11. Recursion

Harald Gall, Prof. Dr.

Institut für Informatik

Universität Zürich

http://seal.ifi.uzh.ch/info1

University of Zurich

Department of Informatics

- *You think you know when you learn,*
  are more sure when you can write,
  even more when you can teach,
  but certain when you can program.

  -- Alan J. Perlis

© 2005 W. Savitch, Pearson Prentice Hall

Objectives

- become familiar with the idea of recursion
- learn to use recursion as a programming tool
- become familiar with the binary search algorithm as an example of recursion
- become familiar with the merge sort algorithm as an example of recursion

The Basics of Recursion: Outline

- Introduction to Recursion
- How Recursion Works
- Recursion versus Iteration
- Recursive Methods That Return a Value

s.e.a.l.
software evolution & architecture lab

Introduction to Recursion

- A recursive algorithm will have one subtask that is a small version of the entire algorithm's task
- A Java method definition is *recursive* if it contains an invocation of itself.
- The method *continues to call itself*, with ever simpler cases, until a base case is reached which can be resolved without any subsequent recursive calls.

Example: Exponent

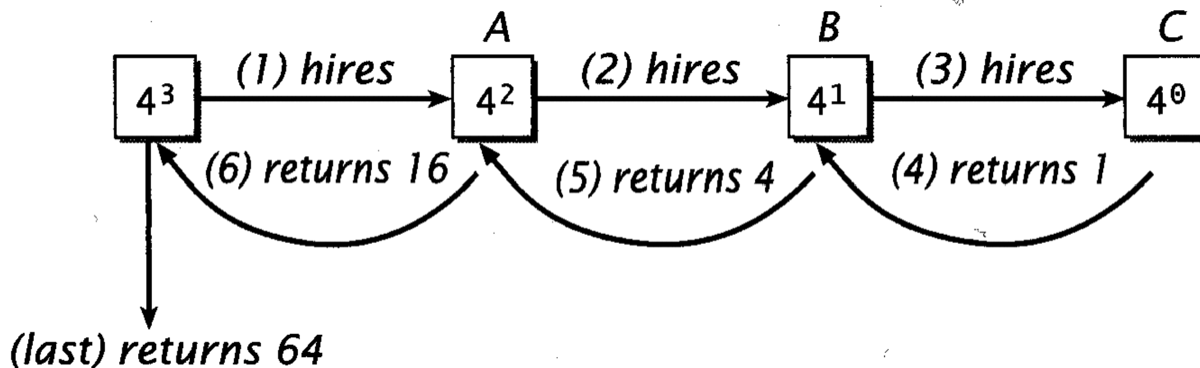
- Aufgabe: x^y berechnen

```
private int power(int x, int y) {
   // y>=0 returns x**y

}
```

- $x^y = 1 * x * x * ... * x$ (y times)
  - wenn y == 0, dann keine weiteren Multiplikationen mehr
  - wenn y > 0, dann
    - berechne $x^{(y-1)}$ und ermittle das Ergebnis $x * x^{(y-1)}$

Exponent /2

```
private int power(int x, int y) {
   // y>=0 returns x**y
   int assistantResult;

   if (y == 0)
      return 1;
   else {
      assistantResult = power(x, y-1);
      return x * assistantResult;
   }
}
```



Termination

- Es gibt eine Return-Bedingung, die keinen weiteren rekursiven Aufruf durchführt:

```
if (y == 0)
   return 1;
else { ...
   // rekursiver Aufruf
}
```

- Der Terminationsschritt ist essenziell für jede rekursive Funktion!
- Der Terminationscode muss vor dem rekursiven Aufruf platziert sein!

