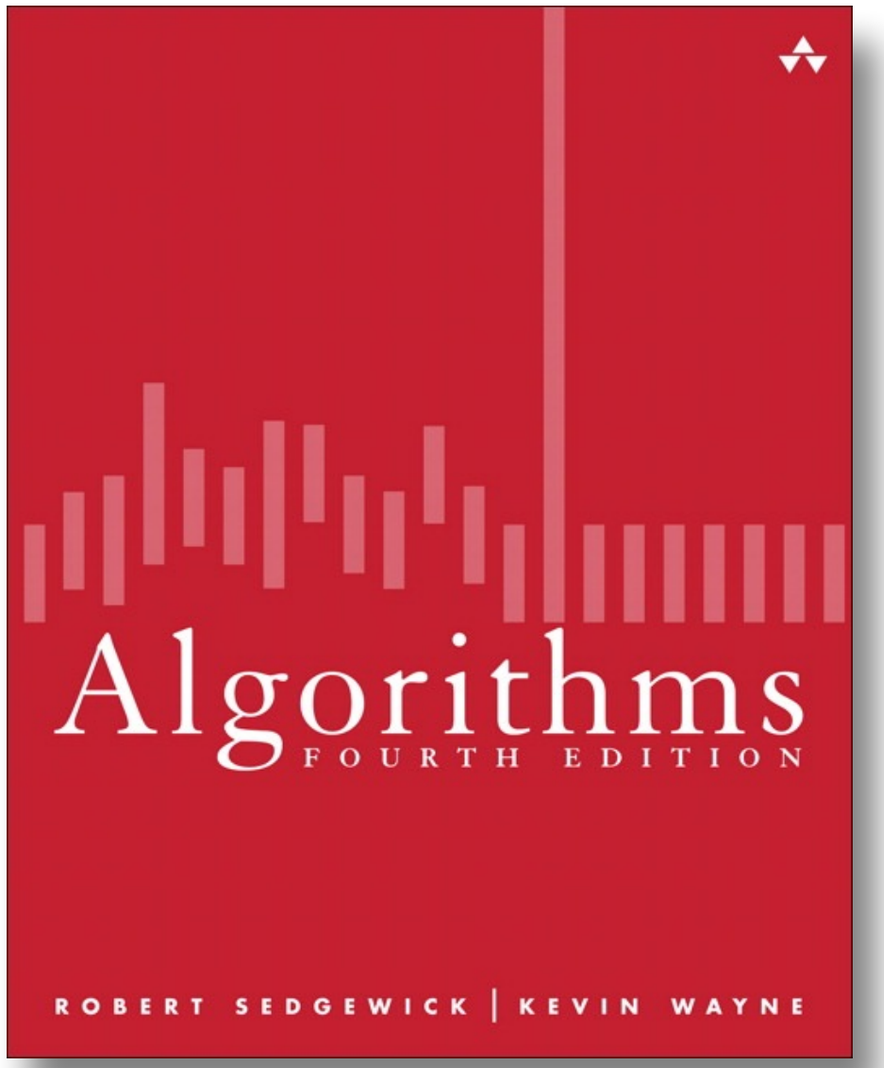
CSC 402

Data Structures I
Lecture 3

Assignment

Modify the Homework2.java application to provide valid logic for the following methods:

- numUnique
- removeDuplicates



Sample Screenshot:

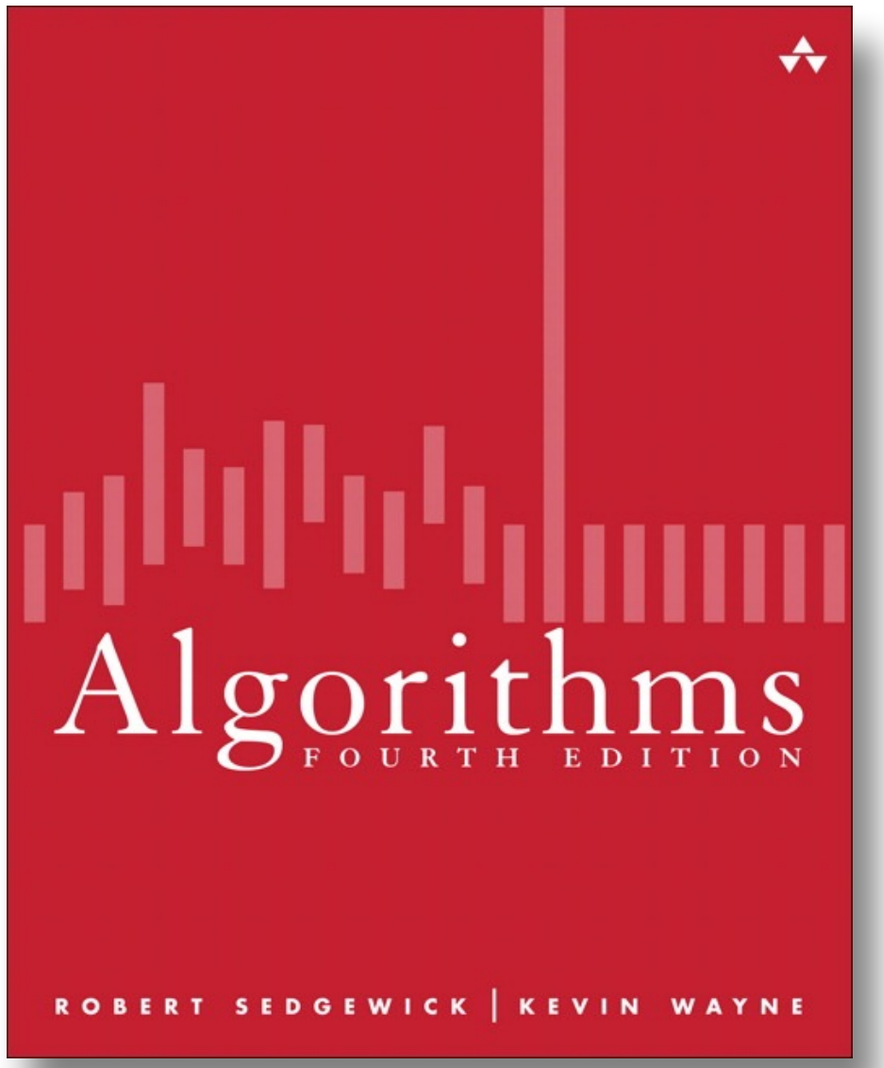
```
<terminated> Homework2 [Java Application] /System/Libı
The uniqueNumbers test was successful.
The removeDuplicates test was successful.
Value [11.0]
Value [22.0]
Value [33.0]
Value [44.0]
Value [55.0]
Value [66.0]
Value [77.0]
Value [88.0]
```

Assignment due next Monday at 11:59 PM

Assignment

Modify the Homework2.java application to provide valid logic for the following methods:

- numUnique
- removeDuplicates



Sample Screenshot:

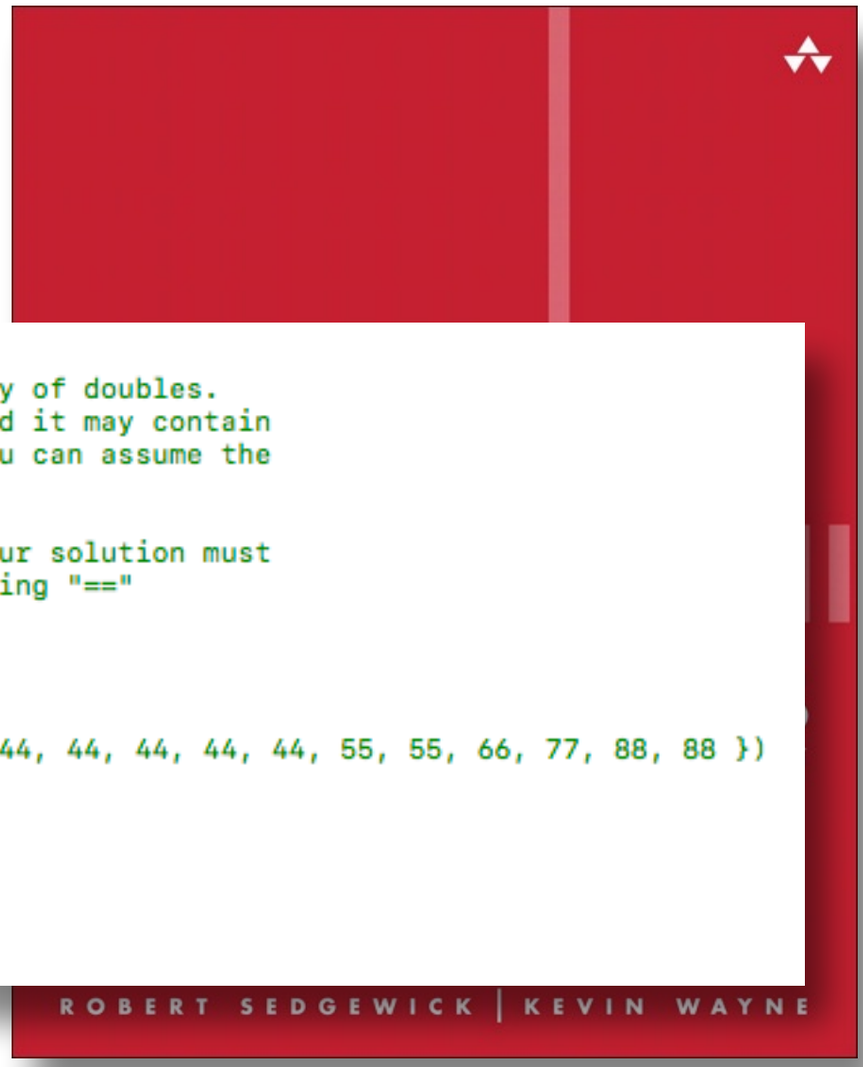
```
<terminated> Homework2 [Java Application] /System/Libı
The uniqueNumbers test was successful.
The removeDuplicates test was successful.
Value [11.0]
Value [22.0]
Value [33.0]
Value [44.0]
Value [55.0]
Value [66.0]
Value [77.0]
Value [88.0]
```

Assignment due next Monday at 11:59 PM

Assignment

Modify the Homework2.java application to provide valid logic for the following methods:

- numUnique
- r




```
/**
 * numUnique returns the number of unique values in an array of doubles.
 * Unlike the previous questions, the array may be empty and it may contain
 * duplicate values. Also unlike the previous questions, you can assume the
 * array is sorted.
 *
 * Your solution must go through the array exactly once. Your solution must
 * not call any other functions. Here are some examples (using "=="
 * informally):
 *
 * <pre>
 *
 *   8 == numUnique(new double[] { 11, 11, 11, 11, 22, 33, 44, 44, 44, 44, 44, 55, 55, 66, 77, 88, 88 })
 *
 * </pre>
 */
public static int numUnique (double[] list) {
    return 0;
}
```

```
Value [44.0]
Value [55.0]
Value [66.0]
Value [77.0]
Value [88.0]
```

Assignment due next Monday at 11:59 PM

Assignment

Modify the Homework2.java application to provide

- n
- r

```
/**
 * numUnique returns the number of unique values in an array of doubles.
 * Unlike the previous questions, the array may be empty and it may contain
 * duplicate values. Also unlike the previous questions, you can assume the
 * array is sorted.
 *
 * Your solution must go through the array exactly once. Your solution must
 * not call any other functions. Here are some examples (using "=="
 * informally):
 *
 * <pre>
 *
 *   8 == numUnique(new double[] { 11, 11, 11, 11, 22, 33, 44, 44, 44, 44, 44, 55, 55, 66, 77, 88, 88 })
 *
 * </pre>
 */
public static int numUnique (double[] list) {
    if (list.length == 0) {
        return 0;
    }
    int result = 1;
    int i = 1;
    while (i < list.length) {
        if (list[i] != list[i - 1]) result++;

        i++;
    }
    return result;
}
```

