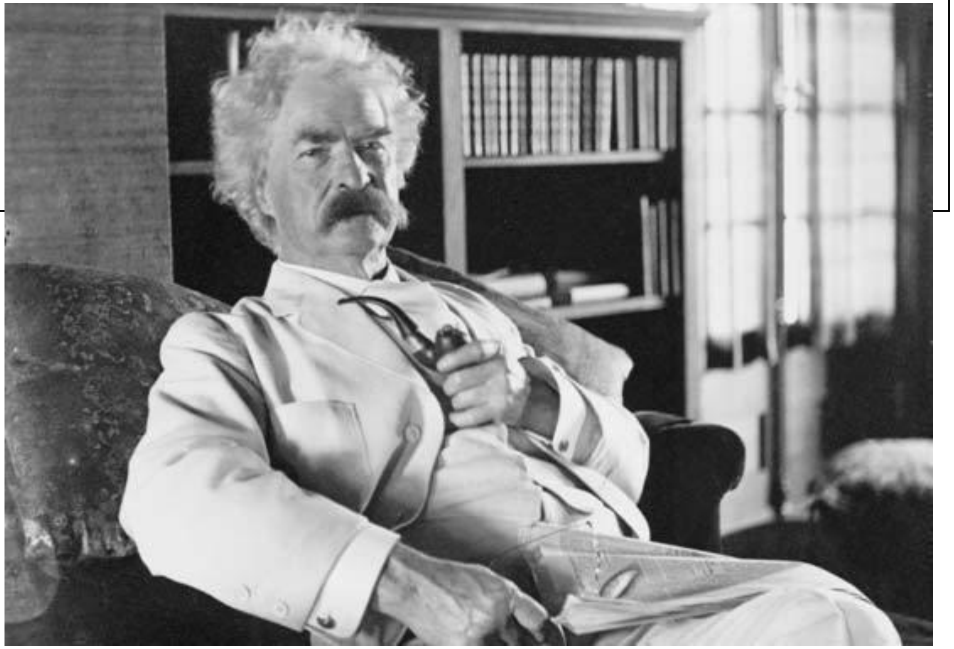
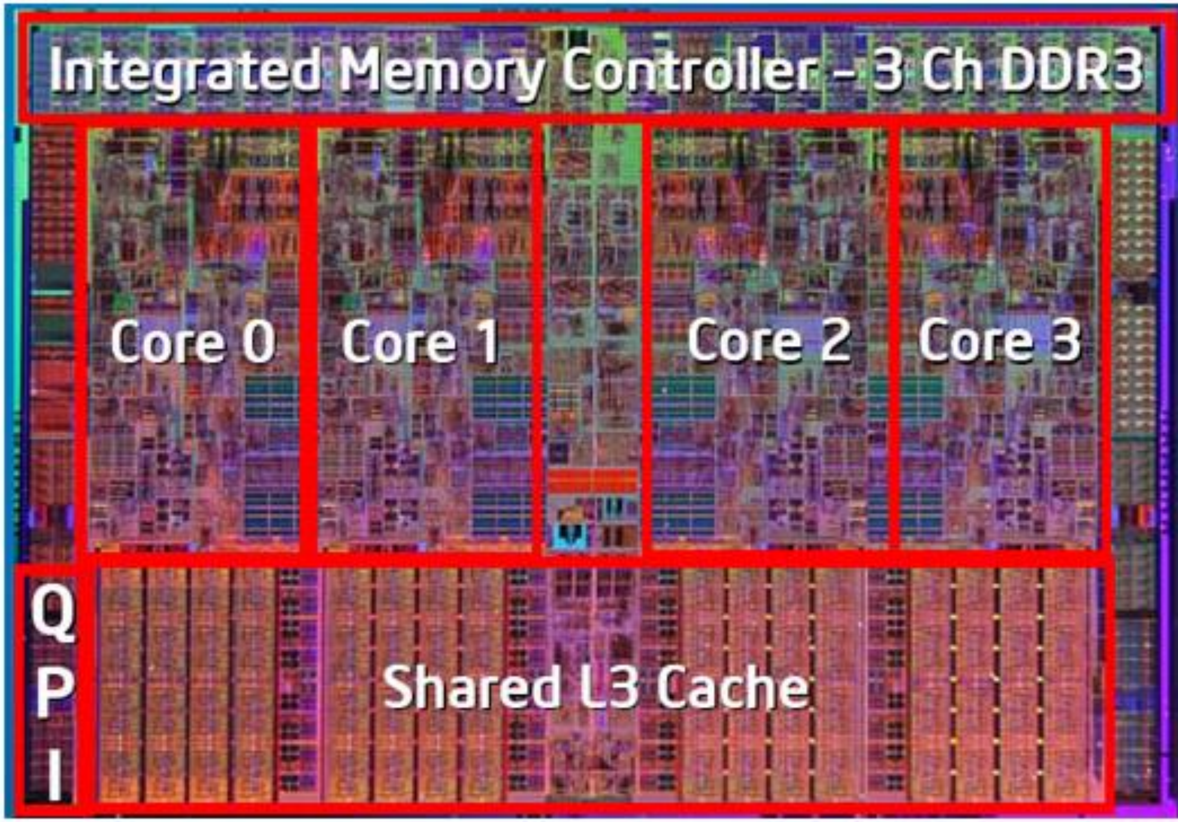
Topic 12
Introduction to Recursion

