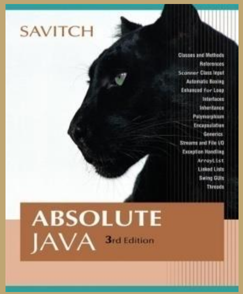
Comp 249
Programming Methodology
Chapter 11 – *Recursion*

Prof. Aiman Hanna

Department of Computer Science & Software Engineering

Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of Absolute Java 3rd Edition by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.



Recursive `void` Methods

- A *recursive* method is a method that includes a call to itself

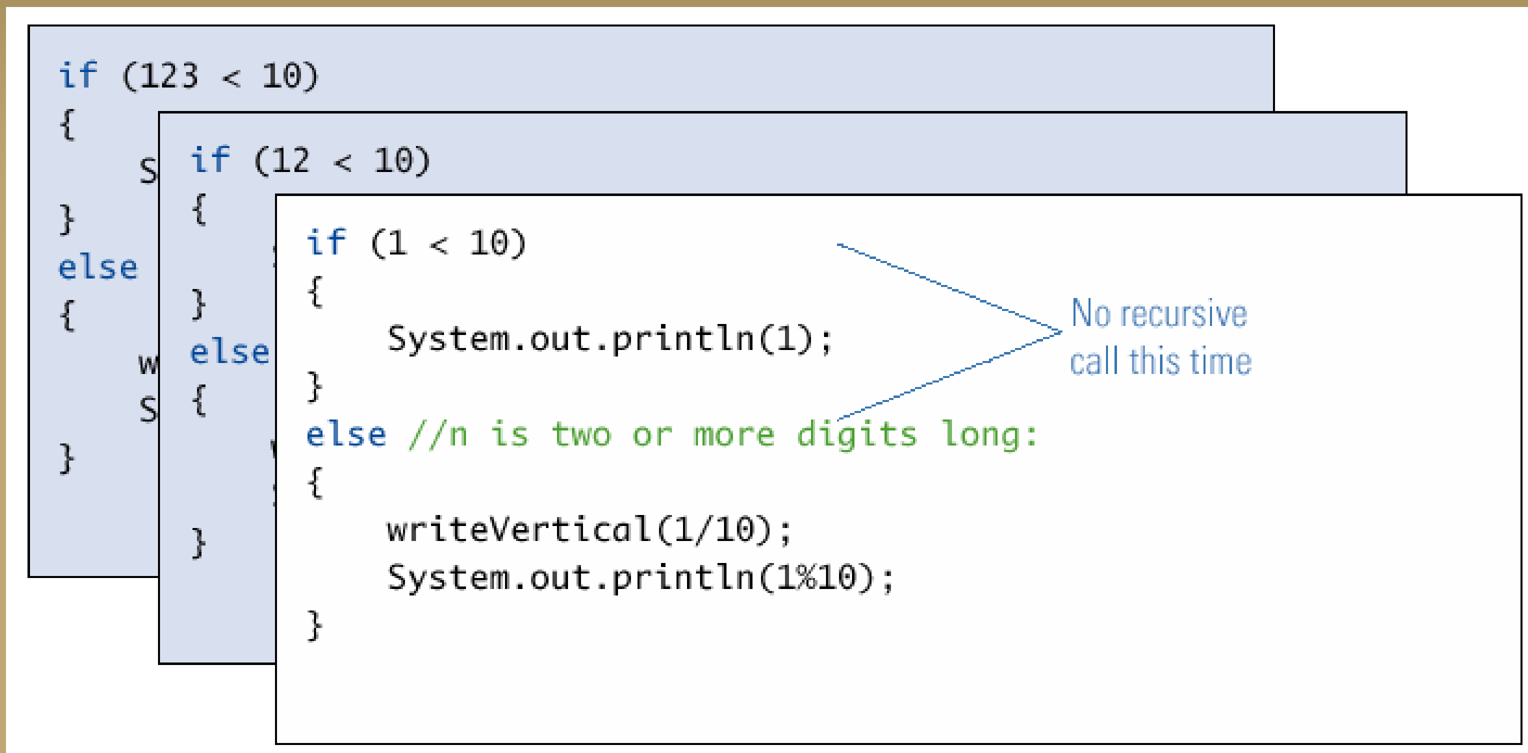
- Recursion is based on the general problem solving technique of breaking down a task into subtasks
  - In particular, recursion can be used whenever one subtask is a smaller version of the original task

