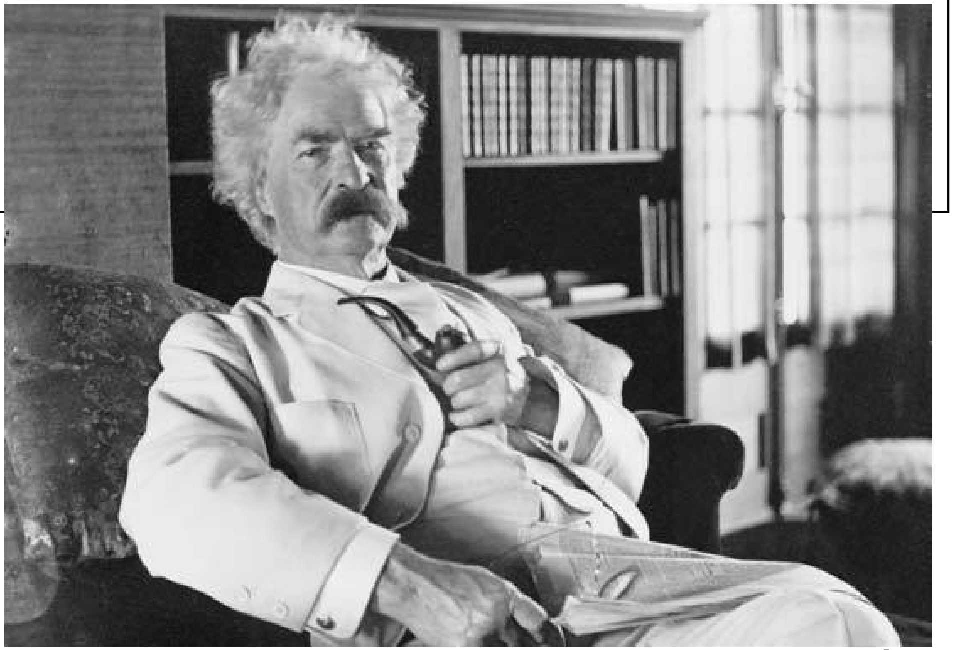
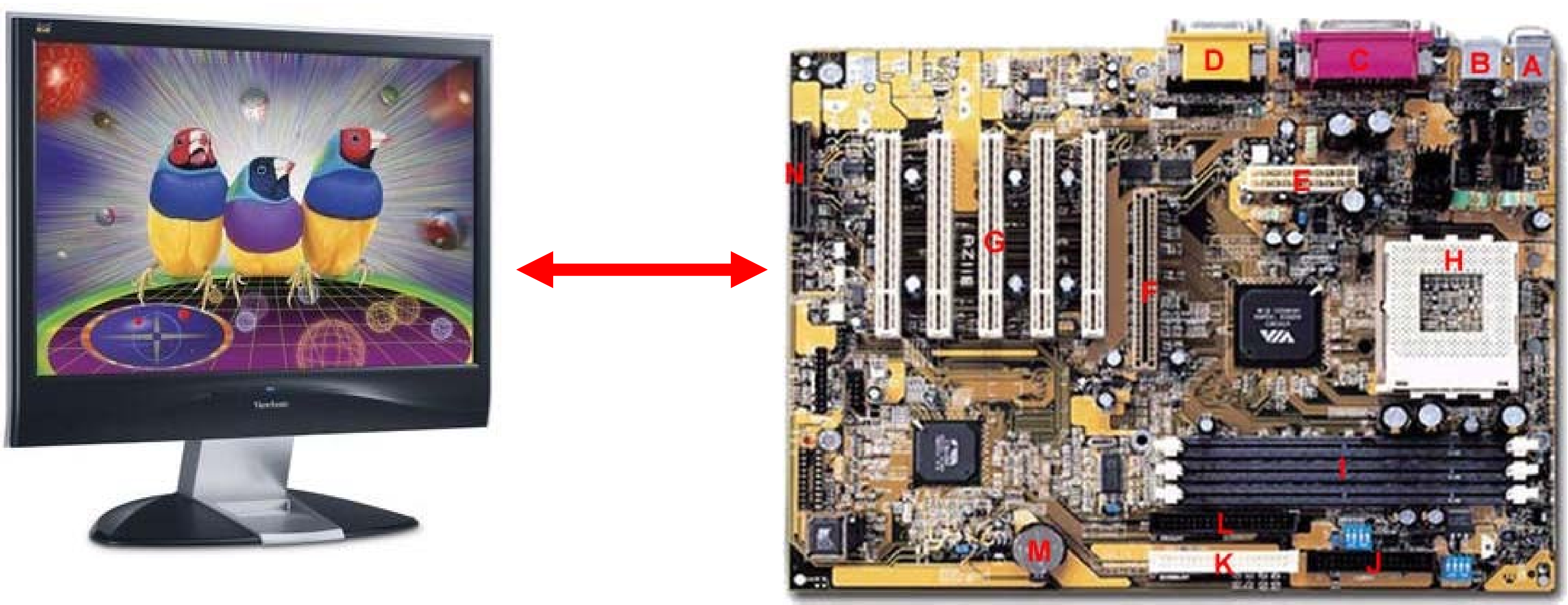
Topic 9
Introduction to Recursion

