Chapter 11

Recursion

- Basics of Recursion
- Programming with Recursion

Overview

Recursion: a definition in terms of itself.

Recursion in algorithms:

- Recursion is a natural approach to some problems
  - » it sounds circular, but in practice it is not
- An *algorithm* is a step-by-step set of rules to solve a problem
  - » it must eventually terminate with a solution
- A *recursive algorithm* uses itself to solve one or more subcases

Recursion in Java:

- Recursive methods implement recursive algorithms
- A *recursive method* is one whose definition includes a call to itself
  - » a method definition with an invocation of the very method used to define it

Recursive Methods Must Eventually Terminate

A recursive method must have at least one base, or stopping, case.

- A base case does not execute a recursive call
  - » it stops the recursion

- Each successive call to itself must be a "smaller version of itself" so that a base case is eventually reached
  - » an argument must be made smaller each call so that eventually the base case executes and stops the recursion

Example: a Recursive Algorithm

One way to search a phone book (which is an alphabetically ordered list) for a name is with the following recursive algorithm:

Search:

middle page = (first page + last page)/2
Open the phone book to middle page;
If (name is on middle page)
  then done; //this is the **base case**
else if (name is alphabetically before middle page)
  last page = middle page //redefine search area to front half
  Search //recursive call with **reduced number of pages**
else //name must be after middle page
  first page = middle page //redefine search area to back half
  Search //recursive call with **reduced number of pages**

Example: A Recursive Method

- `RecursionDemo` is a class to process an integer and print out its digits in words
  - » e.g. entering 123 would produce the output "one two three"
- `inWords` is the method that does the work of translating an integer to words

```
public static void inWords(int numeral)
{
   if (numeral < 10)
      System.out.print(digitWord(numeral) + " ");
   else //numeral has two or more digits
   {
      inWords(numeral/10);
      System.out.print(digitWord(numeral%10) + " ");
   }
}
```

Here is the recursive call: `inWords` definition calls itself

Example: A Base Case

- Each recursive call to `inWords` reduces the integer by one digit
  - » it drops out the least significant digit
- Eventually the argument to `inWords` has only one digit
  - » the if/else statement finally executes the base case and the algorithm terminates

Base case executes when only 1 digit is left

```
public static void inWords(int numeral)
{
   if (numeral < 10)
      System.out.print(digitWord(numeral) + " ");
   else //numeral has two or more digits
   {
      inWords(numeral/10);
      System.out.print(digitWord(numeral%10) + " ");
   }
}
```

What Happens with a Recursive Call

```
inWords(987)                    1
if (987 < 10)
    // print digit here
else //two or more digits left
{
   inWords(987/10);
   // print digit here
}
```

- Suppose that `inWords` is called from the `main` method of `RecursionDemo` with the argument `987`
- This box shows the code of `inWords` (slightly simplified) with the parameter numeral replaced by the argument `987`

What Happens with a Recursive Call

```
inWords(987)
if (987 < 10)
    // print digit here
else //two or more digits left
{
   inWords(987/10);
   // print digit here
}
```

2

```
inWords(98)
if (98 < 10)
    // print digit here
else //two or more digits left
{
   inWords(98/10);
   // print digit here
}
```

The argument is getting shorter and will eventually get to the base case.

Computation waits here until recursive call returns

- The `if` condition is false, so the `else` part of the code is executed
- In the `else` part there is a recursive call to `inWords`, with `987/10` or `98` as the argument

What Happens with a Recursive Call

```
inWords(987)
if (987 < 10)
    // print digit here
else //two or more digits left
{
   inWords(987/10);
   // print digit here
}
```

```
inWords(98)
if (98 < 10)
    // print digit here
else //two or more digits left
{
   inWords(98/10);
   // print digit here
}
```

3

```
inWords(9)
if (9 < 10)
    // print digit here
else //two or more digits left
{
   inWords(numeral/10);
   // print digit here
}
```

- In the recursive call, the `if` condition is false, so again the `else` part of the code is executed and another recursive call is made.