Methodenaufruf

- Eine Nachricht wird an das Empfänger-Objekt gesendet; der Sender wartet auf das Ergebnis
- Der Empfänger erzeugt die lokalen Variablen der Methode (Parameter und andere lokalen Variablen)
- Die Parameter erhalten die Werte der Argumente
- Die Methode wird ausgeführt
- Die Methode terminiert und verwirft die lokalen Variablen; ggf. wird ein Return-Wert an den Sender retourniert
- Der Sender setzt seine Verarbeitung fort

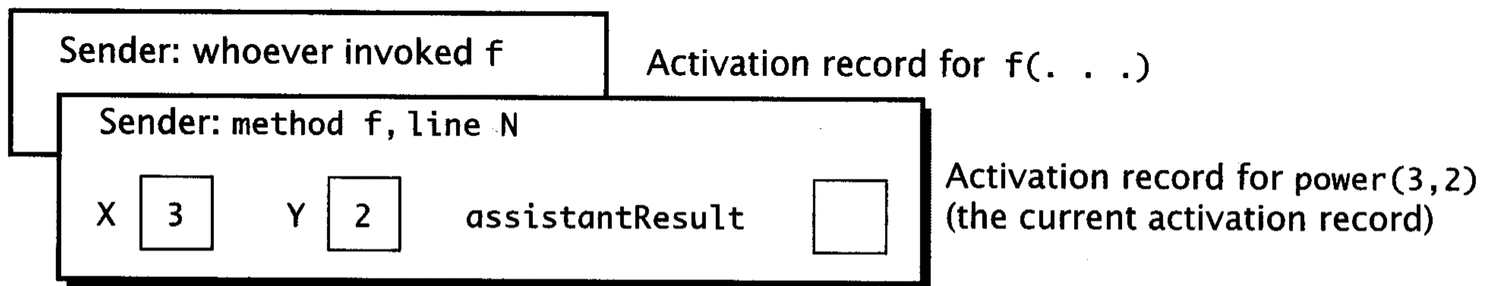
- Speicher wird alloziert für
  - die lokalen Variablen
  - die verwendeten Parameter
  - die Lokation des Codes vom Methodenaufruf im Sender (i.e. Return-Adresse)

Activation records

- `f()` ruft `power()` auf:

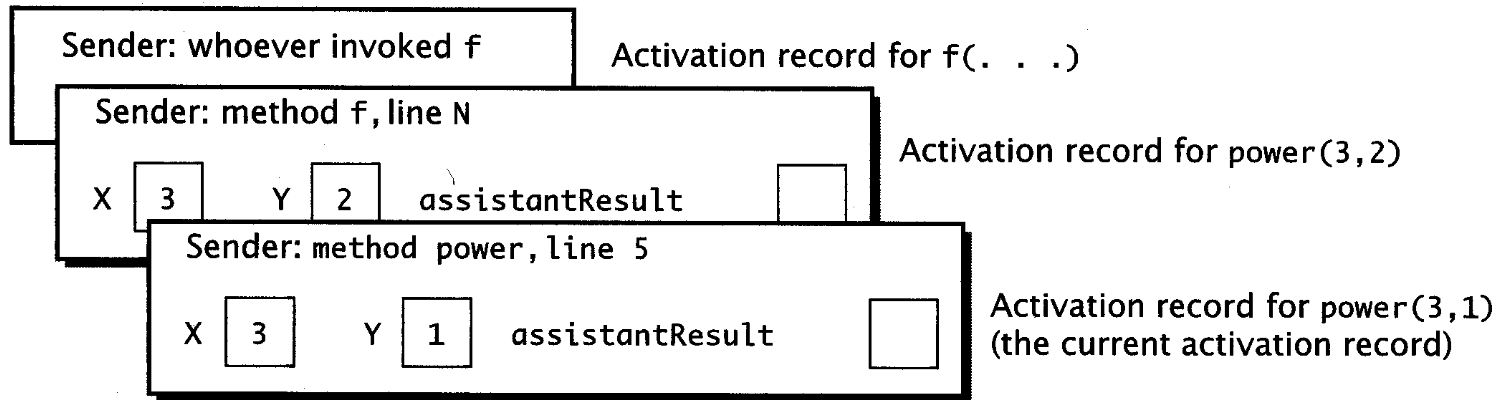
```
void f(..) {
    ...
    int q = power (3,2);
    ...
}
```

- Activation record = Speicherblock, der die aktuellen Parameter und lokalen Variablen mit der Return-Adresse enthält:



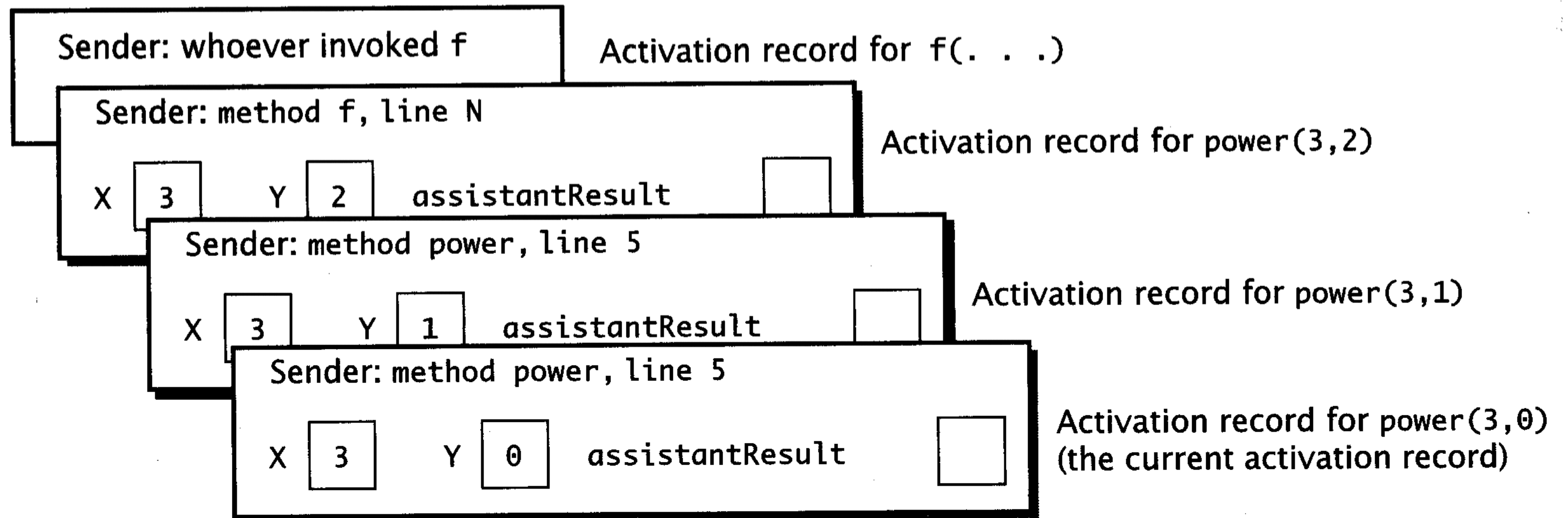
Stack of Activation records /2

- Nach dem Aufruf von `power (3,1)`



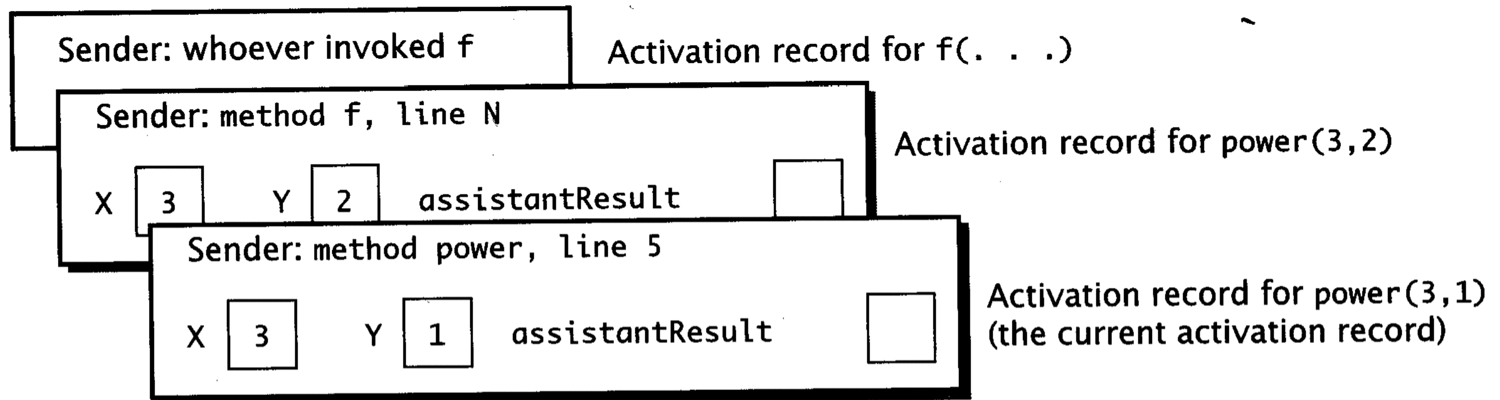
Stack of Activation records /3

- Nach dem Aufruf von `power (3,0)`

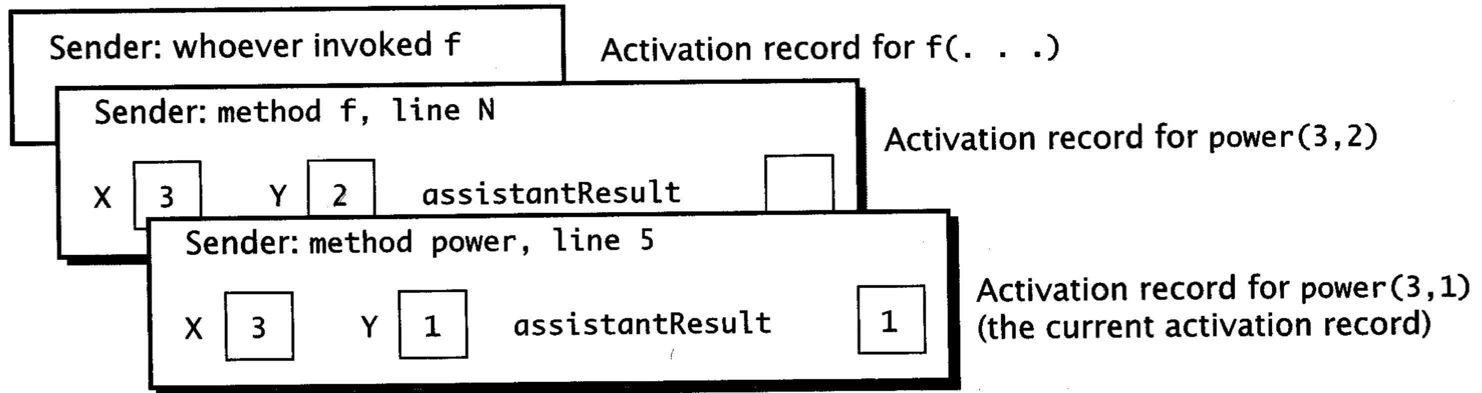


Return