"To a man with a hammer, everything looks like a nail"

-Mark Twain



Underneath the Hood.



The Program Stack

- When you invoke a method in your code what happens when that method is completed?

```
FooObject f = new FooObject();
int x = 3;
f.someFooMethod(x);
f.someBarMethod(x);
```

- How does that happen?
- What makes it possible?

Methods for Illustration

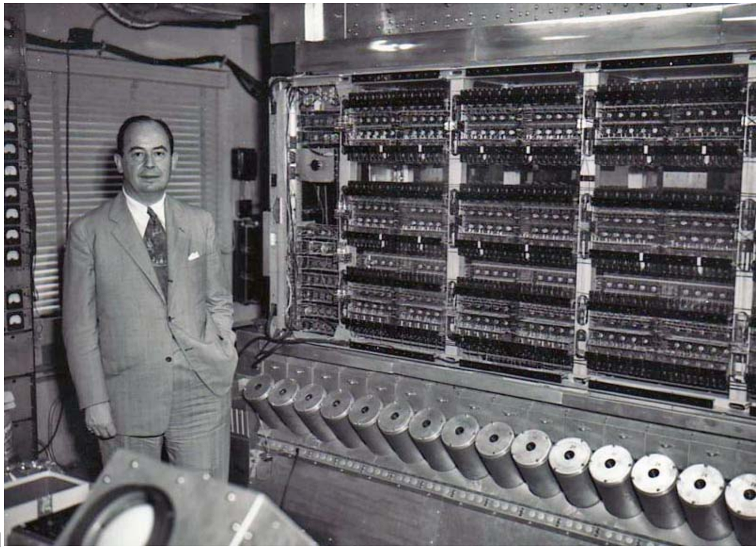
```
public void someFooMethod(int z)
{       int x = 2 * z;
        System.out.println(x);
}

public void someBarMethod(int y)
{       int x = 3 * y;
        someFooMethod(x);
        System.out.println(x);
}
```

The Program Stack

- When your program is executed on a processor the commands are converted into another set of instructions and assigned memory locations.
  - normally a great deal of expansion takes place

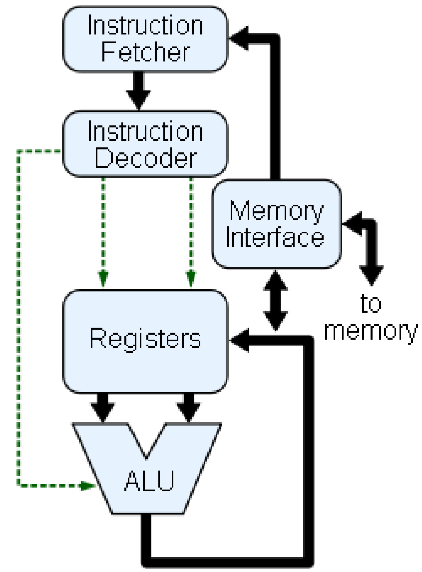
```
101 FooObject f = new FooObject();
102 int x = 3;
103 f.someFooMethod(x);
104 f.someBarMethod(x);
```

- Von Neumann Architecture



Basic CPU Operations

- A CPU works via a fetch command / execute command loop and a program counter
- Instructions stored in memory (Just like data!)




```
101 FooObject f = new FooObject();
102 int x = 3;
103 f.someFooMethod(x);
104 f.someBarMethod(x);
```

- What if `someFooMethod` is stored at memory location 200?

