# Lecture 15: Recursion

## CS 0445: Data Structures


## Constantinos Costa

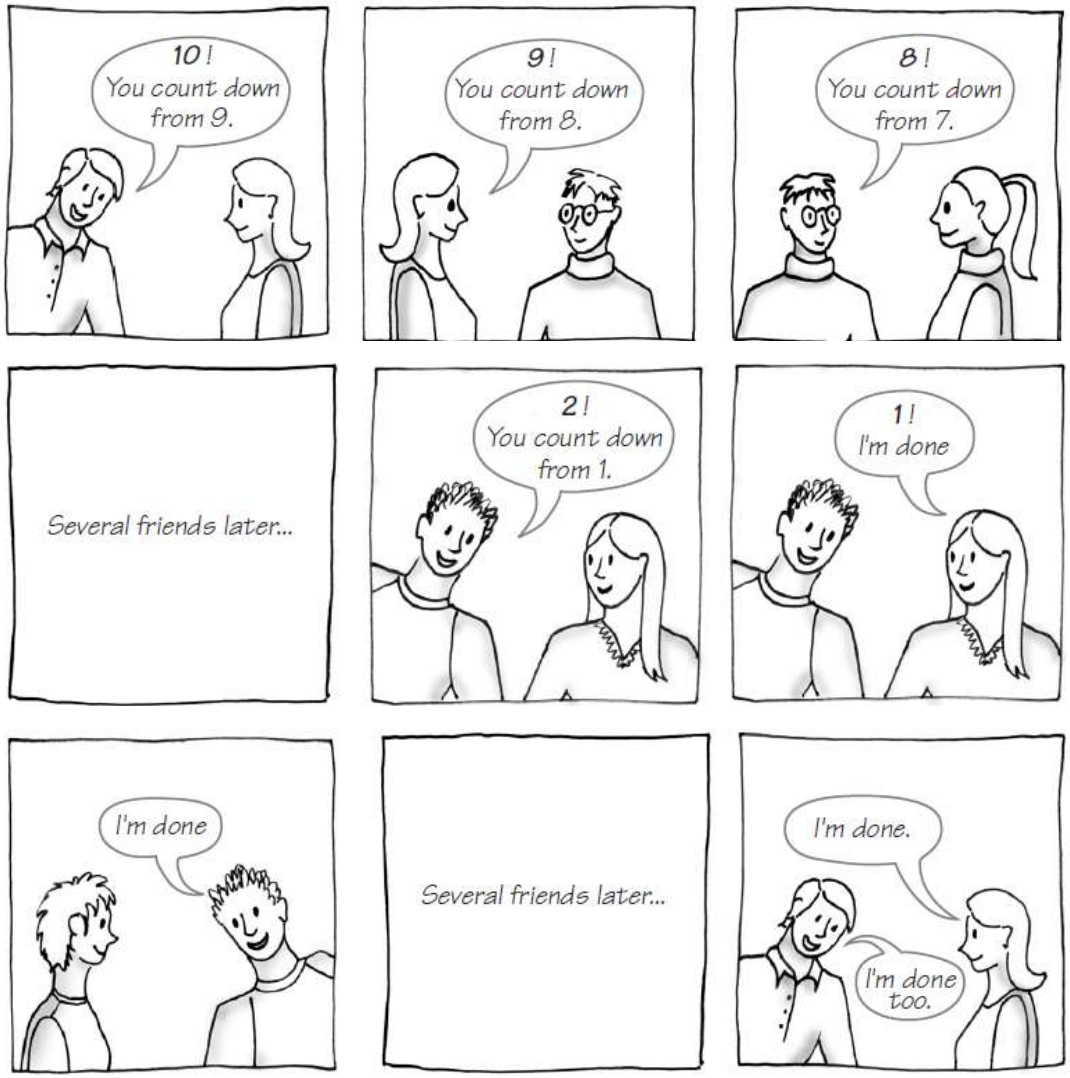

http://db.cs.pitt.edu/courses/cs0445/current.term/

Oct 10, 2019, 8:00-9:15
University of Pittsburgh, Pittsburgh, PA

# What Is Recursion?

- Consider hiring a contractor to build
  - He hires a subcontractor for a portion of the job
  - That subcontractor hires a sub-subcontractor to do a smaller portion of job
- The last sub-sub- … subcontractor finishes
  - Each one finishes and reports “done” up the line

# Example: The Countdown



# Example: The Countdown

- Recursive Java method to do `countDown`.

```
/** Counts down from a given positive integer.
   @param integer  An integer > 0.
*/
public static void countDown(int integer)
{
  System.out.println(integer);
  if (integer > 1)
    countDown(integer - 1);
} // end countDown
```

# Definition

- Recursion is a problem-solving process
  - Breaks a problem into identical but smaller problems.
- A method that calls itself is a recursive method.
  - The invocation is a recursive call or recursive invocation.