See Recursion1.java

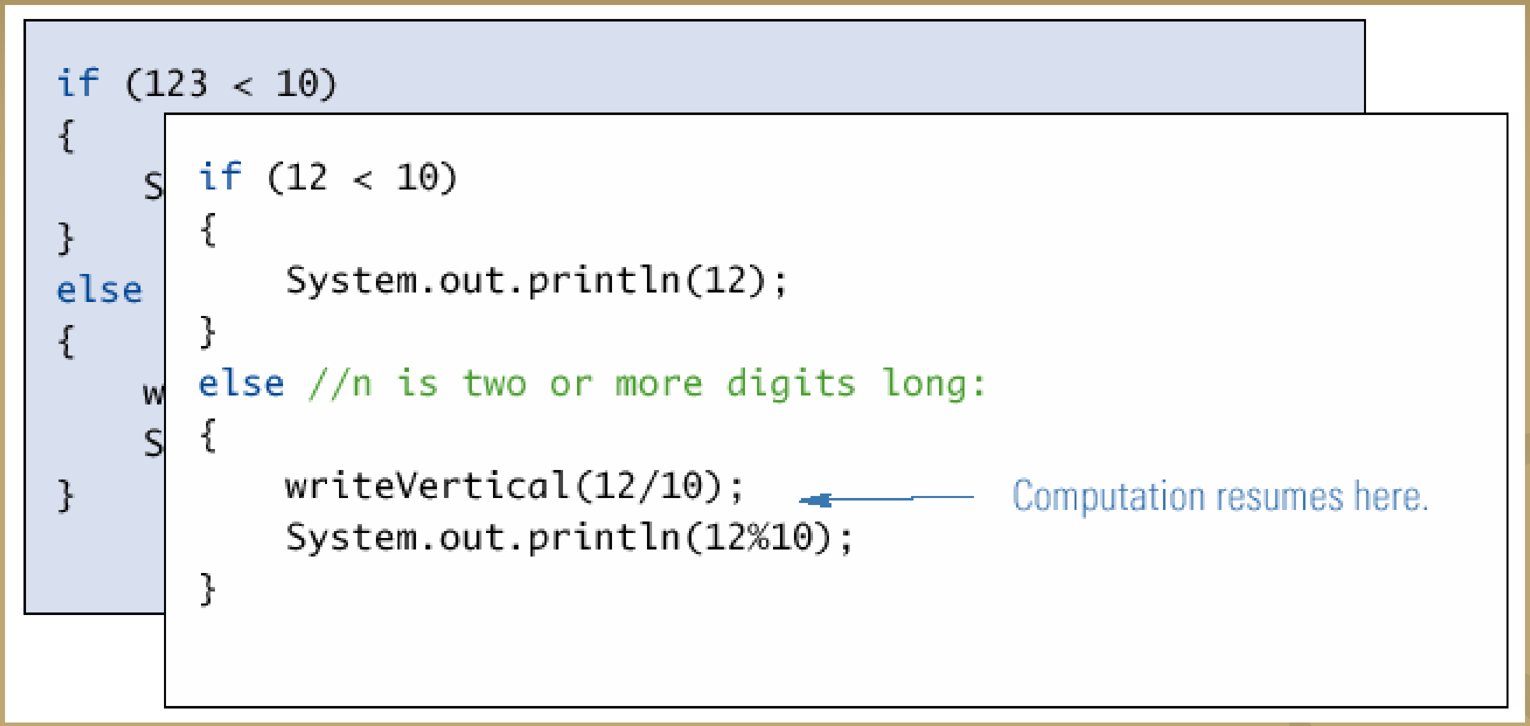
Tracing a Recursive Call

- When the call to the (recursive) method is triggered , the execution of the current call is suspended
- The execution of the current call is resumed once that new call returns
- Similarly, if the new method invocation triggers a new call to the method, the execution is suspended until that later call returns, and so on

Tracing a Recursive Call – An Example



Tracing a Recursive Call – An Example



Tracing a Recursive Call – An Example

```
if (123 < 10)
{
    System.out.println(123);
}
else //n is two or more digits long:
{
    writeVertical(123/10);
    System.out.println(123%10);
}
```

Computation resumes here.