More on the Program Stack

```
101 FooObject f = new FooObject();
102 int x = 3;
103 f.someFooMethod(x);
104 f.someBarMethod(x);
```

- Line 103 is really saying *go to line 200 with f as the implicit parameter and x as the explicit parameter*
- When someFooMethod is done what happens?

A. Program ends   B. goes to line 103

C. Goes back to whatever method called it

Activation Records and the Program Stack

- When a method is invoked all the relevant information about the current method (variables, values of variables, next line of code to be executed) is placed in an *activation record*
- The activation record is *pushed* onto the *program stack*
- A *stack* is a data structure with a single access point, the *top.*

The Program Stack

- Data may either be added (*pushed*) or removed *(popped)* from a stack but it is always from the top.
  - A stack of dishes
  - which dish do we have easy access to?

top

Using Recursion

A Problem

- Write a method that determines how much space is take up by the files in a directory
- A directory can contain files and directories
- How many directories does our code have to examine?
- How would you add up the space taken up by the files in a single directory
  - Hint: don't worry about any sub directories at first
- `Directory` and `File` classes
- in the `Directory` class:

```
public File[] getFiles()
public Directory[] getSubdirectories()
```

- in the `File` class

```
public int getSize()
```

Attendance Question 2

- How many levels of directories have to be visited?