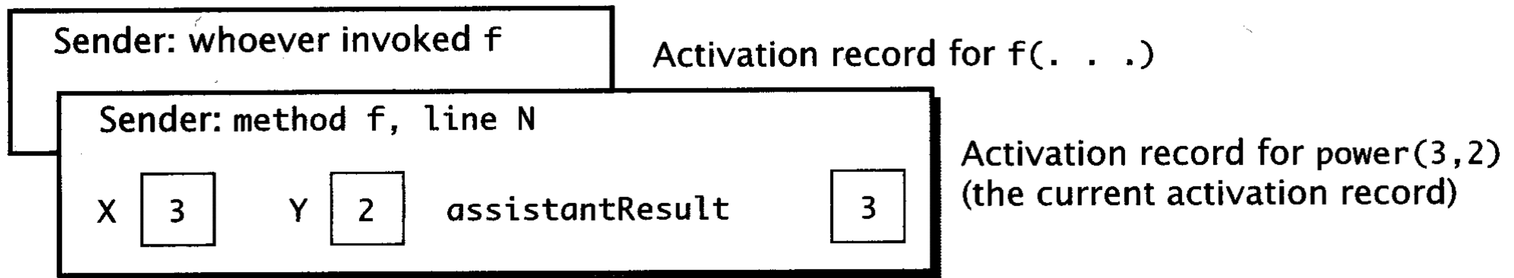
- Ein return-Statement
  - evaluiert den Return-Wert (i.e. 1)
  - löscht den aktuellen Activation Record
  - ersetzt den Ausdruck, der die Methode aufgerufen hat, mit dem Return-Wert
  - setzt die Ausführung des Senders fort