# Design Guidelines

- Method must be given an input value
- Method definition must contain logic that involves this input, leads to different cases
- One or more cases should provide solution that does not require recursion
  - Else infinite recursion
- One or more cases must include a recursive invocation

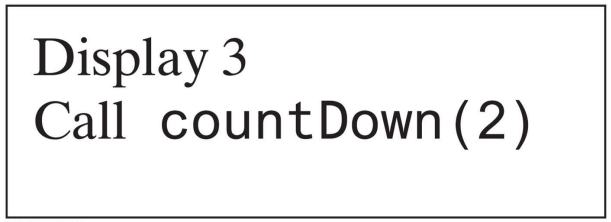
# Programming Tip

- Iterative method contains a loop
- Recursive method calls itself
- Some recursive methods contain a loop and call themselves
  - If the recursive method with loop uses while, make sure you did not mean to use an if statement

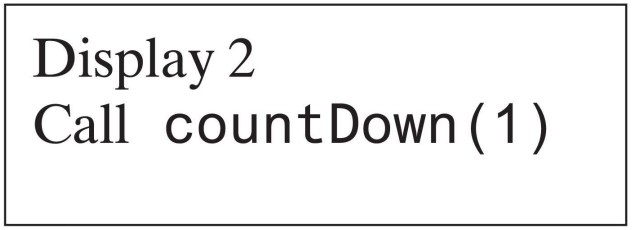
# Tracing a Recursive Method

The effect of the method call `countDown(3)`

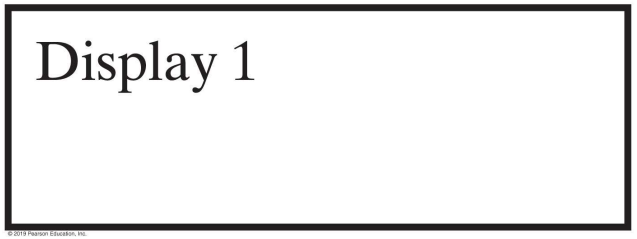