What Happens with a Recursive Call

```
inWords(987)
if (987 < 10)
    // print digit here
else //two or more digits left
{
   inWords(987/10);
   // print digit here
}
```

Output: nine

```
inWords(98)
if (98 < 10)
    // print digit here
else //two or more digits left
{
   inWords(98/10);
   // print 98 % 10
}
```

4

```
    // print nine
else //two or more digits left
{
   inWords(numeral/10);
   // print digit here
}
```

- This time the `if` condition is true and the base case is executed.
- The method prints `nine` and returns with no recursive call.

What Happens with a Recursive Call

```
inWords(987)
if (987 < 10)
    // print out digit here
else //two or more digits left
{
   inWords(987/10);
   // print digit here
}
```



```
if (98 < 10)
    // print out digit here
else //two or more digits left
{
   inWords(98/10);
   // print out 98 % 10 here
}
```

Output: `nine eight`

- The method executes the next statement after the recursive call, prints `eight` and then returns.

What Happens with a Recursive Call

```
inWords(987)
if (987 < 10)
    // print out digit here
else //two or more digits left
{
   inWords(987/10);
   // print 987 % 10
}
```

6

Output: `nine eight seven`

- Again the computation resumes where it left off and executes the next statement after the recursive method call.
- It prints seven and returns and computation resumes in the `main` method.

Remember: Key to Successful Recursion

Recursion will not work correctly unless you follow some specific guidelines:

- The heart of the method definition can be an `if-else` statement or some other branching statement.
- One or more of the branches should include a recursive invocation of the method.
  - » Recursive invocations should use "smaller" arguments or solve "smaller" versions of the task.
- One or more branches should include no recursive invocations. These are the stopping cases or base cases.

Warning: Infinite Recursion May Cause a Stack Overflow Error

- If a recursive method is called and a base case never executes, the method keeps calling itself
- The *stack* is a data structure that keeps track of recursive calls
- Every call puts data related to the call on the stack
  - » the data is taken off the stack only when the recursion stops
- So, if the recursion never stops, the stack eventually runs out of space
  - » and a stack overflow error occurs on most systems

Recursive Versus Iterative Methods

All recursive algorithms/methods can be rewritten without recursion.

- Methods rewritten without recursion typically have loops, so they are called *iterative* methods
- Iterative methods generally run faster and use less memory space
- So when should you use recursion?
  - » when efficiency is not important and it makes the code easier to understand

`numberOfZeros`–A Recursive Method that Returns a Value

- takes a single `int` argument and returns the number of zeros in the number
  - » example: `numberOfZeros(2030)` returns `2`

- uses the following fact:

  If `n` is two or more digits long, then the number of zero digits is (the number of zeros in `n` with the last digit removed) plus an additional 1 if that digit is zero.

  recursive

  Examples:
  - » number of zeros in `20030` is number of zeros in `2003` plus 1 for the last zero
  - » number of zeros in `20031` is number of zeros in `2003` plus 0 because last digit is not zero

numberOfZeros

```
public static int numberOfZeros(int n)
{
   if (n==0)
      return 1;
   else if (n < 10)  // and not 0
      return 0;  // 0 for no zeros
   else if (n%10 == 0)
      return (numberOfZeros(n/10) + 1);
   else  // n%10 != 0
      return (numberOfZeros(n/10));
}
```

Which is (are) the base case(s)? Why?

