CS 112 – MiraCosta College

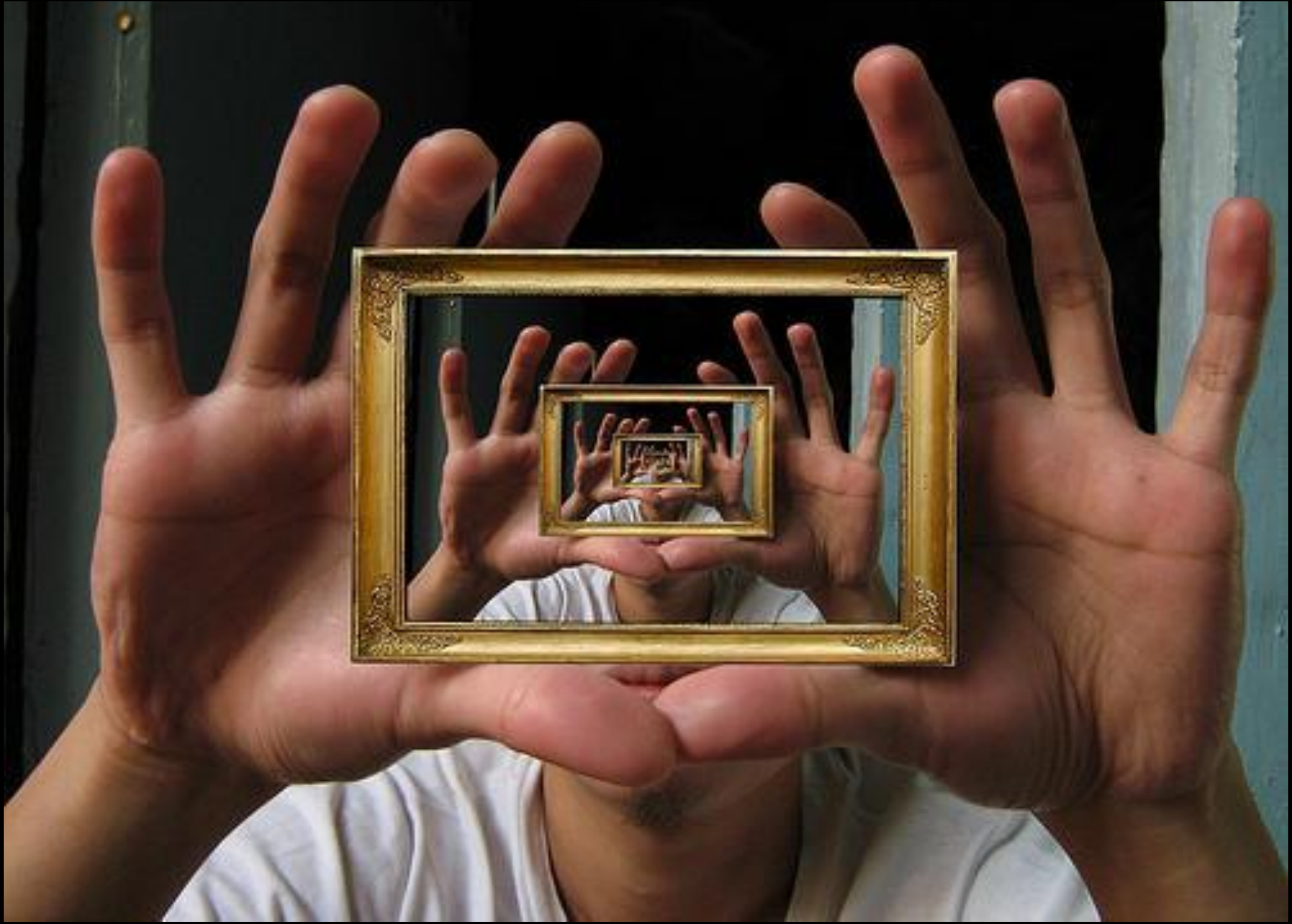
Introduction to Computer Science II
Java

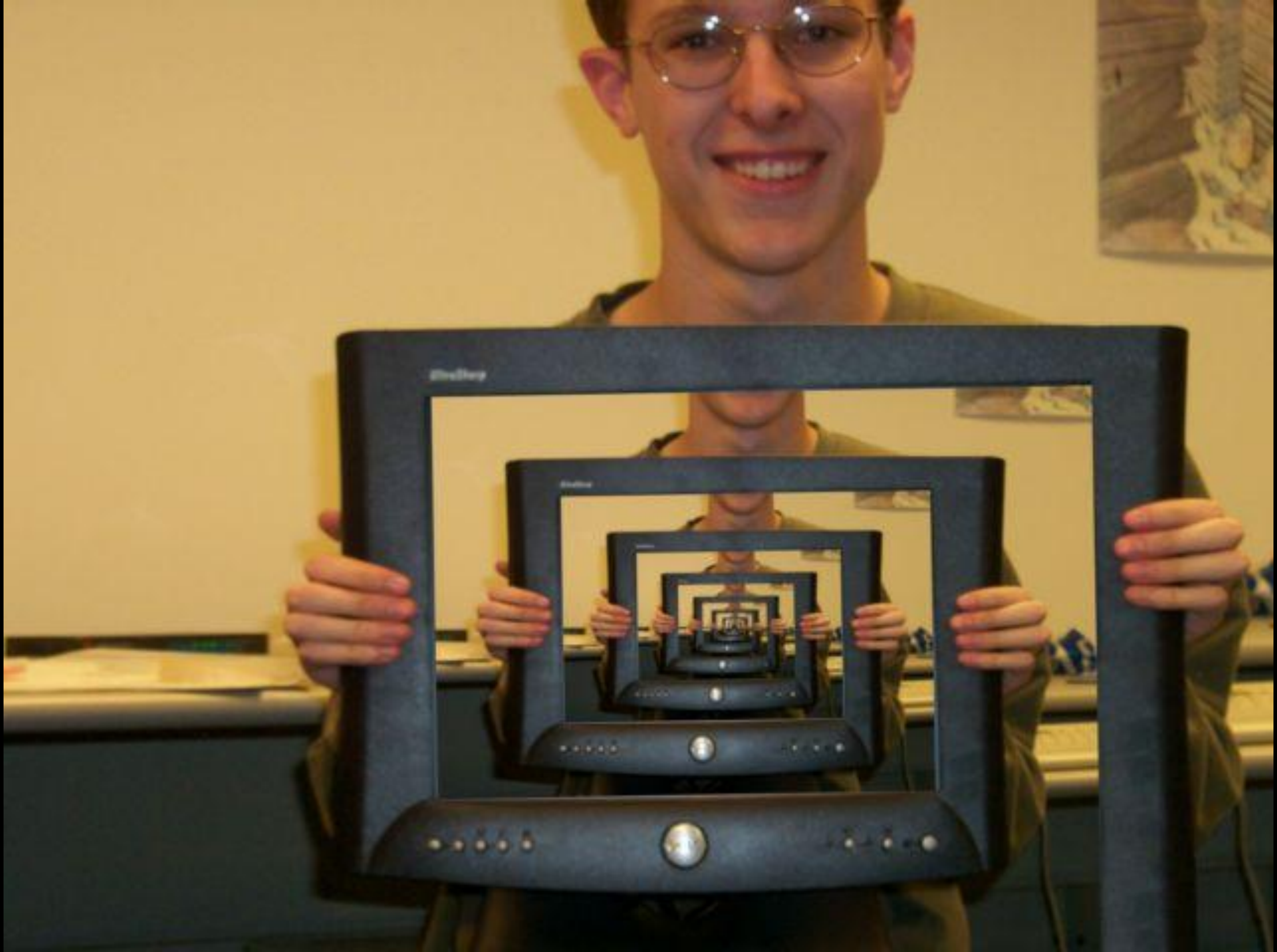
Module 7 - Recursion

Chris Merrill
Computer Science Dept
cmerrill@miracosta.edu

Agenda

- Review
  - Homework, Module 5
- Recursion
- Recursive methods that return a value
- Recursion vs. iteration
- Binary searches
- Lab 7 (Recursion)





TO-DO LIST

1. Make a to-do list

Interesting…

- *coral n.* – 2: a piece of coral…

  – Merriam Webster On-line Dictionary

- The TTP Project

  – Dilbert

- All short statements about programming techniques are false

  – Anonymous

Recursion

- A *recursive* method is a method that includes a call to itself
- Recursion is based on the general problem solving technique of breaking down a task into subtasks
  - In particular, recursion can be used whenever one subtask is a “smaller” or “simpler” version of the original task

Recursive Methods

Unlike the two photos and the clever witticisms we just saw, recursive methods have one property that makes them useful in computer science:

They must always come to an end!