`countDown(3)`




`countDown(2)`

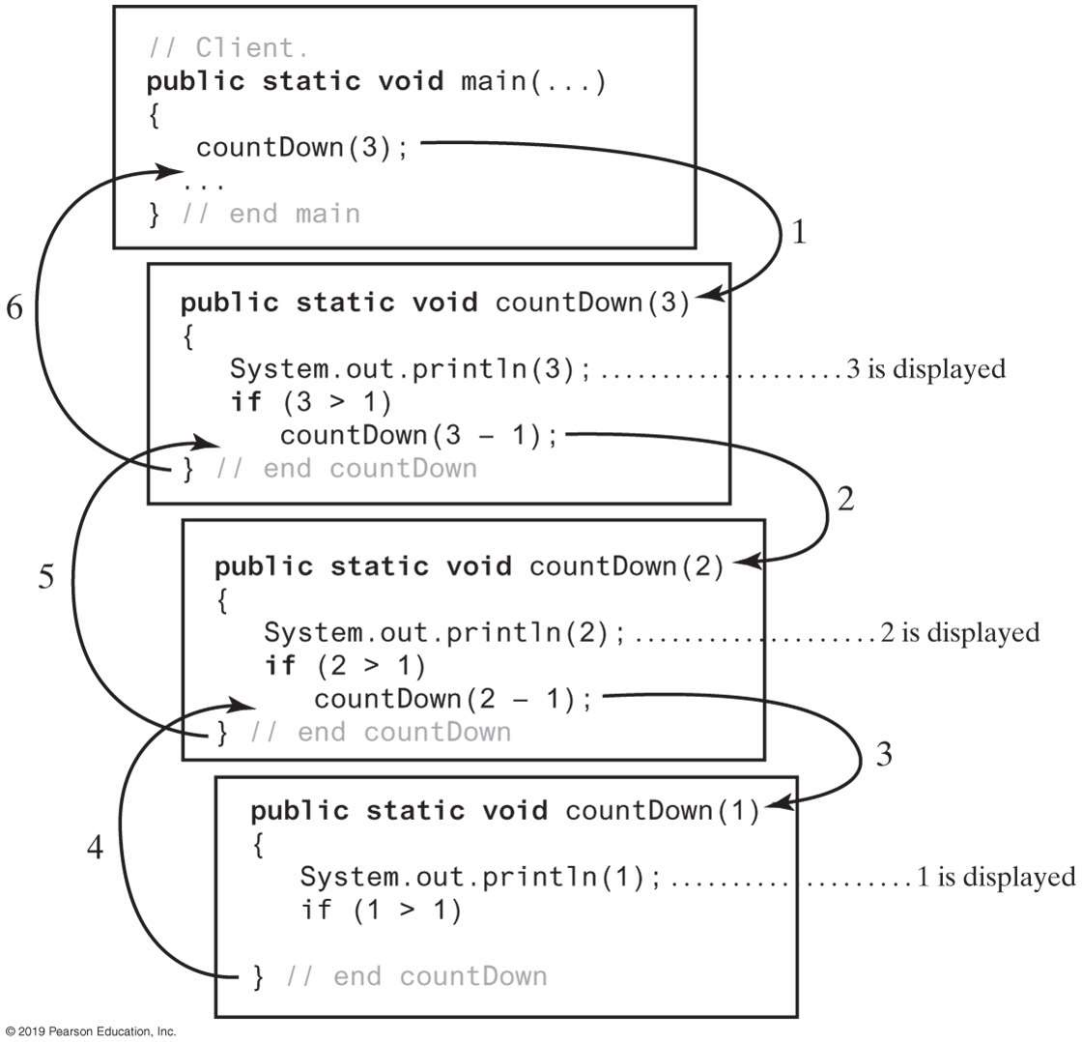



`countDown(1)`




# Tracing a Recursive Method

- Tracing the execution of `countDown(3)`





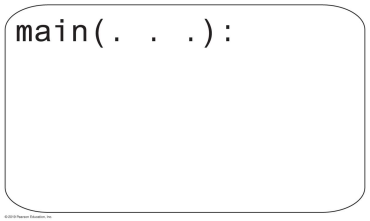

# Stack of Activation Records

- Each call to a method generates an activation record
- Recursive method uses more memory than an iterative method
  - Each recursive call generates an activation record
- If recursive call generates too many activation records, could cause stack overflow

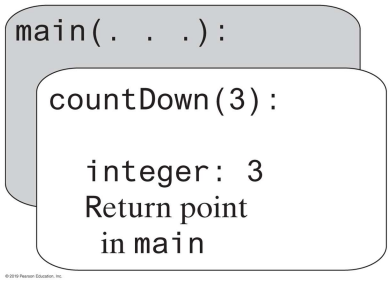
# Stack of Activation Records

The stack of activation records during the execution of the call `countDown(3)`

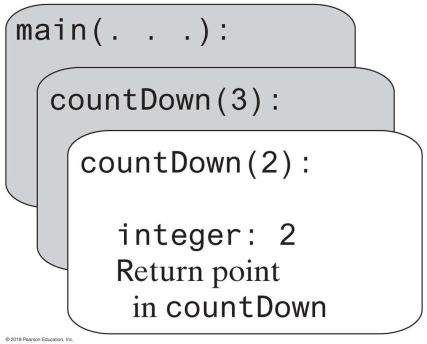
(a)



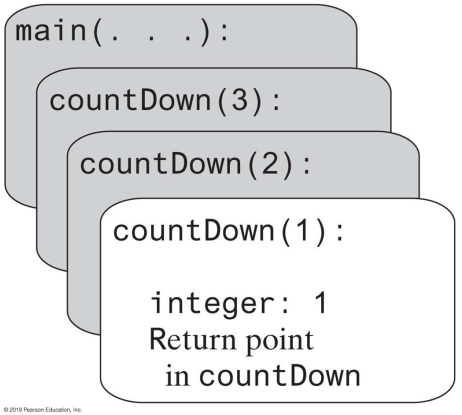

(b)