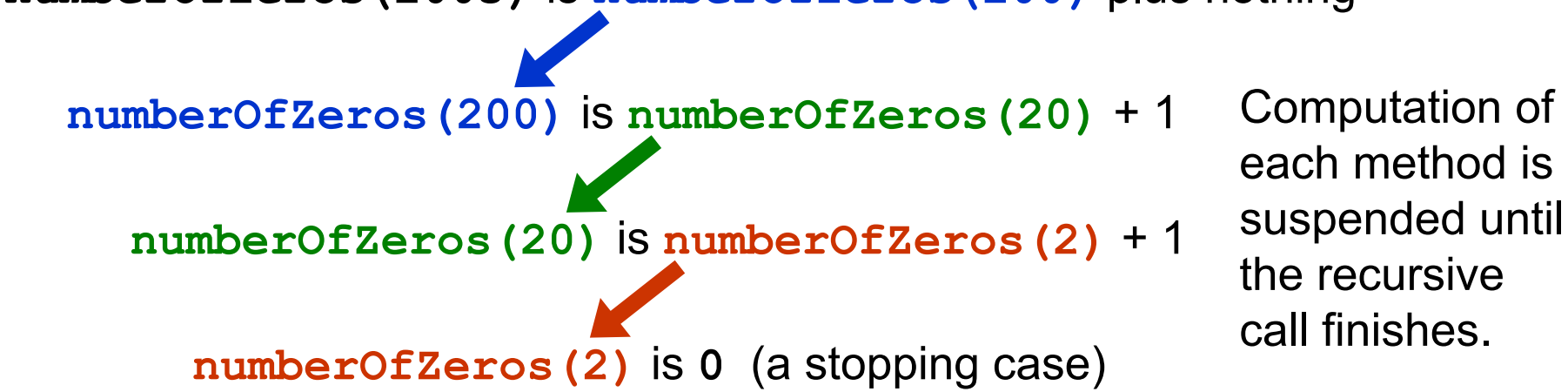
Which is (are) the recursive case(s)? Why?

If n is two or more digits long, then the number of zero digits is (the number of zeros in n with the last digit removed) plus an additional 1 if that digit is zero.

Recursive Calls

```
public static int numberOfZeros(int n)
{
   if (n==0)
      return 1;
   else if (n < 10)  // and not 0
      return 0;  // 0 for no zeros
   else if (n%10 == 0)
      return (numberOfZeros(n/10) + 1);
   else  // n%10 != 0
      return (numberOfZeros(n/10));
}
```

Each method invocation will execute one of the `if-else` cases shown at right.



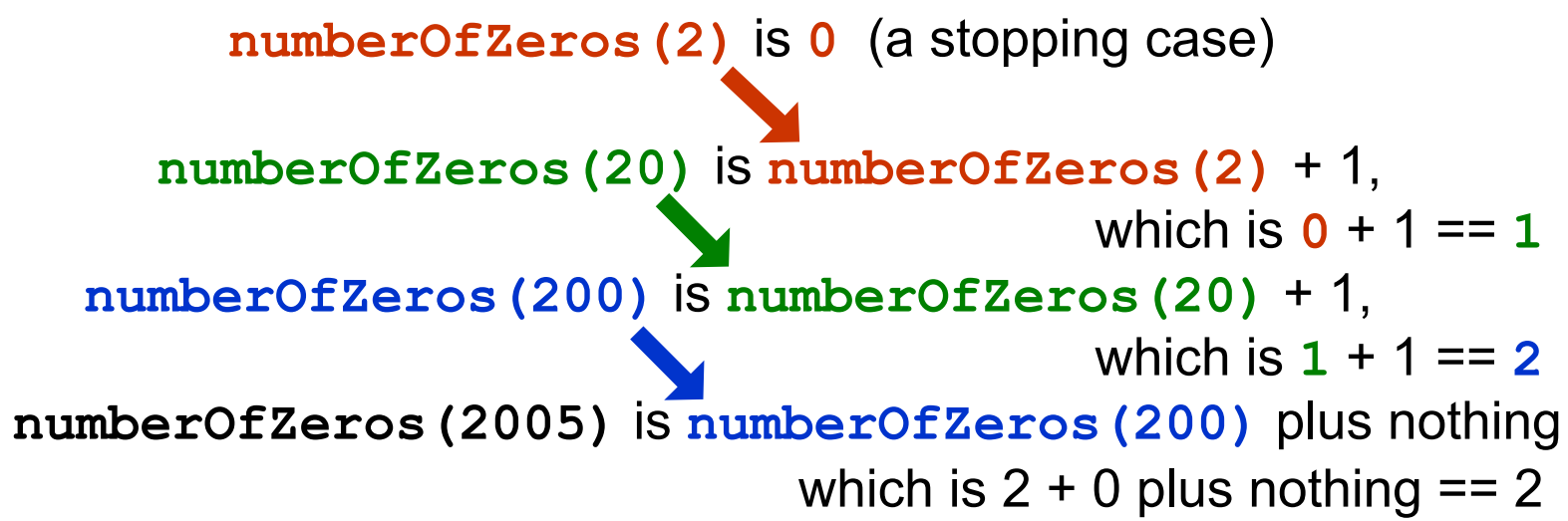
Computation of each method is suspended until the recursive call finishes.