"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 done?

```
public class Mice {
    public static void main(String[] args) {
        int x = 37;
        int y = 12;
        method1(x, y);
        int z = 73;
        int m1 = method1(z, x);
        method2(x, x);
    }

    // method1 and method2
    // on next slide
```

method1 and method2

```
// in class Mice
public static int method1(int a, int b) {
     int r = 0;
     if (b != 0) {
          int x = a / b;
          int y = a % b;
          r = x + y;
     }
     return r;
}
public static void method2(int x, int y) {
     x++;
     y--;
     int z = method1(y, x);
     System.out.print(z);
}
```

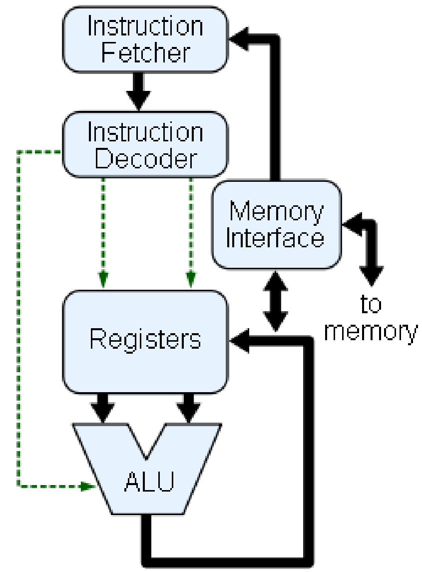
The Program Stack

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

```
public static void main(String[] args) {
    int x = 37;  // 0
    int y = 12;  // 1
    method1(x, y);  // 2
    int z = 73;  // 3
    int m1 = method1(z, x);  // 4
    method2(x, x);  // 7
}
```

Basic CPU Operations

- A CPU works via a fetch command / execute command loop and a program counter



- Instructions stored in memory (Instructions are data!)

```
int x = 37;  // 0
int y = 12;  // 1
method1(x, y);  // 2
int z = 73;  // 3
int m1 = method1(z, x);  // 4
method2(x, x);  // 5
```

- What if the first instruction of the method1 is stored at memory location 50?

```
// in class Mice
public static int method1(int a, int b) {
     int r = 0; // 51
     if (b != 0) { // 52
          int x = a / b; // 53
          int y = a % b; // 54
          r = x + y; // 55
     }
     return r; // 56
}
public static void method2(int x, int y) {
     x++; // 60
     y--; // 61
     int z = method1(y, x); // 62
     System.out.print(z); // 63
}
```

Clicker 1 - The Program Stack

```
int x = 37;  // 1
int y = 12;  // 2
method1(x, y);  // 3
int z = 73;  // 4
int m1 = method1(z, x);  // 5
method2(x, x);  // 6
```

- Instruction 3 is really saying *jump to instruction 50 with parameters x and y*
- ***In general*** what happens when method1 finishes?

A. program ends   B. goes to instruction 4

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

Clicker 2

- How many levels of directories have to be visited?