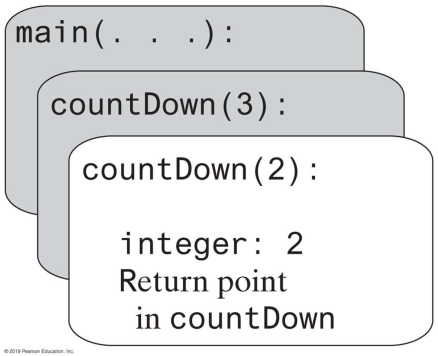

(c)



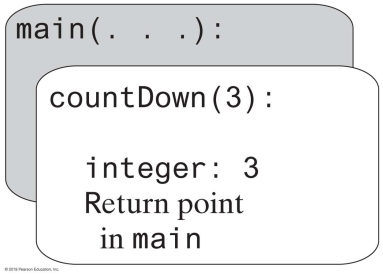

(d)



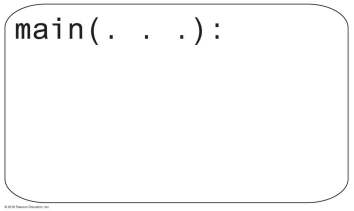

(e)




(f)




(g)




# Recursive Methods That Return a Value

- Recursive method to calculate

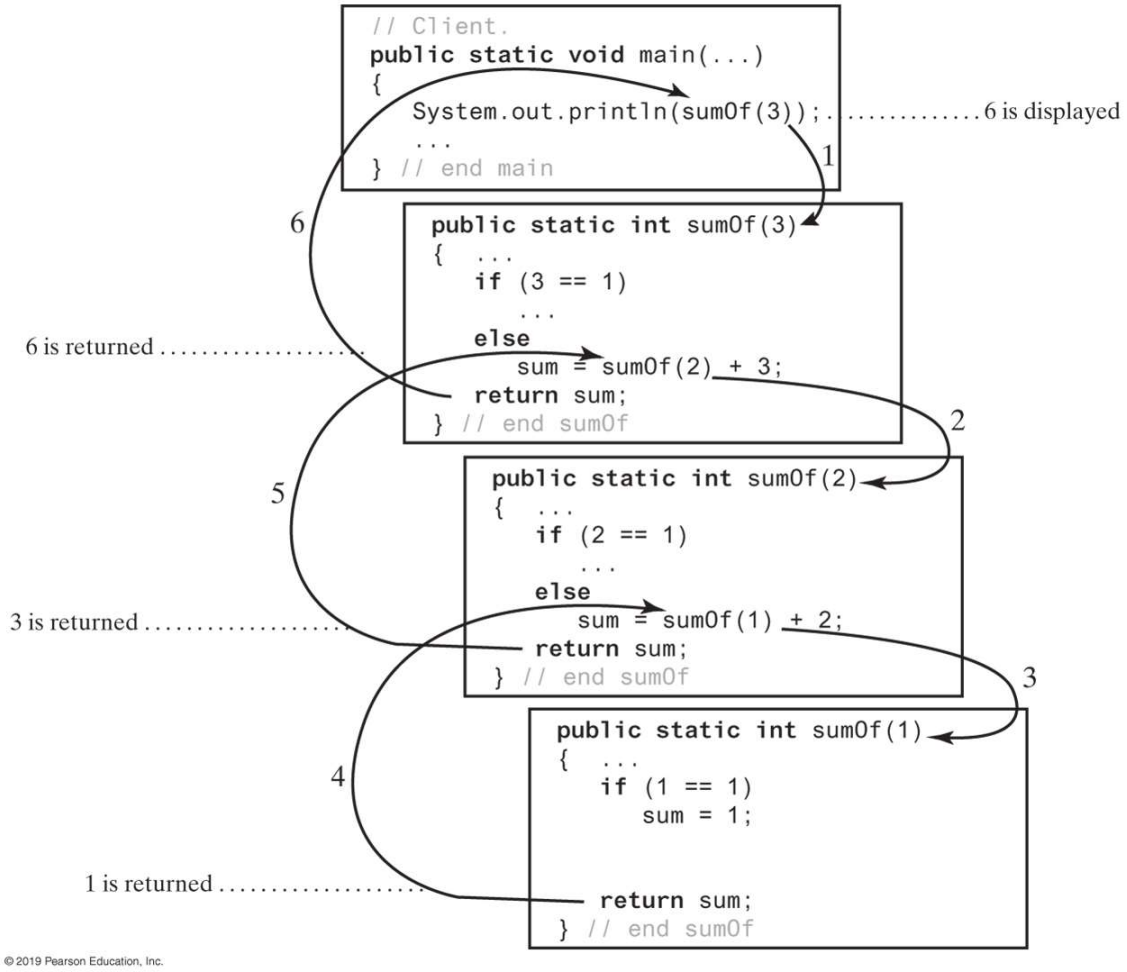
```
/** @param n  An integer > 0.
    @return  The sum 1 + 2 + ... + n. */
public static int sumOf(int n)
{
  int sum;
  if (n == 1)
    sum = 1;              // Base case
  else
    sum = sumOf(n - 1) + n; // Recursive call

  return sum;
} // end sumOf
```