A Closer Look at 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
  - When the recursive call is completed, the computer returns to finish the outer computation

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

General Form of a Recursive Method Definition

- The general outline of a successful recursive method definition is as follows:
  - One or more cases that include one or more recursive calls to the method being defined
    - These recursive calls should solve "smaller" versions of the task performed by the method being defined
  - One or more cases that include no recursive calls: *base cases* or *stopping cases*

Pitfall: Infinite Recursion

- When recursion is used, the series of recursive calls should eventually reach a call of the method that did not involve recursion (a stopping case)
- If, instead, every recursive call had produced another recursive call, then a call to that method would, in theory, run forever
  - This is called *infinite recursion*
  - In practice, such a method runs until the computer runs out of resources, and the program terminates abnormally

See Recursion2.java

Stacks for Recursion

- To keep track of recursion (and other things), most computer systems use a ***stack***
  - A stack is a very specialized kind of memory structure analogous to a container that holds stack of paper
  - As an analogy, there is also an inexhaustible supply of sheets of paper
  - A new sheet is added to the stack by placing it on top of the stack (on top of all previous sheets in the stack
  - Getting an older sheet out from the stack would require that all the ones on top be first removed (more accurately removed and thrown away)

Stacks for Recursion

- Since the last sheet put on the stack is the first sheet that can be taken off the stack, a stack is called a *last-in/first-out* memory structure *(LIFO)*
- Following the previous analogy, 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
  - It writes information about the current method on the *sheet of paper*, and places it on the stack

Stacks for Recursion

- A new *sheet of paper* is then 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 its computation without producing any more recursive calls
  - Its *sheet of paper* is then discarded

- Then the computer goes to the top *sheet of paper* on the stack
  - This sheet contains the partially completed computation that is waiting for the recursive computation that just ended
  - Now it is possible to proceed with that suspended computation

Stacks for Recursion

- After the suspended computation ends, the computer discards its corresponding sheet of paper (the one on top)
- The suspended computation that is below it on the stack now becomes the computation on top of the stack
- This process continues until the computation on the bottom sheet is completed

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
- The stack of paper analogy has its counterpart in the computer
  - The contents 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, but reference a single copy instead

Pitfall: 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*
- A common cause of stack overflow is infinite recursion

See Recursion3.java

Recursion Versus Iteration

- Recursion is not absolutely necessary
  - Any task that can be done using recursion can also be done in a nonrecursive manner
  - A nonrecursive 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

Recursive Methods that Return a Value

- Recursion is not limited to `void` methods
- A recursive method can return a value of any type
- An outline for a successful recursive method that returns a value is as follows:
  - One or more cases in which the value returned is computed in terms of calls to the same method
  - the arguments for the recursive calls should be intuitively "smaller"
  - One or more cases in which the value returned is computed without the use of any recursive calls (the base or stopping cases)

See Recursion4.java

See Recursion5.java

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

  - In specific, `power(x, n)` is the same as `power(x, n-1) * x` for `n > 0`
  - When `n = 0`, then `power(x, n)` should return `1`, This is the stopping case

Thinking Recursively

1. There is no infinite recursion
   - Every chain of recursive calls must reach a stopping case

2. Each 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 follow a technique also known as *mathematical induction*

Recursive Design Techniques

- The same rules can be applied to a recursive `void` method:
  1. There is no infinite recursion
  2. Each stopping case performs the correct action for that case
  3. For each of the cases that involve recursion: if all recursive calls perform their actions correctly, then the entire case performs correctly

Binary Search

- Binary search uses a recursive method to search a sorted array to find a specified value
- The array must be a sorted array; that is:
  a[0] ≤ a[1] ≤a [2] ≤. . . ≤ a[n-1]
- If the value is found, its index is returned
- If the value is not found, -1 is returned
- Note: Each execution 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 in that index, then return it and the search is over
- 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 entire array (or array segment) has been searched in this way without finding the value, then it is not in the array, so return -1 (indicating that there is no index for that value)