A. 0

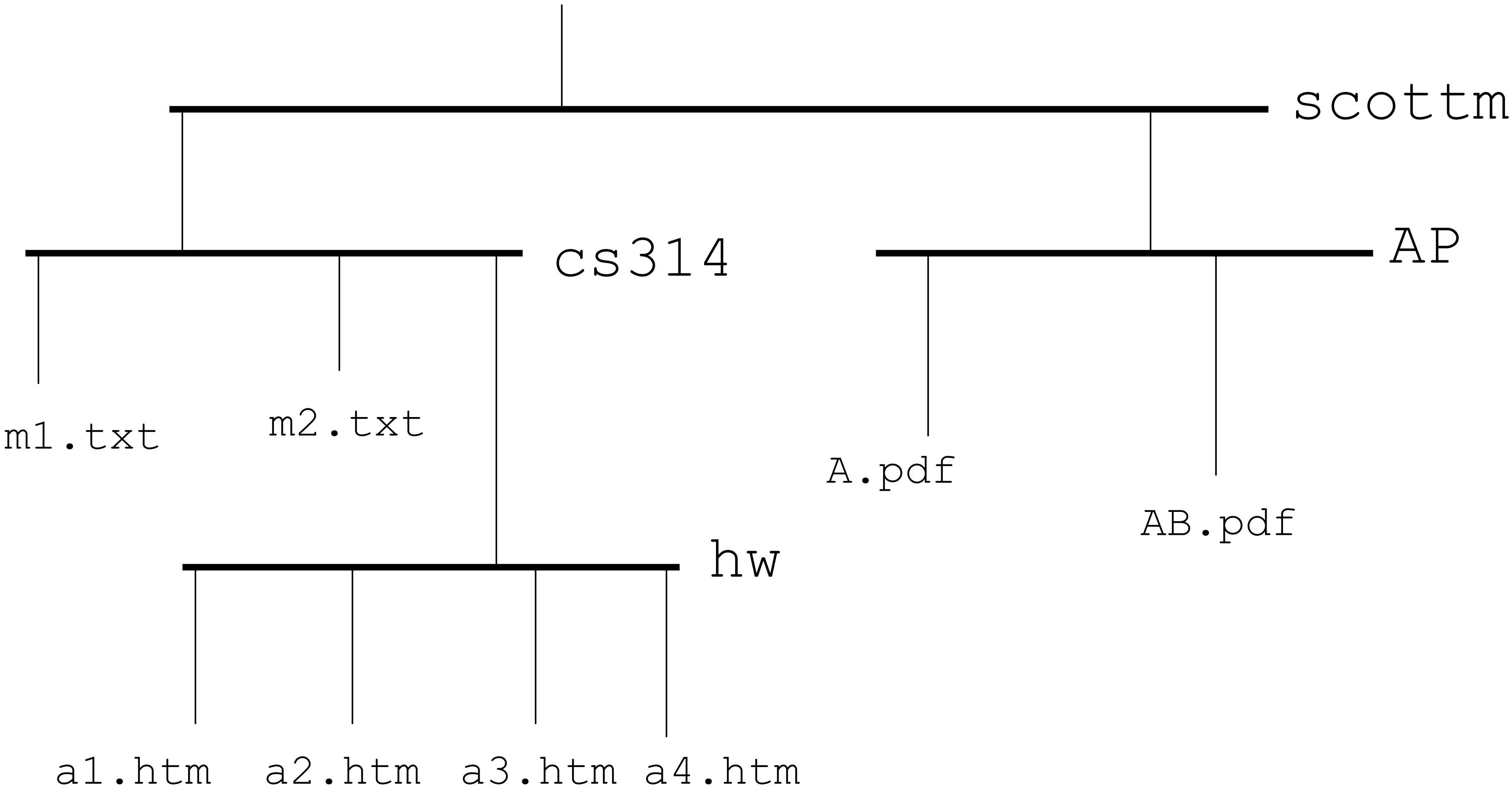
B. 1

C. 8

D. Infinite

E. Unknown

Sample Directory Structure



Java File Class

- **`File`**`(String pathname)` Creates a new File instance by converting the given pathname.
- `boolean isDirectory()` Tests whether the file denoted by this abstract pathname is a directory.
- `File[] listFiles()` Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.

Code for `getDirectorySpace()`

```
// pre: dir is a directory and dir != null
public static long spaceUsed(File dir) {
  if( dir == null || !dir.isDirectory())
      throw new IllegalArgumentException();
  long spaceUsed = 0;
  File[] subFilesAndDirs = dir.listFiles();
  if(subFilesAndDirs != null)
    for(File sub : subFilesAndDirs)
      if(sub != null)
        if(sub.isFile()) // sub is a plain old file
          spaceUsed += sub.length();
        else if (sub.isDirectory())
            // else sub is a directory
          spaceUsed += spaceUsed(sub);
  return spaceUsed;
}
```

Clicker 3

- Is it possible to write a non recursive method to determine space taken up by files in a directory, including its subdirectories, and their subdirectories, and their subdirectories, and so forth?