(Algorithms must have a *finite* number of steps.)

Counting Numbers

Let's begin with a recursive method that prints consecutive numbers down the screen, one per line:

- It will takes a nonnegative `int` argument as the starting point
- It will print consecutive numbers descending from the argument down to 1
- It will be static so that `main` can invoke it, though that's not a requirement for being recursive

Counting Numbers

We can break down the overall task into the following two subtasks, based on the argument **`n`**

- Case 1: If **`n = 1`**, then print t **`n`** on the screen and return
- Case 2: If **`n > 1`**, then print **`n`** on the screen, invoke the method again using **`n - 1`** as the argument, and return

Counting Numbers

Given the argument 4, it will fall into Case 2. We can break it down further into smaller processes:

1. Print the number “4”

2. Perform the same overall process for the number (4 – 1) or 3

3. Return

Counting Numbers

Now using the argument 3, it again will fall into Case 2. We can break it down further into smaller processes:

1. Print the number “3”
2. Perform the same overall process for the number (3 – 1) or 2
3. Return

Counting Numbers

Following the same pattern, we use the argument 2, again falling into Case 2. We can break it down further into smaller processes:

1. Print the number “2”
2. Perform the same overall process for the number (2 – 1) or 1
3. Return

Counting Numbers

Finally we’ve reached the stopping or base case. This case doesn’t call itself, but just:

1. Print the number “1”
2. Return

Counting Numbers

The decomposition of tasks into subtasks can be used to derive the method definition:

- Note that the “Case 1” is a reduced version of the “Case 2”
- The “Case 2” has just one more step than the “Case 1”, which is a call to itself – recursion!
- Lastly, “Case 1” provides the way to stop the recursion.

Algorithm for Counting Numbers

Given argument **`n`**:

1. **`System.out.println(n);`**
2. if **`n`** is greater than 1, then invoke method again using **`n – 1`** as argument
3. **`return`**

Demonstration of Counting Numbers

Let’s look at the code, then determine how would we change this algorithm so that the numbers print in ascending order (from 1 $\rightarrow$ n instead of n $\rightarrow$ 1)?

(CountingNumbers)

Vertical Numbers

The textbook uses another example which prints the individual digits of a multi-digit number down the screen. For example, if the method takes “538” as a argument, then using recursion, it will print:

```
5
3
8
```

Vertical Numbers

We can break down this task into the following two subtasks.

- Simple case: If `n < 10`, then write the number `n` to the screen
- Recursive Case: If `n >= 10`, then do two subtasks:
    1. Print all of the digits except the last digit
    2. Print the last digit

Vertical Numbers

Given the argument “1234”, then using the Recursive Case, the output of its first part would be:

```
1
2
3
```

and output of its second part would be:

```
4
```

Vertical Numbers

The first part of the prior slide, using an argument of "123" can itself be broken down into two parts. Its first part would print

```
1
2
```

and its second part would print

```
3
```

And so on…

Algorithm for Vertical Numbers

Given argument **n**:

```
if (n < 10) {
    System.out.println(n);
} else {
    invoke method with n / 10 as argument
   System.out.println(n % 10);
}
```

Algorithm for Vertical Numbers

Or another version (simpler?):

```
if (n >= 10) {
    invoke method with n / 10 as argument
}
System.out.println(n % 10);
```

Demonstration of Vertical Numbers

Show this algorithm in action…

(VerticalNumbers)

Triangles

- Problem: to compute the area of a triangle of squares of width *n*

  [ ] n = 1
  [ ] [ ] n = 2
  [ ] [ ] [ ] n = 3

- If we assume each square [ ] has an area of 1, then the area of a triangle:

  of width 1 is 1,
  of width 2 is 2 + 1 = 3,
  of width 3 = 3 + 2 + 1 = 6, etc.

An “Iterative” Solution

For any triangle with a width > 1, we can add code to calculate the area of that triangle using a `for` loop:

```
int area = 0 ;
for (int i = 1 ; i <= aWidth ; i++) {
   area += i ;
}
```

Triangles

To solve this problem recursively, we'll use a somewhat different approach:

- In the previous recursion examples, we simply invoked the same method over and over
- In this problem, we'll create a new **`Triangle`** object at each level of the recursion, and invoke a method in that object which uses a property of the **`Triangle`** which converges to a stopping point as we go to deeper and deeper levels.

The “Base” Case

For now, assume that we are only going to deal with triangle widths that are more than 1.

Then we can handle the easiest case first:

```
if (aWidth == 1) {
    return 1 ;
}
```

- Notice that we can’t reduce the problem any further.   This is called the “base” or “stopping” case

More Complex Triangles

Notice that for any triangle of width n, we can calculate its area as *n + (the area of a triangle of width n – 1)*

If n = 4, then the area of that triangle is

4 + (the width of a triangle of width 3)

and if n = 3, then the area of that triangle is:

3 + (the width of a triangle of width 2)

and if n = 2, then the area of that triangle is:

2 + (the width of a triangle of width 1)

We know this is 1!

Triangles

An outline of a `Triangle` class:

```
public class Triangle {
    private int width;

    public Triangle(int aWidth) {
        width = aWidth;
    }
    public int getArea(int aWidth) {
        . . .
    }
}
```

This is what we will be constructing

A “Recursive” Solution

Revisiting the algorithm which uses the area of smaller triangles to determine the area of larger triangles:

The base case: (n == 1)

the area is 1

The general or “recursive” case: (n > 1)

the area of a triangle with width n =

n + (the area of a triangle of width n – 1)

Using Recursion

Recursive code for `getArea():`

```
public int getArea() {
    if (aWidth == 1) {
       return 1;
   }
   return aWidth + getArea(aWidth - 1) ;
}
```

Recursive Solution using The **`Triangle`** class

Creating a new **`Triangle`** object

at each level of recursion

(Triangles)

Mini-Lab #1

What does the following recursive method do?

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

What is `triple(4)`?