A. 0

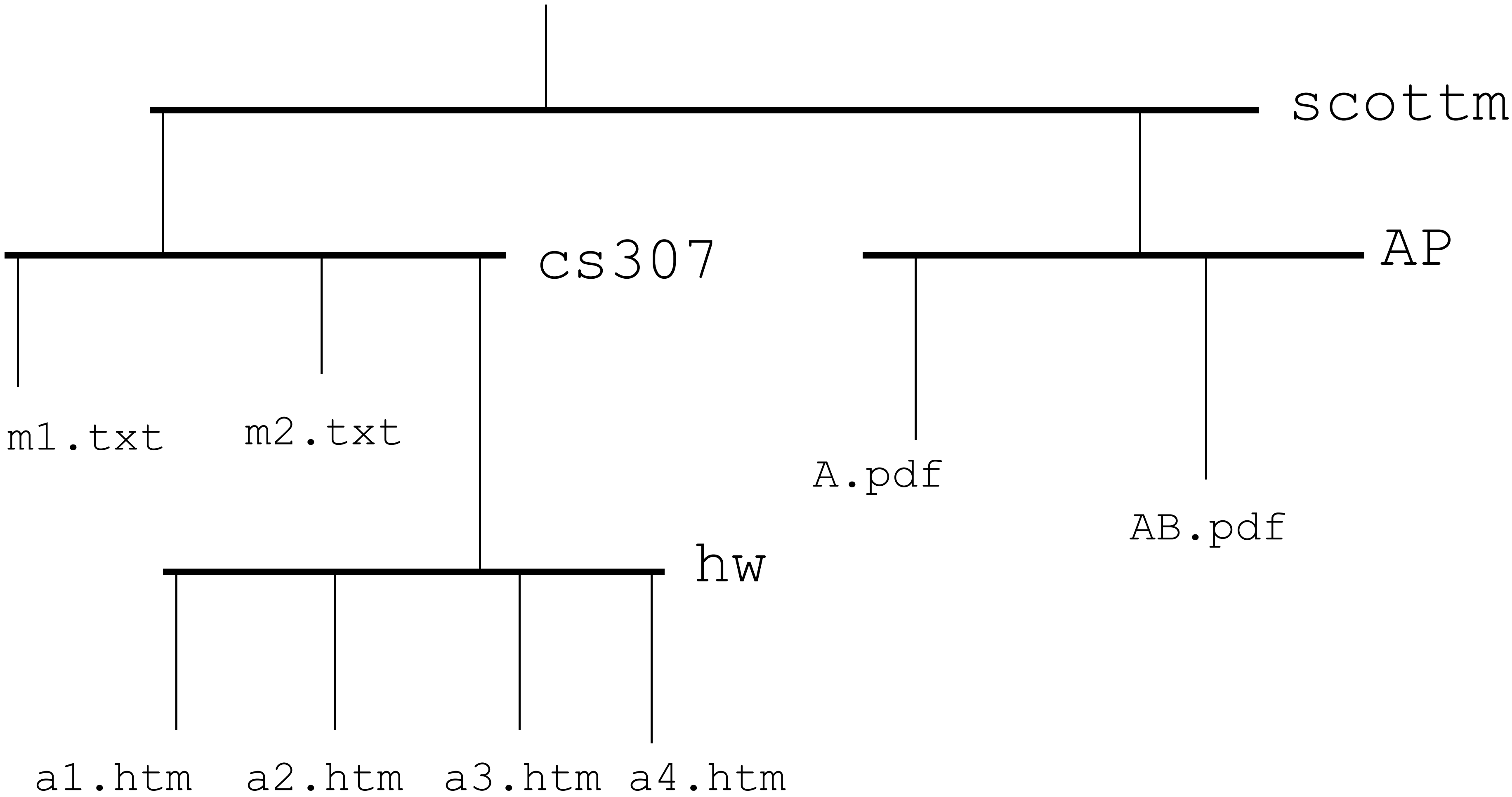
B. Unknown

C. Infinite

D. 1

E. 8

Sample Directory Structure



Code for `getDirectorySpace()`

```
public int getDirectorySpace(Directory d)
{ int total = 0;
  File[] fileList = d.getFiles();
  for(int i = 0; i < fileList.length; i++)
     total += fileList[i].getSize();
  Directory[] dirList = d.getSubdirectories();
  for(int i = 0; i < dirList.length; i++)
     total += getDirectorySpace(dirList[i]);
  return total;
}
```

Attendance Question 3

- Is it possible to write a non recursive method to do this?

A. Yes

B. No

Iterative `getDirectorySpace()`

```
public int getDirectorySpace(Directory d)
{  ArrayList dirs = new ArrayList();
   File[] fileList;
   Directory[] dirList;
   dirs.add(d);
   Directory temp;
   int total = 0;
   while( ! dirs.isEmpty() )
   {   temp = (Directory)dirs.remove(0);
       fileList = temp.getFiles();
       for(int i = 0; i < fileList.length; i++)
             total += fileList[i].getSize();
       dirList = temp.getSubdirectories();
       for(int i =0; i < dirList.length; i++)
             dirs.add( dirList[i] );
   }
   return total;
}
```

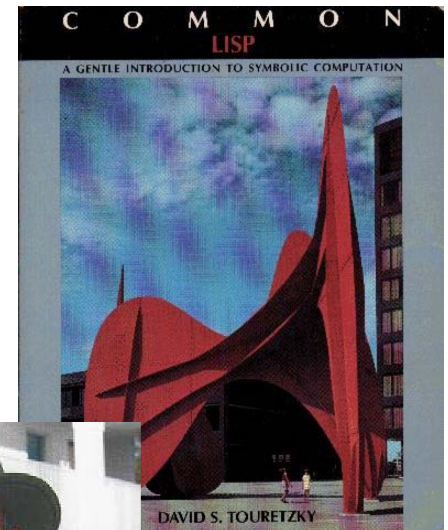
Simple Recursion Examples

Wisdom for Writing Recursive Methods

The 3 plus 1 rules of Recursion

1. Know when to stop
2. Decide how to take one step
3. Break the journey down into that step and a smaller journey
4. Have faith

From *Common Lisp: A Gentle Introduction to Symbolic Computation* by David Touretzky



Writing Recursive Methods

- Rules of Recursion
  1. Base Case: Always have at least one case that can be solved without using recursion
  2. Make Progress: Any recursive call must progress toward a base case.
  3. "You gotta believe." Always assume that the recursive call works. (Of course you will have to design it and test it to see if it works or prove that it always works.)

A recursive solution solves a small part of the problem and leaves the rest of the problem in the same form as the original

N!

- the classic first recursion problem / example
- N!