<u>See Recursion6.java</u>

Pseudocode for Binary Search

Display 11.5 Pseudocode for Binary Search

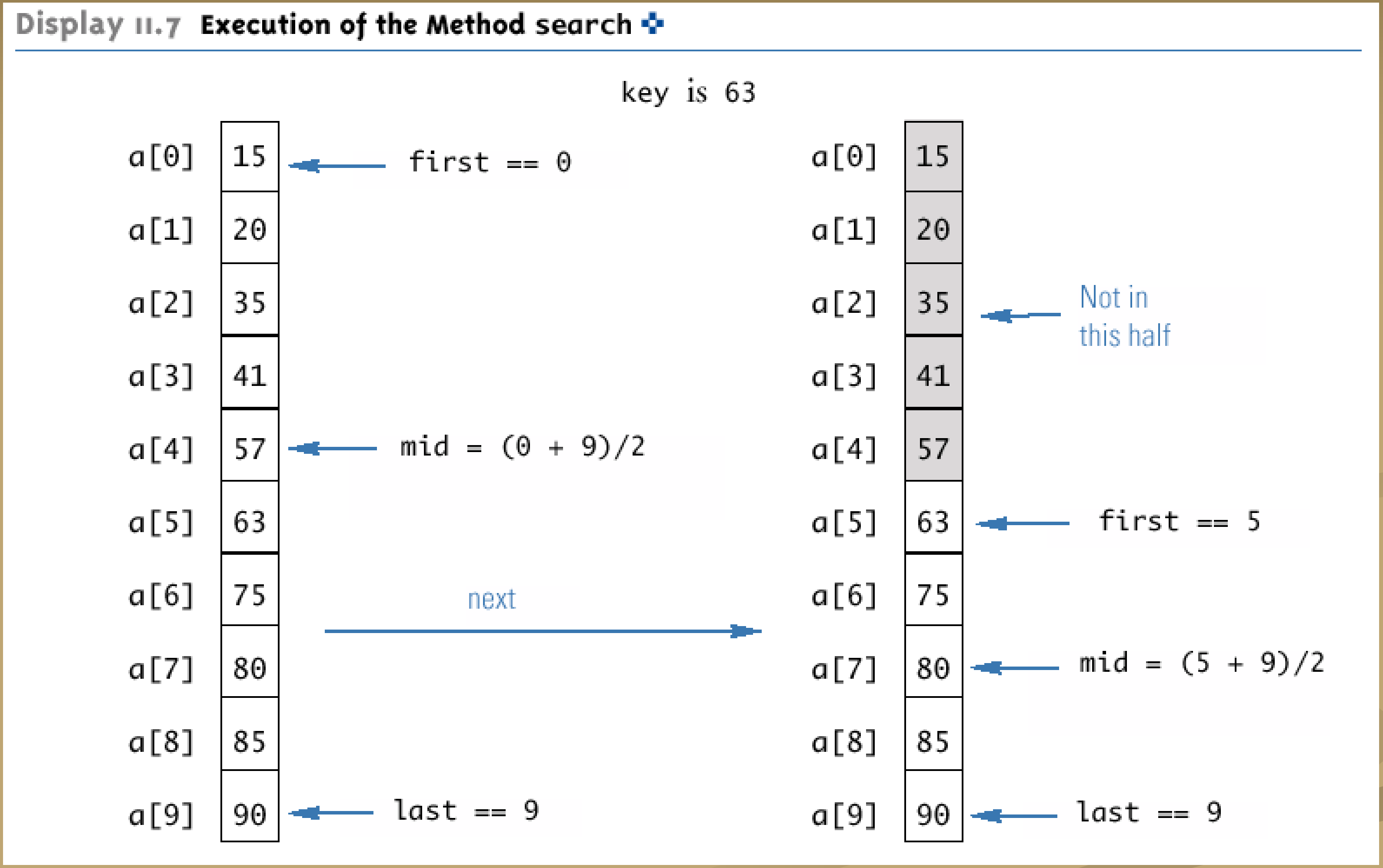
ALGORITHM TO SEARCH a[first] THROUGH a[last]

```
/**
 Precondition:
 a[first]<= a[first + 1] <= a[first + 2] <=... <= a[last]
*/
```

TO LOCATE THE VALUE KEY:

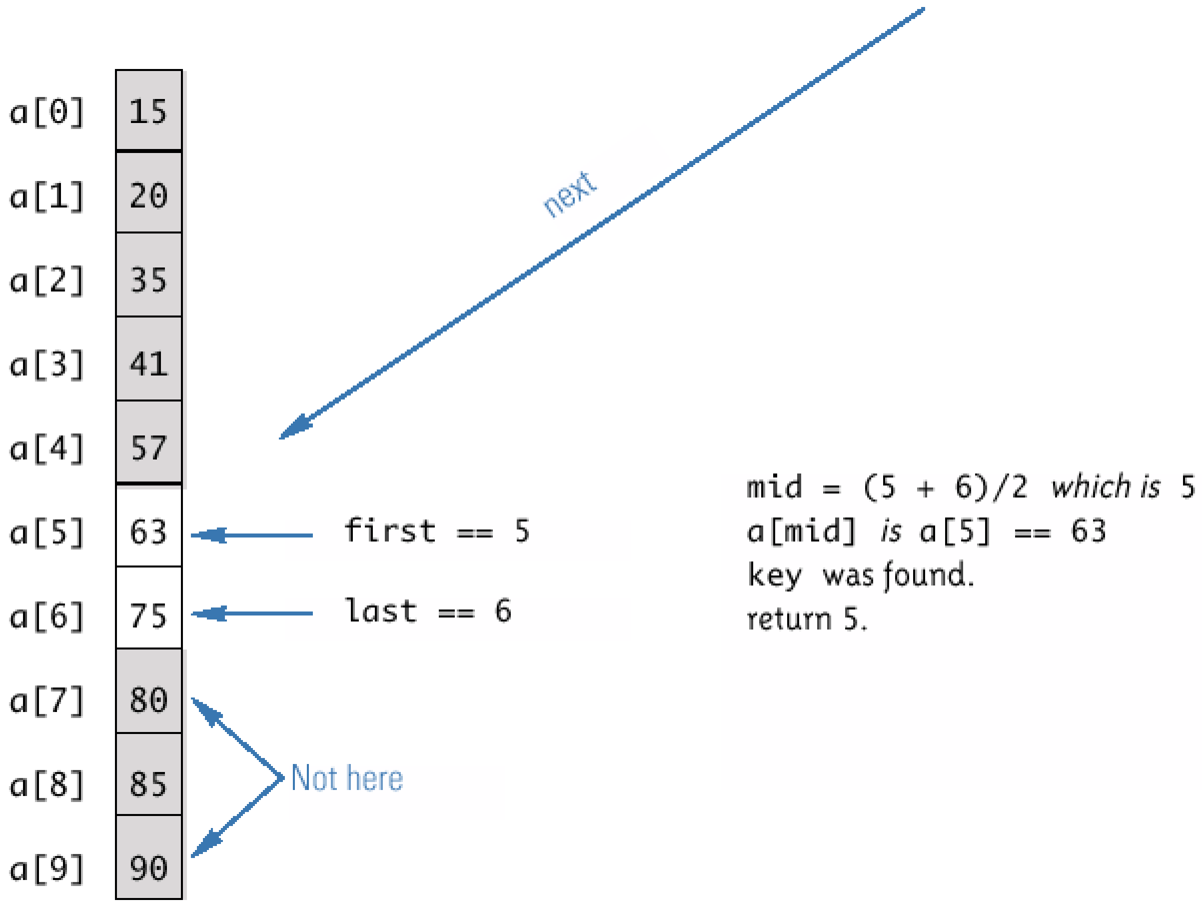
```
if (first > last) //A stopping case
    return -1;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
        return mid;
    else if key < a[mid] //A case with recursion
        return the result of searching a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        return the result of searching a[mid + 1] through a[last];
 }
```

Execution of the Method search – An Example – Part 1 of 2



Execution of the Method search – An Example – Part 2 of 2

Display 11.7 Execution of the Method search (continued)



Checking the `search` Method

1. There is no infinite recursion
   - On each recursive call, the value of **`startIndex`** is increased, or the value of **`endIndex`** is decreased
   - If the chain of recursive calls does not end in some other way, then eventually the method will be called with **`startIndex`** larger than **`endIndex`**

Checking the `search` Method

2. Each stopping case performs the correct action for that case

- If **`startIndex > endIndex`**, there are no array elements between **`A[startIndex]`** and **`a[endIndex]`**, so **`v`** is not in this segment of the array, and **`result`** is correctly set to **`-1`**

- If **`v == A[mid]`**, **`result`** is correctly set to **`mid`**

Checking the `search` Method

3. For each of the cases that involve recursion, *if* all recursive calls perform their actions correctly, *then* the entire case performs correctly

- If **`v < A[mid]`**, then **`v`** must be one of the elements **`A[startIndex]`** through **`A[mid-1]`**, or it is not in the array
- The method should then search only those elements, which it does
- The recursive call is correct, therefore the entire action is correct

Checking the `search` Method

- If `v > A[mid]`, then `v` must be one of the elements `a[mid+1]` through `a[endIndex]`, or it is not in the array
- The method should then search only those elements, which it does
- The recursive call is correct, therefore the entire action is correct

The method **`search`** passes all three tests:

Therefore, it is a good recursive method definition

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 right at the start
  - Then a quarter of the array, then an eighth of the array, and so forth

Efficiency of Binary Search

- Given an array with 1,000 elements, the binary search will only need to compare about 10 array elements to the key value, as compared to an average of 500 for a serial search algorithm
- The binary search algorithm has a worst-case running time that is logarithmic: **O(log *n*)**
  - A serial search algorithm is linear with a worst-case running time of **O(*n*)**
- If desired, the recursive version of the method `search` can be converted to an iterative version

See Recursion7.java