Mini-Lab Tracing

Let’s walk through `triple(4)`

```
triple(4) = 3 + triple(3)
triple(3) = 3 + triple(2)
triple(2) = 3 + triple(1)
triple(1) = 3 + triple(0)
triple(0) = 0
```

Now just walk up the equations and see that triple(4) is 12

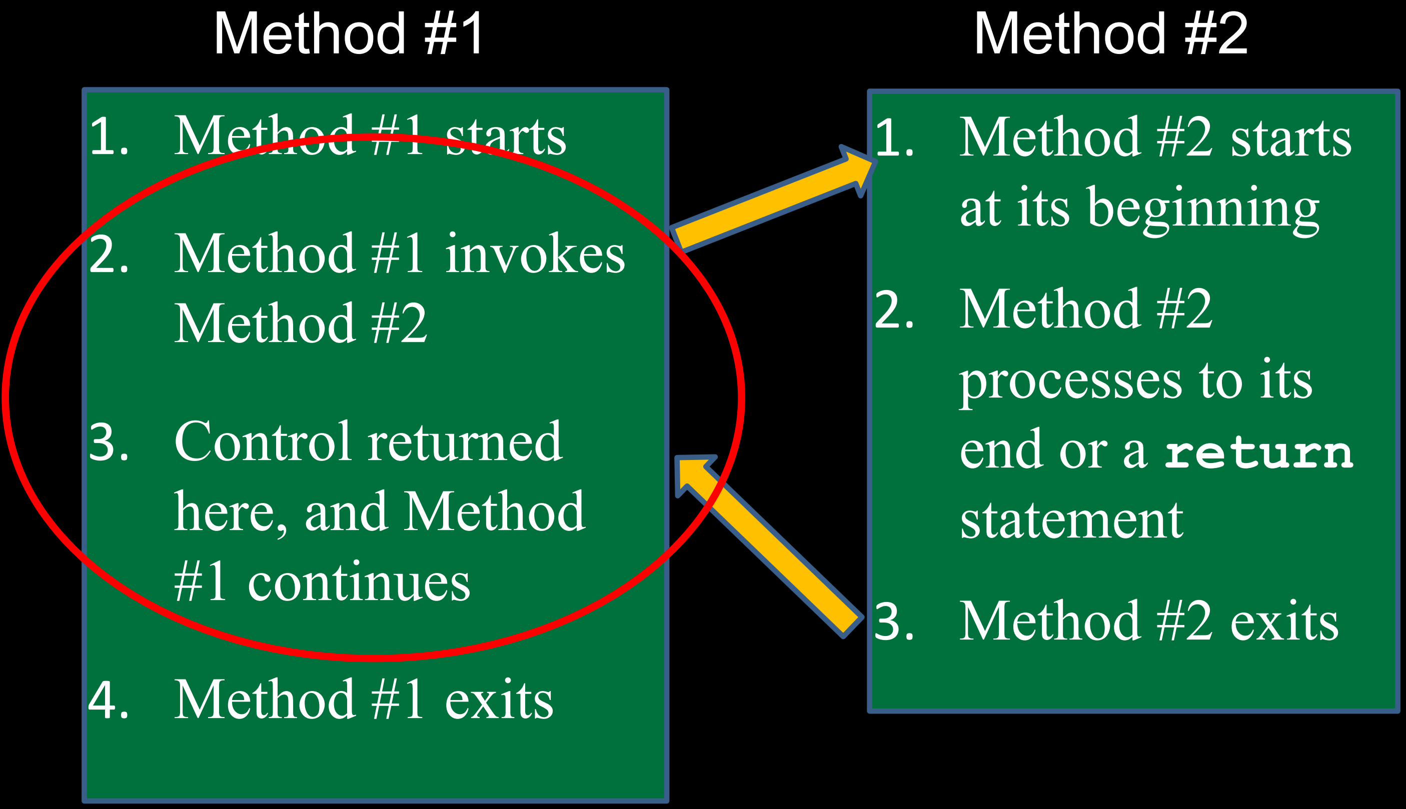
What Happens When Invoking *Any* Method

When one method (say #1) invokes another method (#2)

- Method #1 temporarily halts execution
- The body of Method #2 begins executing at the beginning of its body
- Method #2 eventually halts executing
- Control is returned to the Method #1 and execution is takes up immediately after the statement or expression which invoked Method #2.

Tracing a Call To Any Method

What happens when Method #1 invokes Method #2



Method #1

1. Method #1 starts
2. Method #1 invokes Method #2
3. Control returned here, and Method #1 continues
4. Method #1 exits

Method #2

1. Method #2 starts at its beginning
2. Method #2 processes to its end or a **`return`** statement
3. Method #2 exits

A Method Calling Itself (Recursion)

The computer keeps track of recursive calls as follows:

- When a method is called, the computer plugs in the arguments for the parameter(s), and starts executing the code
- If it encounters a recursive call, it temporarily stops its computation and calls itself with a different argument
- When the recursive call is completed, the computer returns to finish the outer computation

Tracing a Recursive Call

Recursive methods are processed in the same way as any method call

```
writeVertical(123);
```

- When this call is executed, the argument `123` is substituted for the parameter, and the body of the method is executed
- Since `123 >= 10`, the method is invoked a 2nd time with the argument `12` (i.e., `123 / 10`)

Tracing a Recursive Call (page 2)

- The method is executed again with `12` as the parameter
- Since `12 >= 10`, the method is invoked a 3rd time with 1 as the argument (using `12 / 10`)
- The method is entered one final time with `1` as the parameter
- The if statement is ignored, and `1` is printed

Tracing a Recursive Call (page 3)

- The method exits, returning to the 2nd execution
- The method prints `2 (12 % 10)`
- The method exits, returning to the 1st execution
- The method prints `3 (123 % 10)`
- The method exits, returning to `main`

Tracing a Recursive Call to `writeVertical`

1. Invoked with argument 123
2. Recursive call with argument (123 / 10) or 12
3. Control returned
4. Method prints (123 % 10) or “3”
5. Method exits

1. Invoked with argument 12
2. Recursive call with argument (12 / 10) or 1
3. Control returned
4. Method prints (12 % 10) or “2”
5. Method exits

1. Invoked with argument 1
2. No recursive call needed
3. Method prints (1 % 10) or “1”
4. Method exits

A Closer Look at Recursion

- When the computer encounters a recursive call, it must temporarily suspend its execution of a method
  - It does this because *it must know the result of the recursive call before it can proceed*
  - It saves all the information it needs to continue the computation later on, when it returns from the recursive call
- Ultimately, this entire process terminates when one of the recursive calls does not depend upon recursion to return

Stacks

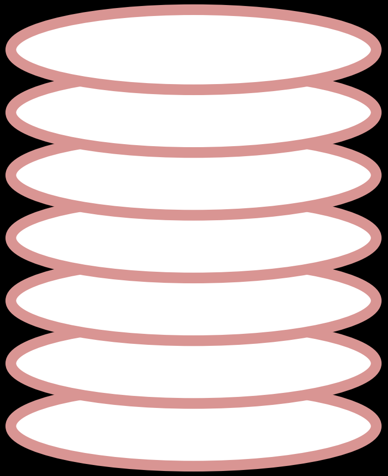
Computer systems often use two types of data structures called *stacks* and *queues*.

The most common analogy for a stack is a stack of plates on a spring at a buffet line

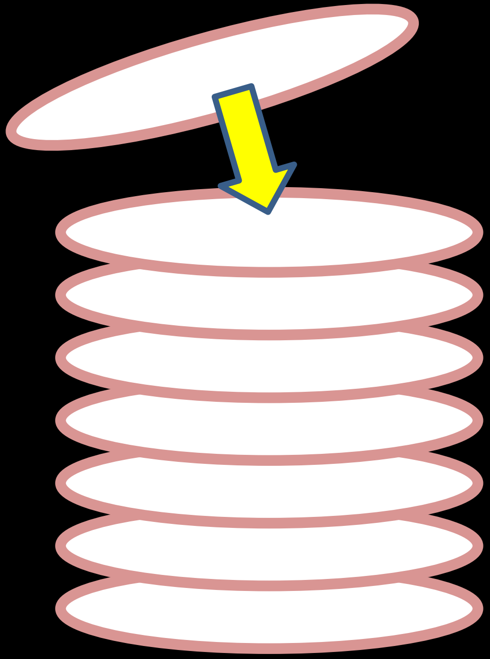
- As plates are washed and dried, they are placed on the *top* of the stack
- When hungry customers takes a clean plate, they take plates from the *top* of the stack