  5! = 5 * 4 * 3 * 2 * 1 = 120

```
int res = 1;
for(int i = 2; i <= n; i++)
      res *= i;
```

Factorial Recursively

- Mathematical Definition of Factorial

```
0! = 1
N! = N * (N - 1)!
```

The definition is recursive.

```
// pre n >= 0
public int fact(int n)
{ if(n == 0)
      return 1;
  else
      return n * fact(n-1);
}
```

Big O and Recursion

- Determining the Big O of recursive methods can be tricky.
- A *recurrence relation* exits if the function is defined recursively.
- The T(N), actual running time, for N! is recursive
- $T(N)_{fact} = T(N-1)_{fact} + O(1)$
- This turns out to be O(N)
  – There are N steps involved

Common Recurrence Relations

- $T(N) = T(N/2) + O(1) \rightarrow O(\log N)$
  - binary search
- $T(N) = T(N-1) + O(1) \rightarrow O(N)$
  - sequential search, factorial
- $T(N) = T(N/2) + T(N/2) + O(1) \rightarrow O(N)$,
  - tree traversal
- $T(N) = T(N-1) + O(N) \rightarrow O(N^2)$
  - selection sort
- $T(N) = T(N/2) + T(N/2) + O(N) \rightarrow O(N\log N)$
  - merge sort
- $T(N) = T(N-1) + T(N-1) + O(1) \rightarrow O(2^N)$
  - Fibonacci

Tracing Fact With the Program Stack

System.out.println(fact(4));

top → System.out.println(fact(4));

Calling fact with 4

n $\boxed{4}$ in method fact

partial result = n * fact(n-1)

top →

```
System.out.println( fact(4) );
```

Calling fact with 3

n 3 in method fact

partial result = n * fact(n-1)

top

n 4 in method fact

partial result = n * fact(n-1)

System.out.println(fact(4));

Calling fact with 2

n 2 in method fact

partial result = n * fact(n-1)

top →

n 3 in method fact

partial result = n * fact(n-1)

n 4 in method fact

partial result = n * fact(n-1)

System.out.println(fact(4));

Calling fact with 1

n 1 in method fact

partial result = n * fact(n-1)

top →

n 2 in method fact partial result = n * fact(n-1)
n 3 in method fact partial result = n * fact(n-1)
n 4 in method fact partial result = n * fact(n-1)
System.out.println(fact(4));

Calling fact with 0 and returning 1

n 0 in method fact

returning 1 to whatever method called me

top

n 1 in method fact

partial result = n * fact(n-1)

n 2 in method fact

partial result = n * fact(n-1)

n 3 in method fact

partial result = n * fact(n-1)

n 4 in method fact

partial result = n * fact(n-1)

System.out.println(fact(4));

Returning 1 from fact(1)

n 1 in method fact
partial result = n * 1,
return 1 to whatever method called me

top →

n 2 in method fact partial result = n * fact(n-1)
n 3 in method fact partial result = n * fact(n-1)
n 4 in method fact partial result = n * fact(n-1)
System.out.println(fact(4));

Returning 2 from fact(2)

n 2 in method fact

partial result = 2 * 1,
return 2 to whatever method called me

top

n 3 in method fact

partial result = n * fact(n-1)

n 4 in method fact

partial result = n * fact(n-1)

System.out.println(fact(4));

Returning 6 from fact(3)

n 3 in method fact

partial result = 3 * 2,
return 6 to whatever method called me

top

n 4 in method fact

partial result = n * fact(n-1)

System.out.println(fact(4));

Returning 24 from fact(4)

n | 4 | in method fact

partial result = 4 * 6,
return 24 to whatever method called me

top → | System.out.println(fact(4)); |

Calling System.out.println

System.out.println(24);

top

??

Evaluating Recursive Methods

Evaluating Recursive Methods

- you must be able to evaluate recursive methods

```
public static int mystery (int n){
    if( n == 0 )
        return 1;
    else
        return 3 * mystery(n-1);
}
// what is returned by mystery(5)
```

Evaluating Recursive Methods

- Draw the program stack!

  m(5) = 3 * m(4)

  m(4) = 3 * m(3)

  m(3) = 3 * m(2)

  m(2) = 3 * m(1)

  m(1) = 3 * m(0)

  m(0) = 1

  -> 3^5 = 243
- with practice you can see the result

Attendance Question 4

- What is returned by `mystery(-3)` ?

A. 0

B. 1

C. Infinite loop

D. Syntax error

E. Runtime error due to stack overflow

Evaluating Recursive Methods

- What about multiple recursive calls?

```
public static int bar(int n){
    if( n <= 0 )
        return 2;
    else
        return 3 + bar(n-1) + bar(n-2);
}
```

- Draw the program stack and REMEMBER your work

Evaluating Recursive Methods

- What is returned by `bar(5)`?