$$\sum_{i=1}^{n} i$$

# Tracing the execution of sumOf(3)

## Tracing the execution of `sumOf(3)`






# Recursively Processing an Array

- Given definition of a recursive method to display array.

```
/** Displays the integers in an array.
   @param array  An array of integers.
   @param first  The index of the first integer displayed.
   @param last   The index of the last integer displayed,
                 0 <= first <= last < array.length. */
public static void displayArray(int[] array, int first, int last)
```

# Recursively Processing an Array

- Starting with `array[first]`

```
public static void displayArray(int array[], int first, int last)
{
  System.out.print(array[first] + " ");
  if (first < last)
    displayArray(array, first + 1, last);
} // end displayArray
```

# Recursively Processing an Array

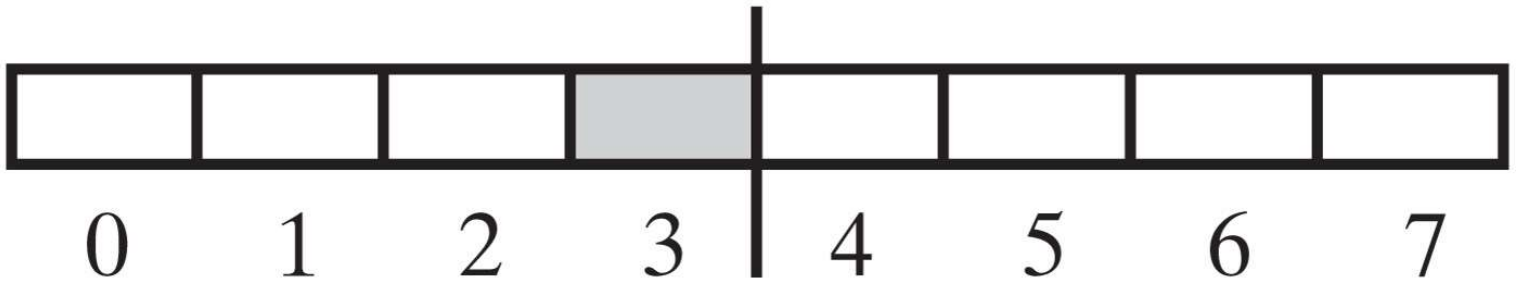
- Starting with `array[last]`

```
public static void displayArray(int array[], int first, int last)
{
  if (first <= last)
  {
    displayArray(array, first, last - 1);
    System.out.print(array[last] + " ");
  } // end if
} // end displayArray
```

# Recursively Processing an Array

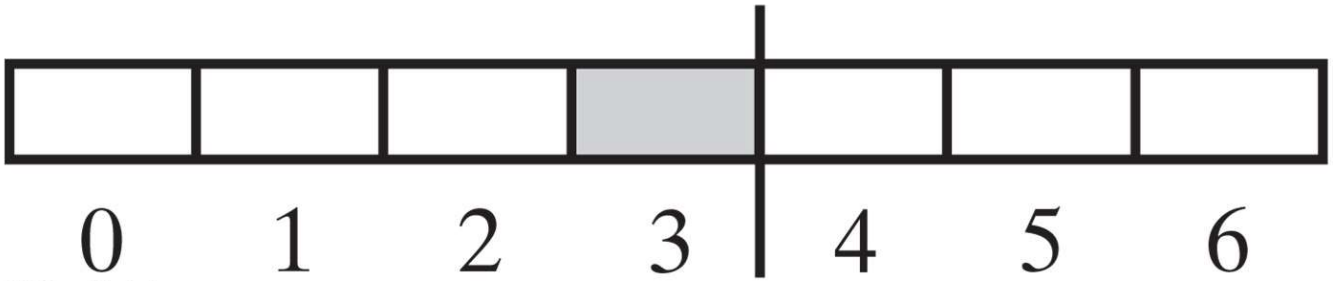
- Two arrays with their middle elements within their left halves

(a)






(b)






# Recursively Processing an Array

- Processing array from middle.

```
public static void displayArray(int array[], int first, int last)
{
  if (first == last)
    System.out.print(array[first] + " ");
  else
  {
    int mid = (first + (last - first) / 2;
    displayArray(array, first, mid);
    displayArray(array, mid + 1, last);
  } // end if
} // end displayArray
```

Why?