Stack Example

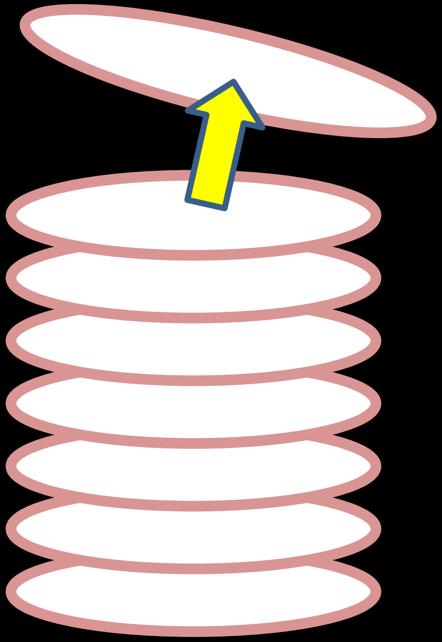
Stack of Plates



Clean Plates Goes on *Top*



Hungry Customers Take New Plates From the *Top*



Queues

Queues work from opposite sides. A popular example is people waiting in line

- As people enter the queue, they go to the *back* of the line
- When people are serviced, they are taken from the *front* of the line

In fact, “lines” are called “queues” in the UK

Queue Example

Exit Here

Enter Here

Stacks and Queues

Since the last (or most recent) plate on a stack is the first one out, stacks are often referred to as “last-in-first-out” structures, or

LIFO

Since the first (or oldest) person in a queue is the first one out, queues are often called “first-in-first-out” structures, or

FIFO

Stacks for Recursion

To keep track of recursion, computer systems use *stacks*

- In this case, a better analogy for a “stack” memory structure is a stack of *paper*
- Unlike the analogy in the textbook, there is NOT an inexhaustible supply of blank sheets of paper, as we’ve seen that the stack eventually runs out of resources.

Stacks for Recursion

We can use the paper to record information about the state of each iteration

- Information such as the value of local variables is placed on the stack by recording it on a sheet and placing it on top of the stack (becoming the new top of the stack)
- The system starts recording information about a new iteration on another sheet, placing it on top of the stack, and so on

Stacks for Recursion

- To get information out of the stack, the top paper can be read and local variables restored, *but only the top paper*
- To get *more* information, the top paper can be thrown away, and then the new top paper can be read, and so on
- Can you think of an example of a stack that we use in Windows (how about Windows Explorer?)

Stacks for Recursion

To keep track of recursion, whenever a method is called, a new "sheet of paper" is taken

- The method definition is copied onto this sheet, and the arguments are plugged in for the method parameters
- The computer starts to execute the method body
- When it encounters a recursive call, it stops the computation in order to make the recursive call

Stacks for Recursion

- It records the current values of local variables and the statement location on the *sheet of paper*, and places it on the stack
- A new *sheet of paper* is used for the recursive call
  - The computer writes a second copy of the method, plugs in the arguments, and starts to execute its body
  - When this copy gets to a recursive call, its information is saved on the stack also, and a new *sheet of paper* is used for the new recursive call

Stacks for Recursion

- This process continues until some recursive call to the method completes without any more recursive calls
  - Its *sheet of paper* is then discarded
- Then the computer goes to the top *sheet of paper* on the stack, which contains the partially-completed computation waiting for the recursive computation just ended
  - Values for local variables are restored, and execution starts where it left off

Stacks for Recursion

- Depending on how many recursive calls are made, and how the method definition is written, the stack may grow and shrink in any fashion
- Using the stack of paper analogy to computer systems
  - The content of one of the *sheets of paper* is called a *stack frame* or *activation record*
  - The stack frames don't actually contain a complete copy of the method definition – they don’t need to record byte code, constants, or static variables

Stack Overflow

There is always some limit to the size of the stack

- If there is a long chain in which a method makes a call to itself, and that call makes another recursive call, and so forth, there will be many suspended computations placed on the stack
- If there are too many, then the stack will attempt to grow beyond its limit, resulting in an error condition known as a *stack overflow*

General Form of Recursive Method Definitions

The general outline of a successful recursive method definition is as follows:

- Any number of cases (including 0) that include recursive calls to the method being invoked
  - These recursive calls should solve smaller and/or simpler versions of the task performed by the current iteration of the method
- One or more cases that include no recursive calls: these named *base* or *stopping cases*

Infinite Recursion

In both **`countingNumbers`** and **`verticalNumbers`**, examples, the series of recursive calls eventually reaches a call of the method that does not involve recursion (a *stopping* or *base* case)

- If every recursive call produces another recursive call, then a call to that method would, in theory, run forever
- This is called *infinite recursion*

Infinite Recursion

Back in our first class, I defined an “algorithm” as having a finite number of steps. (Programs that run forever are *not* algorithms.)

- In the particular case of *infinite recursion,* the computer will simply run out of resources, and the program will terminate abnormally

Infinite Recursion

Here's an alternative version of **`writeVertical`** which doesn't have a base case!

```
public static void writeVertical(int n) {
    writeVertical(n / 10);
    System.out.println(n % 10);
}
```

No "stopping" case

Note that, just like infinite loops, this method compiles just fine, because Java's compiler cannot determine that it will run indefinitely.

Infinite Recursion

- Invoking **`writeVertical(123)`** causes that execution to stop and make a recursive call with argument 12/10 $\rightarrow$ **`writeVertical(12)`**
- Invoking **`writeVertical(12)`** causes that execution to stop and make the call to **`writeVertical(1)`**
- Invoking **`writeVertical(1)`** causes that execution to stop and make the call to **`writeVertical(0)`**
- And we’re stuck in an endless loop of calls invoking **`writeVertical(0)`**

Infinite Recursion Demo

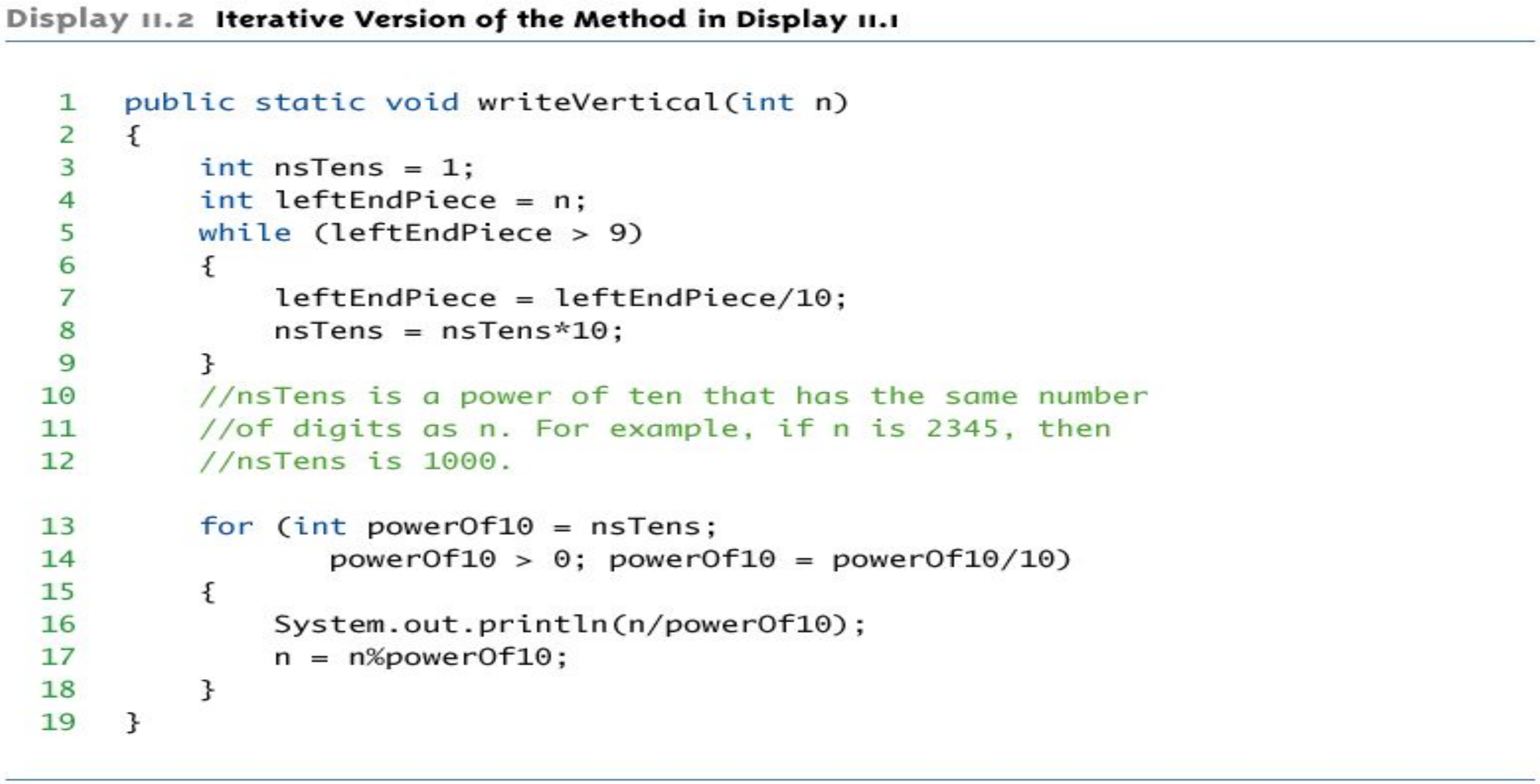
What happens?