$b(5) = 3 + b(4) + b(3)$

$b(4) = 3 + b(3) + b(2)$

$b(3) = 3 + b(2) + b(1)$

$b(2) = 3 + b(1) + b(0)$

$b(1) = 3 + b(0) + b(-1)$

$b(0) = 2$

$b(-1) = 2$

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + b(4) + b(3)

b(4) = 3 + b(3) + b(2)

b(3) = 3 + b(2) + b(1)

b(2) = 3 + b(1) + b(0) //substitute in results

b(1) = 3 + 2 + 2 = 7

b(0) = 2

b(-1) = 2

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + b(4) + b(3)

b(4) = 3 + b(3) + b(2)

b(3) = 3 + b(2) + b(1)

b(2) = 3 + 7 + 2 =12

b(1) = 7

b(0) = 2

b(-1) = 2

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + b(4) + b(3)

b(4) = 3 + b(3) + b(2)

b(3) = 3 + 12 + 7 = 22

b(2) = 12

b(1) = 7

b(0) = 2

b(-1) = 2

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + b(4) + b(3)

b(4) = 3 + 22 + 12 = 37

b(3) = 22

b(2) = 12

b(1) = 7

b(0) = 2

b(-1) = 2

Evaluating Recursive Methods

- What is returned by `bar(5)`?

b(5) = 3 + 37 + 22 = 62

b(4) = 37

b(3) = 22

b(2) = 12

b(1) = 7

b(0) = 2

b(-1) = 2

Unplugged Activity

- Double the number of pieces of candy in a bowl.
- Only commands we know are:
  - take one candy out of bowl and put into infinite supply
  - take one candy from infinite supply and place in bowl
  - do nothing
  - double the number of pieces of candy in the bowl
- Thanks Stuart Reges

Recursion Practice

- Write a method raiseToPower(int base, int power)
- //pre: power >= 0

- Tail recursion refers to a method where the recursive call is the last thing in the method

Finding the Maximum in an Array

- public int max(int[] values){
- Helper method or create smaller arrays each time

Attendance Question 5

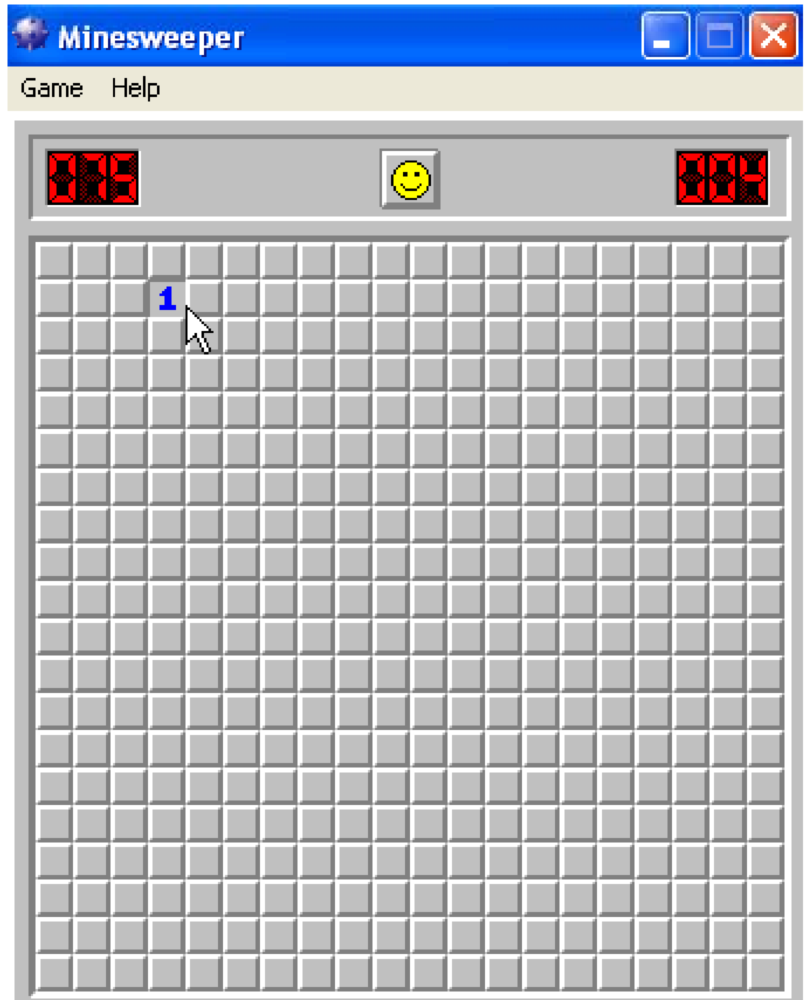
- When writing recursive methods what should be done first?