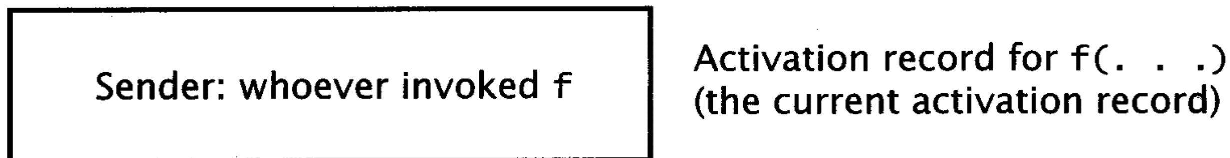


Return /2



Return /3





Example: Digits to Words

- Write a definition that accepts a single integer and produces words representing its digits.
- example
  - input: `223`
  - output: `two two three`
- recursive algorithm
  - output all but the last digit as words
  - output the word for the last digit

Digit to Words: Specification

- If number has multiple digits, decompose algorithm into two subtasks
  1. Display all digits but the last as words
  2. Display last digit as a word
- First subtask is smaller version of original problem
  - Same as original task, one less digit

Case Study

- Algorithm for `displayAsWords(number)`

  1. `displayAsWords` (number after deleting last digits)

  2. `System.out.print` (`getWordFromDigit`(last digit of number + " ")

Case Study

- **`class RecursionDemo`**

Sample screen output

```
Enter an integer:
987
The digits in that number are:
nine eight seven
If you add ten to that number,
the digits in the new number are:
nine nine seven
```

How Recursion Works

- Executing recursive call

`displayAsWords(987)` is equivalent to executing:

```
{//Code for invocation of displayAsWords(987)
     if (987 < 10)
          System.out.print(getWordFromDigit(987) + " ");
     else //987 has two or more digits
     {
          displayAsWords(987 / 10);
          System.out.print(getWordFromDigit(987 % 10) + " ");
     }
}
```

Computation waits here for the completion of the recursive call.

How Recursion Works

- Executing recursive call

`displayAsWords(987/10)` is equivalent to `displayAsWords(98)`, which is equivalent to executing:

```
{//Code for invocation of displayAsWords(98)
    if (98 < 10)
        System.out.print(getWordFromDigit(98) + " ");
    else //98 has two or more digits
    {
        displayAsWords(98 / 10);
        System.out.print(getWordFromDigit(98 % 10) + " ");
    }
}
```

Computation waits here for the completion of the recursive call.

How Recursion Works

- Executing recursive call

displayAsWords(98/10) is equivalent to displayAsWords(9), which is equivalent to executing:

```
{//Code for invocation of displayAsWords(9)
    if (9 < 10)
        System.out.print(getWordFromDigit(9) + " ");
    else //9 has two or more digits
    {
        displayAsWords(9 / 10);
        System.out.print(getWordFromDigit(9 % 10) + " ");
    }
}
```

Another recursive call does not occur.

How Recursion Works

- Nothing special is required to handle a call to a recursive method, whether the call to the method is from outside the method or from within the method.
- At each call, the needed arguments are provided, and the code is executed.
- When the method completes, control returns to the instruction following the call to the method.

How Recursion Works, cont.

- Consider several methods `m1, m2, …, mn` with method `m1` calling method `m2`, method `m2` calling method `m3`,…, calling method `mn`.
  - When each method completes, control returns to the instruction following the call to the method.
- In recursion, methods `m1, m2, …, mn` are all the same method, but each call results in a distinct execution of the method.

How Recursion Works, cont.

- As always, method `m1` cannot complete execution until method `m2` completes execution, method `m2` cannot complete execution until method `m3` completes execution, …, until method `mn` completes execution.
- If method `mn` represents a stopping case, it can complete execution, …, then method `m2` can complete execution, then method `m1` can complete execution.

Recursion Guidelines

- The definition of a recursive method typically includes an `if-else` statement.
  - One branch represents a base case which can be solved directly (without recursion).
  - Another branch includes a recursive call to the method, but with a "simpler" or "smaller" set of arguments.
- Ultimately, a base case must be reached.