(InfiniteRecursion)

Recursion Versus Iteration

- Recursion is not absolutely necessary
  - Any task that can be done using recursion can also be done in a non-recursive manner
  - A non-recursive version of a method is called an *iterative version*
- An iteratively written method will typically use loops of some sort in place of recursion
- A recursively written method can be simpler, but will usually run slower and use more storage than an equivalent iterative version

Iterative version of **`writeVertical`**



Display 11.2 Iterative Version of the Method in Display 11.1

```java
public static void writeVertical(int n)
{
    int nsTens = 1;
    int leftEndPiece = n;
    while (leftEndPiece > 9)
    {
        leftEndPiece = leftEndPiece/10;
        nsTens = nsTens*10;
    }
    //nsTens is a power of ten that has the same number
    //of digits as n. For example, if n is 2345, then
    //nsTens is 1000.

    for (int powerOf10 = nsTens;
           powerOf10 > 0; powerOf10 = powerOf10/10)
    {
        System.out.println(n/powerOf10);
        n = n%powerOf10;
    }
}
```

Recursion Using The Bullseye Swing Program Project from Lab 1

Draw smaller and smaller circles

around a concentric point

(Bullseye)

Recursive Methods that Return a Value

Recursion is not limited to `void` methods

- A recursive method can return a value of any type
- The stopping or base case returns an initial value for a calculation
- As each iteration completes, its returned value is then used in the calculation of the return value for the next iteration "up the chain"

Another Powers Method Example

The method **`pow`** from the **`Math`** class computes powers

- It takes two arguments of type **`double`** and returns a value of type **`double`**
- Our recursive method **`power`** takes two arguments of type **`int`** and returns a value of type **`int`**
  - The definition of **`power`** is based on the formula:

x^{n} is equal to $x^{n-1} * x$

Another Powers Method Example

- In Java, the value returned by **`power(x, n)`** when **`n > 0`** should be the same as

```
power(x, n-1) * x
```

- When **`n = 0`**, then **`power(x, n)`** should be 1
  - This is the stopping case

The Recursive Method **power**

```
public static int power(int x, int n) {
    if (n == 0) {
        return 1 ;
    }
    return (power(x, n - 1) * x ;
}
```

The Recursive Method **power**

Here is another version of the `power` method

```
public static int power(int x, int n) {
    return (n > 0 ? power(x, n - 1) * x : 1) ;
}
```

Factorials

A second common example of recursion in computer science involves the calculation of factorials.

- Notation is “n!” in math or factorial(n) for computer code.
- The factorial of a nonnegative number is defined as:
  - If n = 0, then factorial(n) = 1.
  - If n > 0, then factorial(n) = 1 * 2 * 3 *...* n or the product of all positive integers from 1 up to n.
  - If n > 0, we can represent the code as

```
factorial(n) = n * factorial(n - 1)
```

Factorials

Here are the factorial values for 0 through 4:

```
0! = 1
1! = 1
2! = 2 * 1 = 2
3! = 3 * 2 * 1 = 6
4! = 4 * 3 * 2 * 1 = 24
```

The recursive formula: `n! = n * (n - 1)!`

The stopping case: `0! = 1`

Factorials

Factorials can be used to determine the possible number of ways that “n” items can be listed (called *permutations*):

- The 1st position can be filled “n” ways
- The 2nd position can be filled “n – 1” ways
- The 3rd position can be filled “n – 2” ways
- . . .
- The nth (or last) position can be filled 1 way

For example, three numbers can be listed 6 ways (123, 132, 213, 231, 312, and 321)

Factorials

- Factorials are also useful for calculating functions:

$$\sin x = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \ldots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \ldots$$

$$e^x = \frac{x^0}{1!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \ldots$$

Factorial Method

Here’s one possible listing for a method that takes an `int` as an argument and returns its factorial:

```
public int factorial(int n) {
    if (n <= 0) {
        return 1 ;
    }
    return n * factorial(n - 1) ;
}
```

Factorial Method

Here's another solution:

```
public int factorial(int n) {
   return (n > 0 ? n * factorial(n - 1) : 1) ;
}
```

Thinking Recursively

If a problem lends itself to recursion, it is more important to think of it in recursive terms, rather than concentrating on the stack and the suspended computations

```
power(x, n) ≡ power(x, n-1) * x
```

- In the case of methods that return a value, there are three properties that must be satisfied, as follows:

Thinking Recursively

1. There is no infinite recursion – every chain of recursive calls must reach a stopping case
2. Every stopping case returns the correct value for that case
3. For the cases that involve recursion: *if* all recursive calls return the correct value, *then* the final value returned by the method is the correct value

These properties use a logic technique also known as *mathematical induction*

Types of Recursion: Tail Recursion

When the recursive method does nothing after the recursive call except return the value, then the method is called *tail recursive*

- Tail recursive methods can easily be converted into an equivalent iterative algorithm
- Java may do this automatically for greater efficiency
- A similar effect can be achieved if the compiler re-uses the stack frame for successive recursive calls

Iterative version of `verticalWrite`

Digits from right-to-left (the “tail recursion” version)...

```
int number = 531 ;
while (number > 0) {
    System.out.println(number % 10) ;
    number /= 10 ;
}
```

Without converting the number to a `String`, how would you do it from left-to-right?