Recursive Calls Returning

Suspended computations completed as follows:

(*bottom* to *top* on previous slide)

```
public static int numberOfZeros(int n)
{
   if (n==0)
      return 1;
   else if (n < 10)  // and not 0
      return 0;  // 0 for no zeros
   else if (n%10 == 0)
      return (numberOfZeros(n/10) + 1);
   else  // n%10 != 0
      return (numberOfZeros(n/10));
}
```



Programming Tip: Ask Until the User Gets It Right

```
public void getCount()
{
   System.out.println("Enter a positive number:");
   count = SavitchIn.readLineInt();
   if (count <= 0)
   {
      System.out.println("Input must be positive.
      System.out.println("Try again.");
      getCount();  //start over
   }
}
```

read a number

Use a recursive call to get another number.

`getCount` method from class `CountDown`

Recursion continues until user enters valid input.

The "Name in the Phone Book" Problem Revisited

A recursive solution to the problem was shown in pseudocode on an earlier slide and is repeated here:

```
Search:
   middle page = (first page + last page)/2
   Open the phone book to middle page;
   If (name is on middle page)
            then done;//this is the base case
   else if (name is alphabetically before middle page)
            last page = middle page//redefine search area to
   front half
            Search//recursive call with reduced number of pages
   else //name must be after middle page
            first page = middle page//redefine search area to
   back half
            Search//recursive call with reduced number of pages
```

Binary Search Algorithm

- Searching a list for a particular value is a very common problem
  - » searching is a thoroughly-studied topic
  - » *sequential* and *binary* are two common search algorithms

- *Sequential search:* inefficient, but easy to understand and program

- *Binary search*: more efficient than sequential, but it only works if *the list is sorted first!*

- The pseudocode for the "find the name in the phone book" problem is an example of a binary search
  - » notice that names in a phone book are already sorted, so you may use a binary search algorithm

Why Is It Called "Binary" Search?

Compare sequential and binary search algorithms:

How many elements are eliminated from the list each time a value is read from the list and it is not the "target" value?

Sequential search: each time a value is read from the list and it is not the "target" value, *only one item* from the list is eliminated

Binary search: each time a value is read from the list and it is not the "target" value, *half the list* is eliminated!

That is why it is called *binary* -

each unsuccessful test for the target value

reduces the remaining search list by 1/2.

Binary Search Code

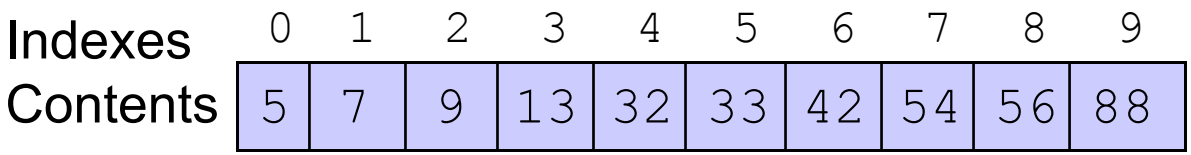
- The `find` method of `ArraySearcher` implements a binary search algorithm
- It returns the index of the entry if the target value is found or -1 if it is not found
- Compare it to the pseudocode for the "name in the phone book" problem

```
private int search(int target, int first, int last)
 {
    int result = -1;//to keep the compiler happy.
    int mid;
    if (first > last)
       result = -1;
    else
    {
       mid = (first + last)/2;

       if (target = = a[mid])
          result = mid;
       else if (target < a[mid])
          result = search(target, first, mid - 1);
       else //(target > a[mid])
          result = search(target, mid + 1, last);
    }

    return result;
 }
```

Binary Search Example

`target` is `33`

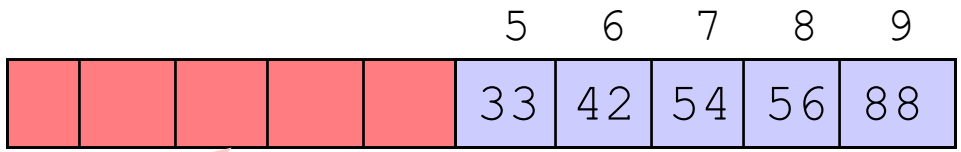
The array **`a`** looks like this:



