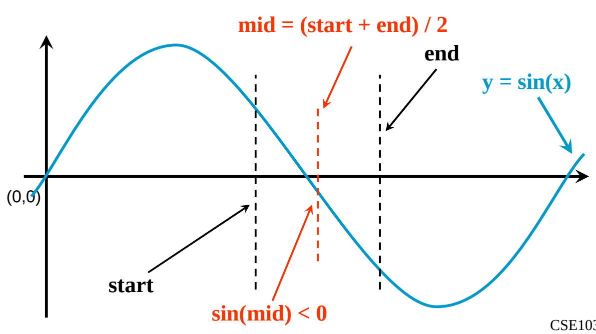
Mathematical Bisection

- Bisection is a technique used to find the point where a function crosses zero (to find **x** where **f(x) = 0**)
- We sandwich the zero between two points (**start** & **end**)



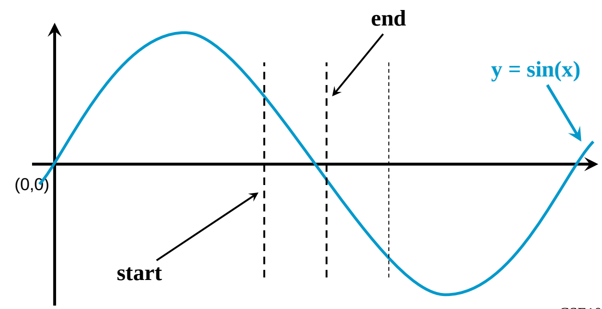
Mathematical Bisection

- Then we choose a value between **start** & **end** and check whether the function is positive or negative there



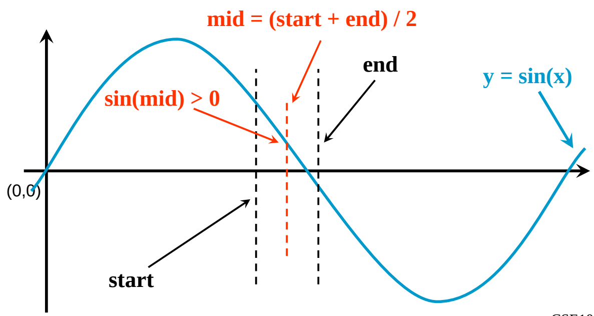
Mathematical Bisection

- Because **sin(mid)** < **0**, we should move the **end** point inwards…

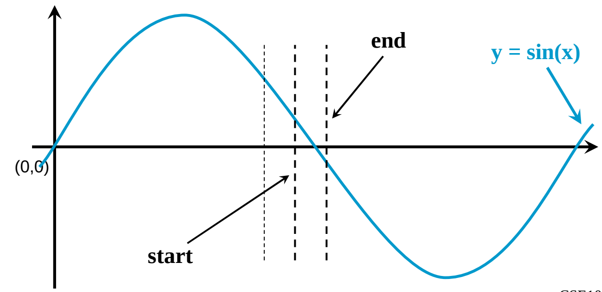


Mathematical Bisection

- And we repeat…
- This time, **sin(mid)** > **0**, so we move the **start** point in…

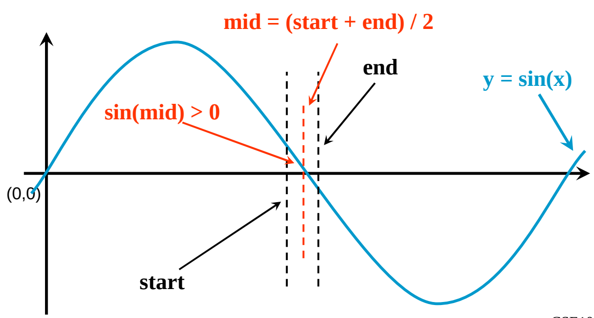


Mathematical Bisection



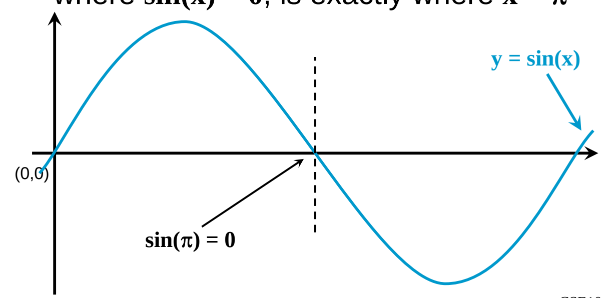
Mathematical Bisection

- Continue repeating the process until **start** & **end** are close enough together that we have achieved the desired accuracy



Mathematical Bisection

- We can use bisection to calculate the value of π, because the point in $x \in [2,4]$ where **sin(x)** = **0**, is exactly where **x** = π



Mathematical Bisection: Code

```
class zero
{
   static double f(double x)
   {
      return Math.sin(x);
   }

   static final double errorTolerance = 0.000000000000001;

   static public void main(String[] args)
   {
      System.out.println(" Pi = " + bisect(2.0, 4.0));
   }
}
```

Just for convenience, we'll use f(x) to denote the function that we are zeroing

How much accuracy do we want? How close should start & end get?


```
static double bisect(double start, double end)
{
   double mid = (start + end) / 2.0;

   if(Math.abs(start - end) < errorTolerance)
      return mid;

   if(f(mid) > 0)
      return bisect(mid, end);
   else
      return bisect(start, mid);
}
```

Recursive Solution


```
static double bisect(double start, double end)
{
   double mid = 0;

   while(Math.abs(start - end) > errorTolerance)
   {
      mid = (start + end) / 2.0;

      if(f(mid) > 0)
         start = mid;
      else
         end = mid;
   }

   return mid;
}
```

Iterative Solution

Output

```
>java zeroRecursive
 Pi = 3.1415926535897936
```

- With this program we calculate the digits of Pi
- This algorithm can be used to achieve any accuracy we want (by reducing the value of the "errorTolerance"
- Notice that in this example (as in the previous examples) the recursive solution is shorter

CSE1030 – Lecture #19

- Introduction to Recursion
- Execution Stack
- Example: Reversing a String
- Example: Mathematical Bisection
- We're Done!

Next topic…

Recursion I