Mutual or Indirect Recursion

When one method calls a second method, and the second method calls the first, this is called “mutual” or “indirect” recursion. For example, to determine if a number is even:

```
1 public static boolean isOdd(int n) {
     return (n == 0 ? false : isEven(n - 1)) ;
  }
2 public static boolean isEven(int n) {
      return (n == 0 ? true : isOdd(n - 1)) ;
  }
```

Single and Multiple Recursion

When a recursive method invokes itself only once, it is called “single” recursion:

- The factorial and power problems are examples of single recursion

When a recursive method invokes itself more than once, it is called “multiple” recursion:

- The Fibonacci sequence is an example of this

Fibonacci Numbers

Another interesting series numbers is known as the "Fibonacci sequence". It is characterized by computing each number in the sequence as the sum of the prior two numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…

Fibonacci Numbers

These numbers are commonly found in nature, such as:

- petals of a flower
- the seeds of fruits
- rows of seeds on a sun flower
- lobes of pinecones
- spirals on a shell

Fibonacci numbers also are used heavily in predicting the price of stocks, analyzing trend changes, and setting high and low targets

Fibonacci Method

Here's a possible listing for a method that takes an `int` as an argument and returns its Fibonacci number:

```
public int fibonacci(int n) {
    if (n <= 0)
        return 0 ;
    if (n == 1)
        return 1 ;
    return (fibonacci(n - 1) + fibonacci(n - 2);
}
```

Fibonacci Demo

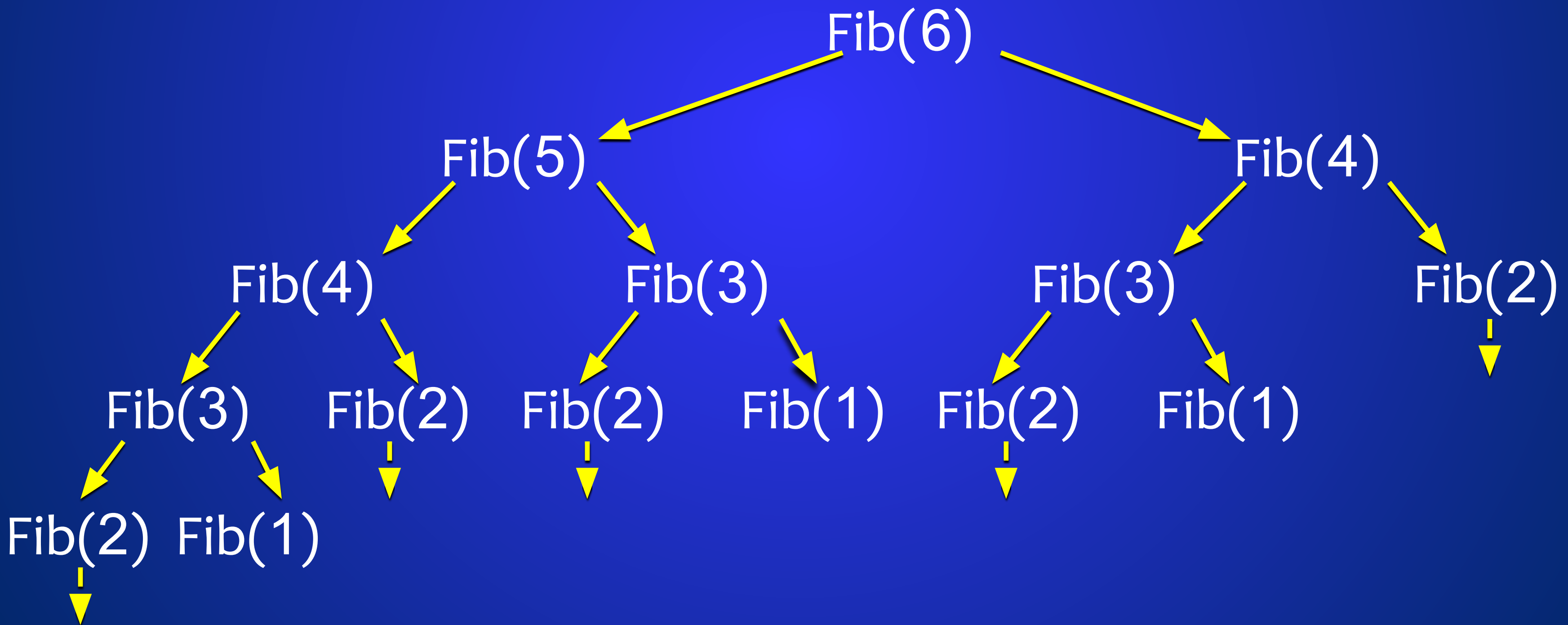
Run the Fibonacci sequences for numbers 1 – 50.
Use both recursive and iterative solutions.

What goes wrong?

(Fibonacci)

Inefficiencies in Recursion

When we try to find the Fibonacci number for “n” using the recursive solution, we actually create a tree of calls:



The Call Tree For Fibonacci Number 6

Fib(6) is called 1 time

Fib(5) is also called 1 time

Fib(4) is called 2 times

Fib(3) is called 3 times

Fib(2) is called 5 times

Fib(1) is called 8 times

Total calls: <u>20</u>

Big O is $O(n^2)$

Efficiencies in Recursion – Binary Search

- Binary search uses a recursive method to search an *sorted* array to find a specified value

`a[0] ≤a [1] ≤a [2] ≤. . . ≤ a[finalIndex]`

- If the value is found, its index is returned
- If the value is not found, -1 is returned
- In a binary search, each iteration of the recursive method reduces the search space by about a half

Binary Search

- An algorithm to solve this task looks at the middle of the array or array segment first
- If the value looked for is smaller than the value in the middle of the array
  - Then the second half of the array or array segment can be ignored
  - This strategy is then applied to the first half of the array or array segment

Binary Search

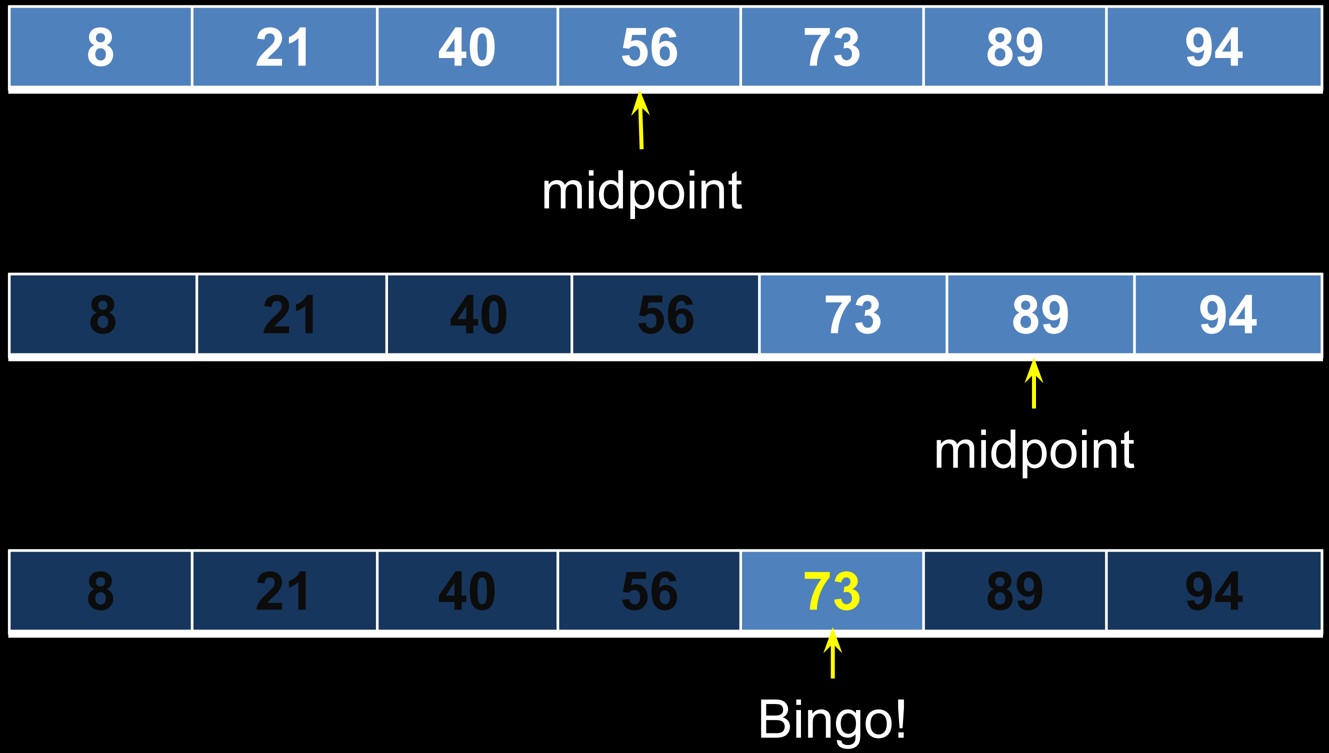
- If the value looked for is larger than the value in the middle of the array or array segment, then the first half of the array or array segment can be ignored
  - This strategy is then applied to the second half of the array or array segment
- If the value looked for is at the middle of the array or array segment, then it has been found
- If the entire array (or array segment) has been searched in this way without finding the value, then it is not in the array