Keys to Successful Recursion

- Must have a branching statement that leads to different cases
- One or more of the branches should have a recursive call of the method
  - Recursive call must us "smaller" version of the original argument
- One or more branches must include *no* recursive call
  - This is the base or stopping case

Infinite Recursion

- If the recursive invocation inside the method does not use a “simpler” or “smaller” parameter, a base case may never be reached.
- Such a method continues to call itself forever (or at least until the resources of the computer are exhausted as a consequence of *stack overflow*)
- This is called *infinite recursion*

Infinite Recursion

- Suppose we leave out the stopping case

```
public static void displayAsWords(int number)//Not quite right
{
    displayAsWords(number / 10);
    System.out.print(getWordFromDigit(number % 10) + " ");
}
```

- Nothing stops the method from repeatedly invoking itself
  - Program will eventually crash when computer exhausts its resources (stack overflow)

Recursive Versus Iterative

- Any method including a recursive call can be rewritten
  - To do the same task
  - Done *without* recursion
- Non recursive algorithm uses *iteration*
  - Method which implements is *iterative method*
- `class IterativeDemo`

Recursive Versus Iterative

- Recursive method
  - Uses more storage space than iterative version
  - Due to overhead during runtime
  - Also runs slower
- However in *some* programming tasks, recursion is a better choice, a more elegant solution

Recursive Methods that Return a Value

- Follow same design guidelines as stated previously
- Second guideline also states
  - One or more branches includes recursive invocation *that leads to the returned value*
- View program with recursive value returning method, listing 11.3

  `class RecursionDemo2`

Recursive Methods that Return a Value

```
Enter a nonnegative number:
2008
2008 contains 2 zeros.
```

Sample screen output

- Note recursive method **`NumberOfZeros`**
  - Has two recursive calls
  - Each returns value assigned to **`result`**
  - Variable **`result`** is what is returned

Recursion vs. Iteration, cont.

- A recursive version of a method typically executes less efficiently than the corresponding iterative version.
- This is because the computer must keep track of the recursive calls and the suspended computations.
- However, it can be much easier to write a recursive method than it is to write a corresponding iterative method.

Overloading is Not Recursion

- If a method name is overloaded and one method calls another method with the same name but with a different parameter list, this is **not** recursion
- Of course, if a method name is overloaded and the method calls itself, this **is** recursion
- Overloading and recursion are neither synonymous nor mutually exclusive

Programming with Recursion: Outline

- Programming Example: Insisting that User Input Be Correct
- Case Study: Binary Search
- Programming Example: Merge Sort – A Recursive Sorting Method

s.e.a.l.
software evolution & architecture lab

Programming Example

- Insisting that user input be correct
  - Program asks for a input in specific range
  - Recursive method makes sure of this range
  - Method recursively invokes itself as many times as user gives incorrect input
- View program, listing 11.4

```
class CountDown
```

Programming Example

```
Enter a positive integer:
0
Input must be positive.
Try again.
Enter a positive integer:
3
Counting down:
3, 2, 1, 0, Blast Off!
```

Sample screen output

Example: Search for a Name in a Phone Book

- Open the phone book to the middle.
- If the name is on this page, you're done.
- If the name alphabetically precedes the names on this page, use the same approach to search for the name in the first half of the phone book.
- Otherwise, use the same approach to search for the name in the second half of the phone book.

Case Study

- Binary Search
  - We design a recursive method to tell whether or not a given number is in an array
  - Algorithm assumes array is sorted
- First we look in the middle of the array
  - Then look in first half or last half, depending on value found in middle

Binary Search

- Draft 1 of algorithm

```
1. m = an index between 0 and (a.length - 1)
2. if (target == a[m])
3.       return m;
4. else if (target < a[m])
5.       return the result of searching a[0] through a[m - 1]
6. else if (target > a[m])
7.       return the result of searching a[m + 1] through a[a.length - 1]
```

  - Algorithm requires additional parameters

Binary Search

- Draft 2 of algorithm to search **`a[first]`** through **`a[last]`**

```
1. mid = approximate midpoint between first and last
2. if (target == a[mid])
3.       return mid
4. else if (target < a[mid])
5.       return the result of searching a[first] through a[mid – 1]
6. else if (target > a[mid])
7.       return the result of searching a[mid + 1] through a[last]
```

  - What if target is not in the array?