# Displaying a Bag

- Recursive method that is part of an implementation of an ADT often is `private`.

```
public void display()
{
  displayArray(0, numberOfEntries - 1);
} // end display

private void displayArray(int first, int last)
{
  System.out.println(bag[first]);
  if (first < last)
    displayArray(first + 1, last);
} // end displayArray
```

# Recursively Processing a Linked Chain

- Display data in first node and recursively display data in rest of chain.

```
public void display()
{
  displayChain(firstNode);
} // end display

private void displayChain(Node nodeOne)
{
  if (nodeOne != null)
  {
    System.out.println(nodeOne.getData()); // Display data in first node
    displayChain(nodeOne.getNextNode());   // Display rest of chain
  } // end if
} // end displayChain
```

# Recursively Processing a Linked Chain

- Displaying a chain backwards. Traversing chain of linked nodes in reverse order easier when done recursively.

```
public void displayBackward()
{
  displayChainBackward(firstNode);
} // end displayBackward

private void displayChainBackward(Node nodeOne)
{
  if (nodeOne != null)
  {
    displayChainBackward(nodeOne.getNextNode());
    System.out.println(nodeOne.getData());
  } // end if
} // end displayChainBackward
```

# Time Efficiency of Recursive Methods

- Using proof by induction, we conclude method is O(n).

```
public static void countDown(int n)
{
  System.out.println(n);
  if (n > 1)
    countDown(n - 1);
} // end countDown
```

# Time Efficiency of Computing $x^n$

- Efficiency of algorithm is O(log *n*)

$$x^n = (x^{n/2})^2 \text{ when } n \text{ is even and positive}$$

$$x^n = x\,(x^{(n-1)/2})^2 \text{ when } n \text{ is odd and positive}$$

$$x^0 = 1$$

# Time Efficiency of Computing $x^n$

```
class ECP {
    /* Function to calculate x raised to the power y */
    static int power(int x, int y)
    {
        if (y == 0)
            return 1;
        else if (y % 2 == 0)
            return power(x, y / 2) * power(x, y / 2);
        else
            return x * power(x, y / 2) * power(x, y / 2);
    }

    /* Program to test function power */
    public static void main(String[] args)
    {
        int x = 2;
        int y = 3;

        System.out.printf("%d", power(x, y));
    }
}
```

# Tail Recursion

- When the last action performed by a recursive method is a recursive call.
- In a tail-recursive method, the last action is a recursive call
- This call performs a repetition that can be done by using iteration.
- Converting a tail-recursive method to an iterative one is usually a straightforward process.

```
public static void countDown(int n)
{   System.out.println(n);
  if (n > 1)
    countDown(n - 1);
} // end countDown
```

# Using a Stack Instead of Recursion

- Converting a recursive method to an iterative one

```
public static void countDown(int integer)
{
  if (integer >= 1)
  {
    System.out.println(integer);
    countDown(integer - 1);
  } // end if
} // end countDown
```

- **An iterative version**

```
public static void countDown(int integer)
{
  while (integer >= 1)
  {
    System.out.println(integer);
    integer = integer – 1;
  } // end while
} // end countDown
```

# Using a Stack Instead of Recursion

- An iterative displayArray to maintain its own stack

```
public void displayArray(int first, int last)
{
  boolean done = false;
  StackInterface<Record> programStack = new LinkedStack<>();
  programStack.push(new Record(first, last));
  while (!done && !programStack.isEmpty())
  {
    Record topRecord = programStack.pop();
    first = topRecord.first;
    last = topRecord.last;

    if (first == last)
      System.out.println(array[first] + " ");
    else
    {
      int mid = first + (last - first) / 2;
      // Note the order of the records pushed onto the stack
      programStack.push(new Record(mid + 1, last));
      programStack.push(new Record(first, mid));
    } // end if
  } // end while
} // end displayArray
```