Pseudocode for Binary Search

```
if (first > last)
    return -1                // not found!
midpoint = (first + last) / 2
if (key equals array[midpoint])
    return midpoint     // found!
~~otherwise~~ if (key < array[midpoint])
    return result of searching array[first] to array[mid – 1]
~~otherwise~~
    return result of searching array[midpoint + 1] to array[last]
```

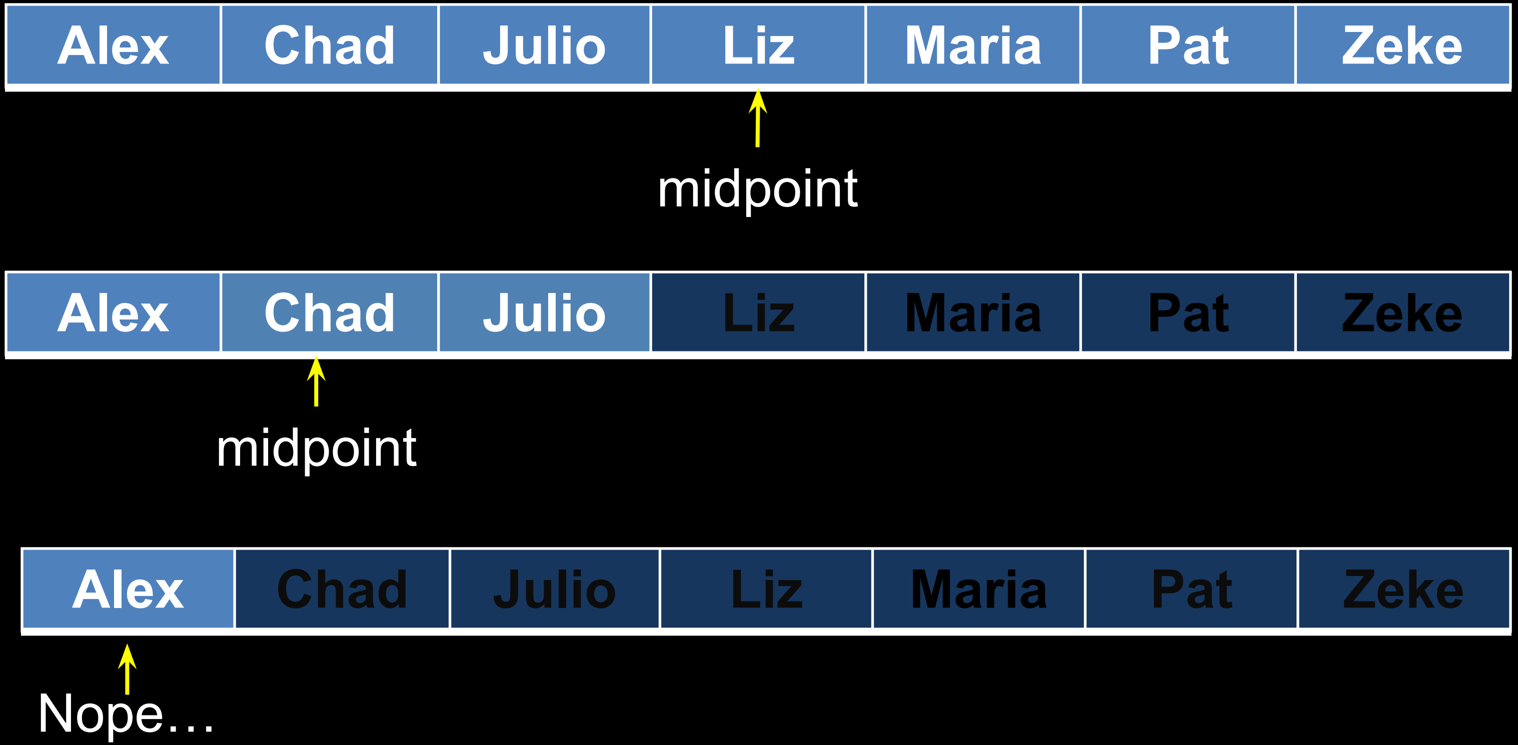
Example of A Binary Search

Search for “73”



Another Example

Search for “Bob”



Demo of Binary Search

Example of a binary search method

in a long list of sorted numbers

(BinarySearch)

Efficiency of Binary Search

The binary search algorithm is extremely fast compared to an algorithm that tries all array elements in order

- About half the array is eliminated from consideration on the very first iteration
- On the second iteration, another quarter of the array is eliminated, leaving only a quarter of the array
- On the third iteration, another eighth, leaving an eighth, and so forth

Efficiency of Binary Search

As we’ve seen, using an array with 1,000 elements, the binary search will only need to compare about 10 array elements to the key value (and a million elements about 20 comparisons).

- Binary searches will divide the size of the 1,000 element array to be searched into 500, 250, 125, 63, 32, 16, 8, 4, 2, and finally 1 elements
- The binary search algorithm has a worst-case running time that is logarithmic: $O(\log_2 n)$

Efficiency of Binary Search

A serial search algorithm is linear: O(*n*)

- An array of 1,000 elements will need to make, on average, about 500 comparisons before finding the desired element (or 1,000 if the element isn't found.)
- However, in this case, the array does *not* have to be sorted!
- We can convert the search program to an iterative version which will run more efficiently

Mini-Lab #2

Write a recursive method which calculates the square root of 2

- Start with first = 1 and last = 2
- What is the midpoint?
- Where does midpoint^2 fall?
- Stop when midpoint^2 falls within 0.01 if the right answer
- If you have time, can you count how many levels it took to get to your answer?

Convert **`search`** to An Iterative Algorithm

Can we convert the **`search`** method

to an iterative version which will run

more efficiently (?)

(IterativeSearch)

Backtracking

Backtracking is a problem solving technique that builds up partial solutions that get increasingly closer to the goal.

- If a partial solution cannot be completed, one abandons it and returns to examining the other candidates.

- If a partial solution can be completed, then continue examining other solutions which use this solution as a beginning.

Backtracking

Characteristic properties needed to use backtracking for a problem.

1. A procedure to examine a partial solution and determine whether to:
   - Accept it as an actual solution.
   - Abandon it (either because it violates some rules or because it is clear that it can never lead to a valid solution).
   - Continue extending it.

Backtracking

2. A procedure to extend a partial solution, generating one or more solutions that come closer to the goal.

In a backtracking algorithm, one explores all paths towards a solution. When one path is a dead end, one needs to backtrack and try another choice.