Binary Search

- Final draft of algorithm to search **`a[first]`** through **`a[last]`** to find **`target`**

```
1. mid = approximate midpoint between first and last
2. if (first > last)
3.       return -1
4. else if (target == a[mid])
5.       return mid
6. else if (target < a[mid])
7.       return the result of searching a[first] through a[mid - 1]
8. else if (target > a[mid])
9.       return the result of searching a[mid + 1] through a[last]
```

Binary Search

- Figure 11.2a Binary search example

target *is* 33

Eliminate half of the array elements:

mid

0	1	2	3	4	5	6	7	8	9
5	7	9	13	32	33	42	54	56	88

1. mid = (0 + 9)/2 (which is 4).
2. 33 > a[mid] (that is, 33 > a[4]).
3. So if 33 is in the array, 33 is one of a[5], a[6], a[7], a[8], a[9].

Binary Search

- Figure 11.2b Binary search example

Eliminate half of the remaining array elements:

mid

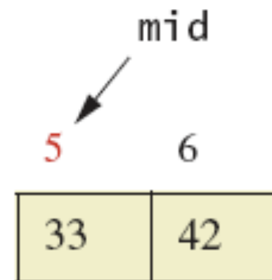
5	6	7	8	9
33	42	54	56	88

1. `mid = (5 + 9)/2` (which is 7).
2. 33 < `a[mid]` (that is, 33 < `a[7]`).
3. So if 33 is in the array, 33 is one of `a[5], a[6]`.

Binary Search

- Figure 11.2c Binary search example

Eliminate half of the remaining array elements:



1. `mid = (5 + 6)/2` (which is 5).
2. 33 equals `a[mid]`, so we found 33 at index 5.

`33` *found in* `a[5]`.

Binary Search

- View final code, listing 11.5
  `class ArraySearcher`
- Note demo program, listing 11.6
  `class ArraySearcherDemo`

Binary Search

```
Enter 10 integers in increasing order.
Again?
yes
Enter a value to search for:
0
0 is at index 0
Again?
yes
Enter a value to search for:
2
2 is at index 1
Again?
yes
Enter a value to search for:
13
13 is not in the array.
Again?
no
May you find what you're searching for.
```

Sample screen output

Example: Merge Sort

- Merge sort – A recursive sorting method
- A divide-and-conquer algorithm
  - Array to be sorted is divided in half
  - The two halves are sorted by recursive calls
  - This produces two smaller, sorted arrays which are merged to a single sorted array

Merge Sort

- Algorithm to sort array `a`

1. If the array a has only one element, do nothing (base case).
   Otherwise, do the following (recursive case):
2. Copy the first half of the elements in a to a smaller array named `firstHalf`.
3. Copy the rest of the elements in the array a to another smaller array named `lastHalf`.
4. Sort the array `firstHalf` using a recursive call.
5. Sort the array `lastHalf` using a recursive call.
6. Merge the elements in the arrays `firstHalf` and `lastHalf` into the array a.

- View Java implementation, listing 11.7
  `class MergeSort`

Merge Sort

- View demo program, listing 11.8

  `class MergeSortDemo`

```
Array values before sorting:
7 5 11 2 16 4 18 14 12 30
Array values after sorting:
2 4 5 7 11 12 14 16 18 30
```

Sample screen output

Merge Sort

- Efficient sorting algorithms often are stated recursively.
- One such sort, merge sort, can be used to sort an array of items.
- Merge sort takes a “divide and conquer” approach.
  - The array is divided in halves and the halves are sorted recursively.
  - Sorted subarrays are merged to form a larger sorted array.

Merge Sort, cont.

- pseudocode

```
If the array has only one element,
    stop.
Otherwise
    Copy the first half of the elements
        into an array named front.
    Copy the second half of the elements
        into an array named back.
    Sort array front recursively.
    Sort array tail recursively.
    Merge arrays front and tail.
```

Merging Sorted Arrays

- The smallest element in array `front` is `front[0]`.
- The smallest element in array `tail` is `tail[0]`.
- The smallest element will be either `front[0]` or tail `[0]`.
- Once that element is removed from either array `front` or array `tail`, the smallest remaining element once again will be at the beginning or array `front` or array `tail`.

Merging Sorted Arrays, cont.