`mid = (0 + 9) / 2` (which is `4`)

`33 > a[mid]` (that is, `33 > a[4]`)

So, if `33` is in the array, then `33` is one of:

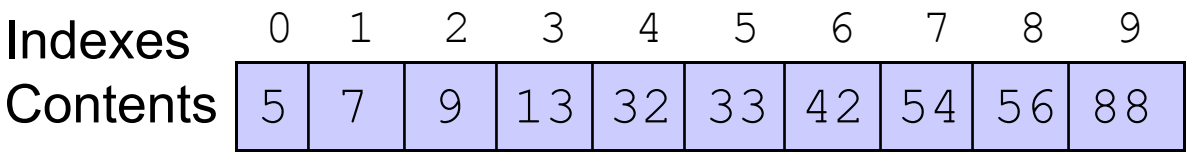


Eliminated half of the remaining elements from consideration because array elements are sorted.

Binary Search Example

`target` is `33`

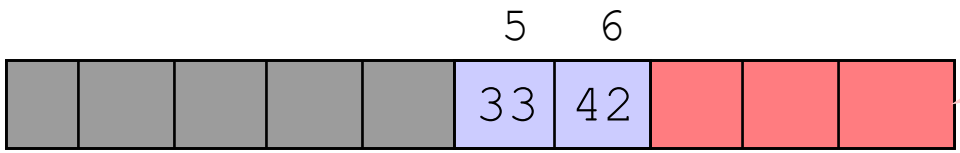
The array **`a`** looks like this:



`mid = (5 + 9) / 2` (which is `7`)

`33 < a[mid]` (that is, `33 < a[7]`)

So, if `33` is in the array, then `33` is one of:

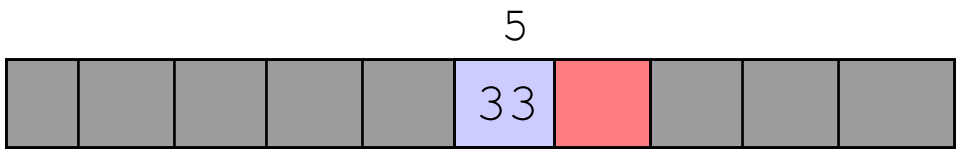


Eliminate half of the remaining elements

`mid = (5 + 6) / 2` (which is `5`)

`33 == a[mid]`

So we found `33` at index `5`:



Merge Sort — A Recursive Sorting Method

- Example of divide and conquer algorithm
- Divides array in half and sorts halves recursively
- Combines two sorted halves

Merge Sort Algorithm to Sort the Array `a`

If the array `a` has only one element, do nothing (stopping case).

Otherwise, do the following (recursive case):

Copy the first half of the elements in `a` to a smaller array named `front`.

Copy the rest of the elements in the array `a` to another smaller array named `tail`.

Sort the array `front` with a recursive call.

Sort the array `tail` with a recursive call.

Merge the elements in the arrays `front` and `tail` into the array `a`.

Merge Sort

```
public static void sort(int[] a)
{
   if (a.length >= 2)
   {
      int halfLength = a.length / 2;
      int[] front = new int[halfLength];
      int[] tail = new int[a.length - halfLength];
      divide(a, front, tail);
      sort(front);
      sort(tail);
      merge(a, front, tail);
   }
   // else do nothing.
}
```

do recursive case if true, base case if false

recursive calls

make "smaller" problems by dividing array

make "smaller" problems by dividing array

base case: a.length == 1 so a is sorted and no recursive call is necessary.

Summary

- If a method definition includes an invocation of the very method being defined, the invocation is called a *recursive call*.
- Recursive calls are legal in Java and sometimes can make code easier to read.
- To avoid infinite recursion, a recursive method definition should contain two kinds of cases: one or more recursive calls and one or more stopping cases that do not involve any recursive calls.
- Recursion can be a handy way to write code that says "if there is a problem then start the whole process over again."