A. Determine recursive case
B. Determine recursive step
C. Make recursive call
D. Determine base case(s)
E. Determine Big O

Your Meta Cognitive State

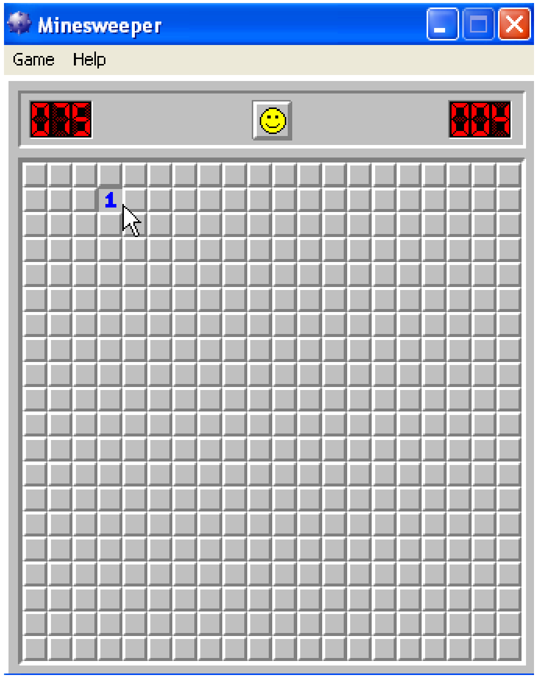
- Remember we are learning to use a tool.
- It is not a good tool for ***all*** problems.
  - In fact we will implement several algorithms and methods where an iterative (looping without recursion) solution would work just fine
- After learning the mechanics and basics of recursion the real skill is knowing what problems or class of problems to apply it to

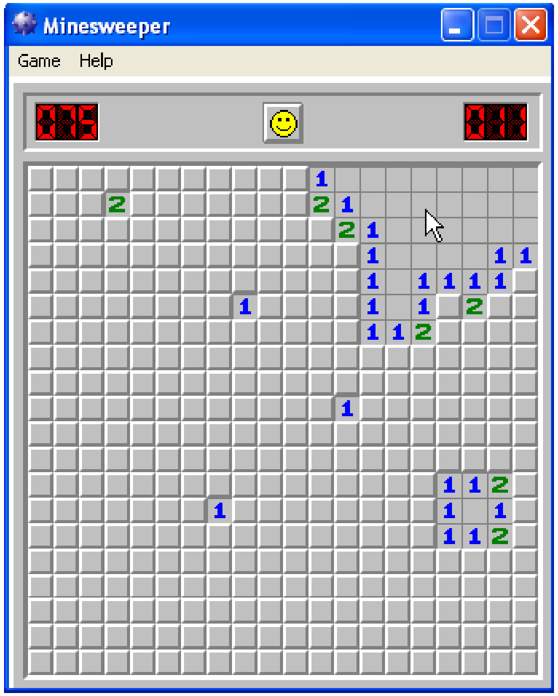
A Harder(??) Problem



Mine Sweeper

- Game made popular due to its inclusion with Windows (from 3.1 on)
- What happens when you click on a cell that has 0 (zero) mines bordering it?





Result of clicking marked cell.

The update method

- Initially called with the x and y coordinates of a cell with a 0 inside it meaning the cell does not have any bombs bordering it.
- Must reveal all cells neighboring this one and if any of them are 0s do the same thing

-1 indicates a mine in that cell

2	-1	2	0	0	0
2	-1	3	2	2	1
1	1	3	-1	-1	1
0	0	2	-1	3	1
0	0	1	1	1	0
0	0	0	0	0	0

Update Code