- Generalizing, two sorted arrays can be merged by selectively removing the smaller of the elements from the beginning of (the remainders) of the two arrays and placing it in the next available position in a larger “collector” array.
- When one of the two arrays becomes empty, the remainder of the other array is copied into the “collector” array.

Merging Sorted Arrays, cont.

```
int frontIndex = 0, tailIndex = 0, aIndex = 0;
while (frontIndex < front.length) &&
     (tailIndex < tail.length))
{
     if(front[frontIndex] < tail[tailIndex]}
     {
          a[aIndex] = front[frontIndex];
          aIndex++;
          frontIndex++;
     }
```

Merging Sorted Arrays, cont.

```
    else
    {
        a[aIndex] = tail[tailIndex];
       aIndex++;
       tailIndex++
    }
}
```

Merging Sorted Arrays, cont.

- Typically, when either array `front` or array `tail` becomes empty, the other array will have remaining elements which need to be copied into array `a`.
- Fortunately, these elements are sorted and are larger than any elements already in array `a`.

Merge Sort, cont.

- `class MergeSort`

```
/**
 Class for sorting an array of ints from smallest to largest,
 using the merge sort algorithm.
*/
public class MergeSort
{
    /**
     Precondition: Every indexed variable of a has a value.
     Action: Sorts a so that a[0] <= a[1] <= ... <= a[a.length - 1].
    */
    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. a.length == 1, so a is sorted.
    }

    /**
     Precondition: a.length = front.length + tail.length.
     Postcondition: All the elements of a are divided
     between the arrays front and tail.
    */
    private static void divide(int[] a, int[] front, int[] tail)
    {
        int i;
        for (i = 0; i < front.length; i++)
            front[i] = a[i];

        for (i = 0; i < tail.length; i++)
            tail[i] = a[front.length + i];
    }
```

Display 11.9

The MergeSort Class

Merge Sort, cont.

```
/**
 Precondition: Arrays front and tail are sorted from smallest
 to largest, and a.length = front.length + tail.length.
 Postcondition: a contains all the values from front and tail,
 and a is sorted from smallest to largest.
*/
private static void merge(int[] a, int[] front, int[] tail)
{
    int frontIndex = 0, tailIndex = 0, aIndex = 0;
    while ((frontIndex < front.length)
                      && (tailIndex < tail.length))
    {
        if (front[frontIndex] < tail[tailIndex])
        {
            a[aIndex] = front[frontIndex];
            aIndex++;
            frontIndex++;
        }
        else
        {
            a[aIndex] = tail[tailIndex];
            aIndex++;
            tailIndex++;
        }
    }
    //At least one of front and tail has been
    //completely copied to a.
    while (frontIndex < front.length)//Copy rest of front,
                                     //if any.
    {
        a[aIndex] = front[frontIndex];
        aIndex++;
        frontIndex++;
    }
    while (tailIndex < tail.length)//Copy rest of tail, if any.
    {
        a[aIndex] = tail[tailIndex];
        aIndex++;
        tailIndex++;
    }
}
}
```

Display 11.9

The MergeSort Class

Merge Sort, cont.

```
public class MergeSortDemo
{
    public static void main(String[] args)
    {
        int[] b = {7, 5, 11, 2, 16, 4, 18, 14, 12, 30};

        System.out.println("Array values before sorting:");
        int i;
        for (i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println();

        MergeSort.sort(b);

        System.out.println("Array values after sorting:");
        for (i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println();
    }
}
```

Display 11.10

Demonstration of the MergeSort Class

Merge Sort, cont.

Screen Output

```
Array values before sorting:
7 5 11 2 16 4 18 14 12 30
Array values after sorting:
2 4 5 7 11 12 14 16 18 30
```

Display 11.10

Demonstration of the MergeSort Class

Merge Sort, cont.

- The merge sort algorithm is much more efficient than the selection sort algorithm

Summary

- Method with self invocation
  - Invocation considered a recursive call
- Recursive calls
  - Legal in Java
  - Can make some method definitions clearer
- Algorithm with one subtask that is smaller version of entire task
  - Algorithm is a recursive method

Summary

- To avoid infinite recursion recursive method should contain two kinds of cases
  - A recursive call
  - A base (stopping) case with no recursive call
- Good examples of recursive algorithms
  - Binary search algorithm
  - Merge sort algorithm