Backtracking Algorithm

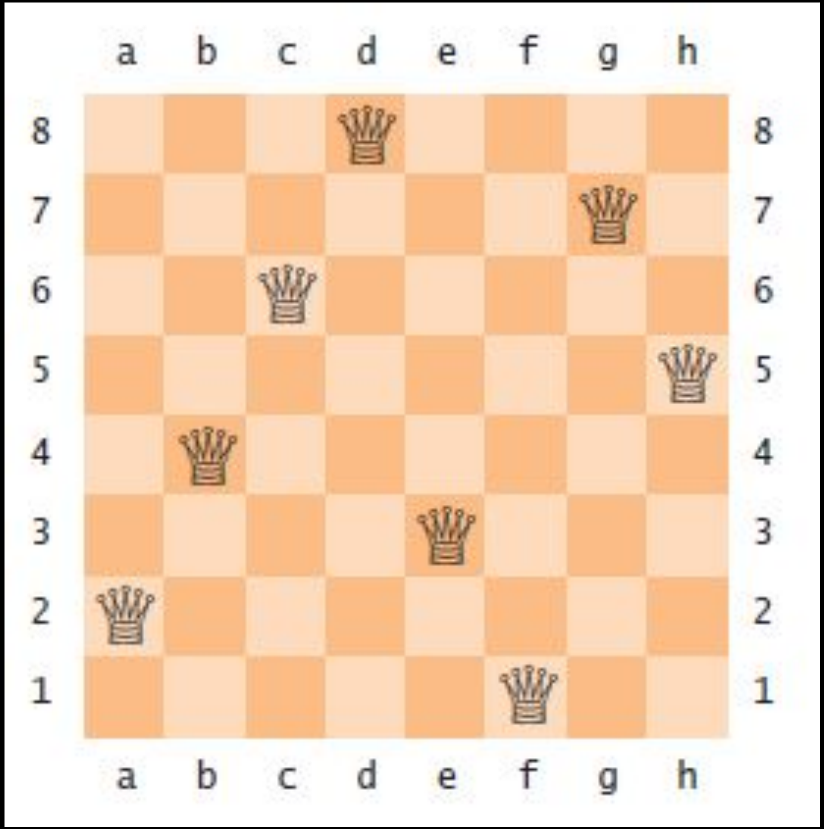
Backtracking can be performed using the following recursive algorithm.

```
Solve (partialsolution)
    Examine (partialsolution)
    If partialsolution solves the problem
        Add partialsolution to the list of solutions
    Else if ok to continue
        For each p in extend(partialsolution)
            Solve (p)
```

The "Eight Queens" Problem

Place 8 queens on a chessboard so that no queen threatens another queen.

(Queens can move vertically, horizontally, or along any diagonal).



Backtracking Algorithm

To examine a partial solution:

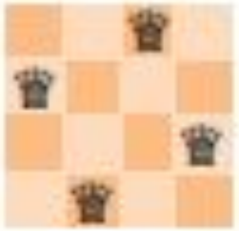
- If two queens attack each other, then reject this solution
- If eight queens are played, then accept this solution
- Otherwise, continue

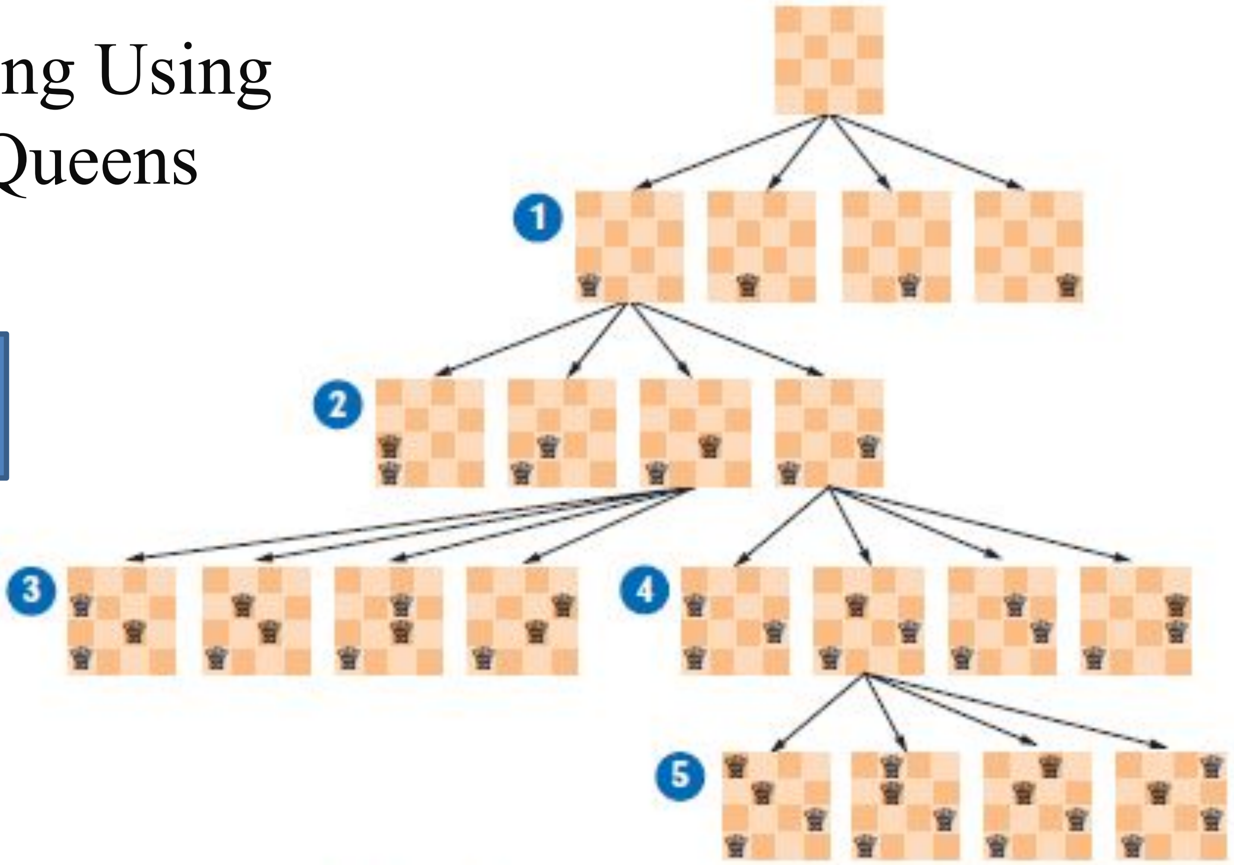
To extend a partial solution:

- Find the next empty square and place a queen on it
  - For efficiency, place first queen in row 1, the next in row 2, and so on

Backtracking Using Only 4 Queens

The Solution





Solution Using a Three Classes

`Queen` – describes the location (row, column) of a Queen placed on the board with a method to determine if another Queen is attacking **`this Queen`**

`PartialSolution` – contains an array of **`Queen`** objects, with a method to see if any queen is attacking another **`Queen`**, and a method to extend the board.

`EightQueens` – contains **`main`**, which looks at all possible solutions, maintains a list of actual solutions, and prints all actual solutions when done.

Eight Queens Solution

Some rather nice OOP techniques

(EightQueens)

A Child’s Game Can Demonstrate Real-World Recursive Thinking

The Tower of Hanoi is a great example of developing smaller, like problems to solve a much larger problem

(Homework Project 3)

Homework

- Complete old homework and labs.
- Complete the Lab 7 – Recursion
- Homework Assignment, Module 7, projects 1, 2, and 3
- Turn in everything (with beautiful Introductory Comments and documented code) at the ***beginning*** of next class

Group Lab 7

- Determine whether a **`String`** is a palindrome
- Determine the greatest common denominator between two numbers
- Write a decimal-to-binary conversion program
- Write a binary-to-decimal conversion program (EC)