A. No

B. Yes

C. It Depends

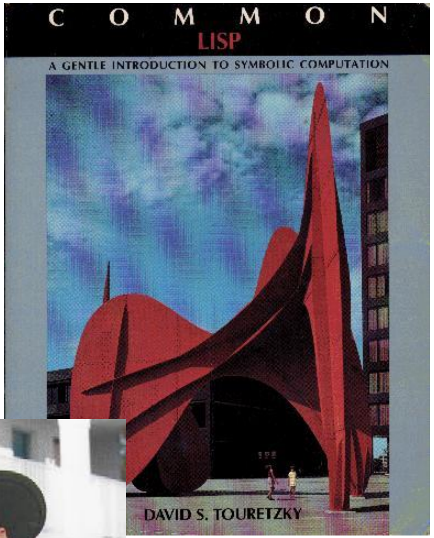
Iterative `getDirectorySpace()`

```
public long getDirectorySpace(File d) {
  ArrayList<File> dirs = new ArrayList<>();
  dirs.add(d);
  long total = 0;
  while (dirs.size() > 0) {
     File temp = dirs.remove(dirs.size() - 1);
     File[] filesAndSubs = temp.listFiles();
     if (filesAndSubs != null) {
         for (File f : filesAndSubs) {
             if (f != null) {
                 if (f.isFile())
                     total += f.length();
                 else if (f.isDirectory())
                     dirs.add(f);
             }
         }
  }
  return total;
}
```

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
- for N >= 0, N! is:

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

The definition is recursive.

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

Tracing Fact With the Program Stack

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

top → System.out.println(fact(4));

Calling fact with 4

n | 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)

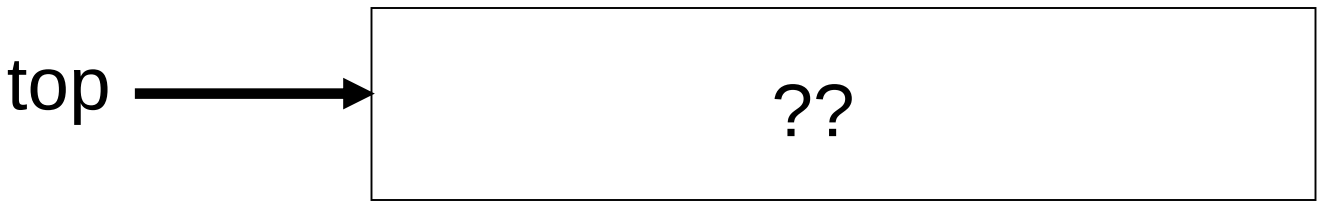
n | 4 | in method fact

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);



Evaluating Recursive Methods

Evaluating Recursive Methods

- you must be able to evaluate recursive methods

```
public static int mystery (int n){
    if(n == 0)
        return 2;
    else
        return 3 * mystery(n-1);
}
// what is returned by mystery(3)
```

Evaluating Recursive Methods

- Draw the program stack!

  m(3) = 3 * m(2) -> 3 * 18 = 54

  m(2) = 3 * m(1) -> 3 * 6 = 18

  m(1) = 3 * m(0) -> 3 * 2 = 6

  m(0) = 2

  -> 54

- with practice you can see the result

Clicker 4

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

A. 0

B. 1

C. Infinite loop

D. Syntax error

E. Runtime error

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

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);
}
```

- Clicker 5 - What does bar(4) return?

A. 2 B. 3 C. 12 D. 22 E. 37

Evaluating Recursive Methods

- What is returned by `bar(4)`?

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(4)`?

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(4)`?

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(4)`?

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(4)`?

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

b(3) = 22

b(2) = 12

b(1) = 7

b(0) = 2

b(-1) = 2

Recursion Practice

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

- Simple recursion (also called tail recursion)

Finding the Maximum in an Array

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

Clicker 6

- When writing recursive methods what should be done first?

A. Determine recursive case

B. Determine recursive step

C. Make a recursive call

D. Determine base case(s)

E. Determine the 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

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