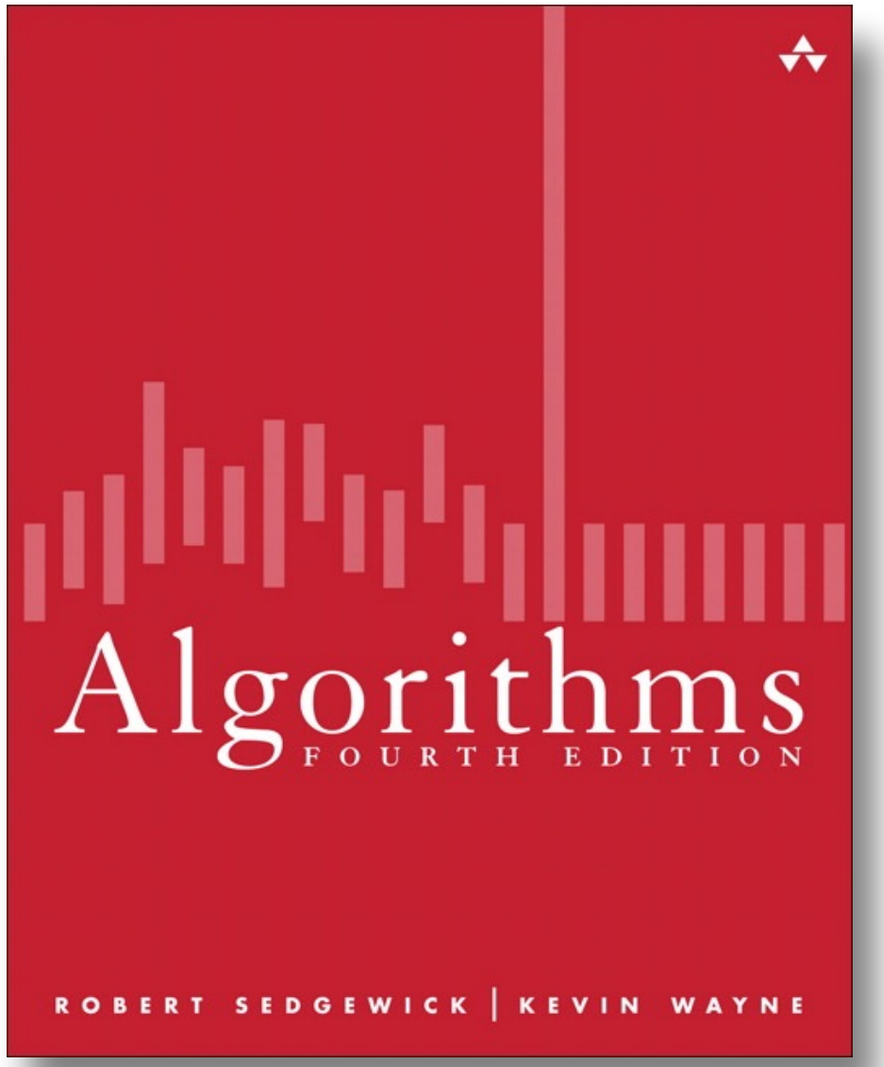
```
Value [88.0]
```

Assignment due next Monday at 11:59 PM

Assignment

Modify the Homework2.java application to provide valid logic for the following methods:

- numUnique
- removeDuplicates



Sample Screenshot:

```
<terminated> Homework2 [Java Application] /System/Libı
The uniqueNumbers test was successful.
The removeDuplicates test was successful.
Value [11.0]
Value [22.0]
Value [33.0]
Value [44.0]
Value [55.0]
Value [66.0]
Value [77.0]
Value [88.0]
```

Assignment due next Monday at 11:59 PM

Assignment

Modify the Homework2.java application to provide valid logic for the following methods:

```
/**
 * removeDuplicates returns a new array containing the unique values in the
 * array. There should not be any extra space in the array --- there should
 * be exactly one space for each unique element (Hint: numUnique tells you
 * how big the array should be). You may assume that the list is sorted, as
 * you did for numUnique.
 *
 * Your solution may call numUnique, but should not call any other
 * functions. After the call to numUnique, you must go through the array
 * exactly one more time. Here are some examples (using "==" informally):
 *
 * <pre>
 *
 * double noDuplicates[] = removeDuplicates (new double[] { 11, 11, 11, 11, 22, 33, 44, 44, 44, 44, 44, 55, 55, 66, 77, 88, 88 });
 *
 * </pre>
 */

public static double[] removeDuplicates (double[] list) {
    return list;
}
```

```
Value [44.0]
Value [55.0]
Value [66.0]
Value [77.0]
Value [88.0]
```

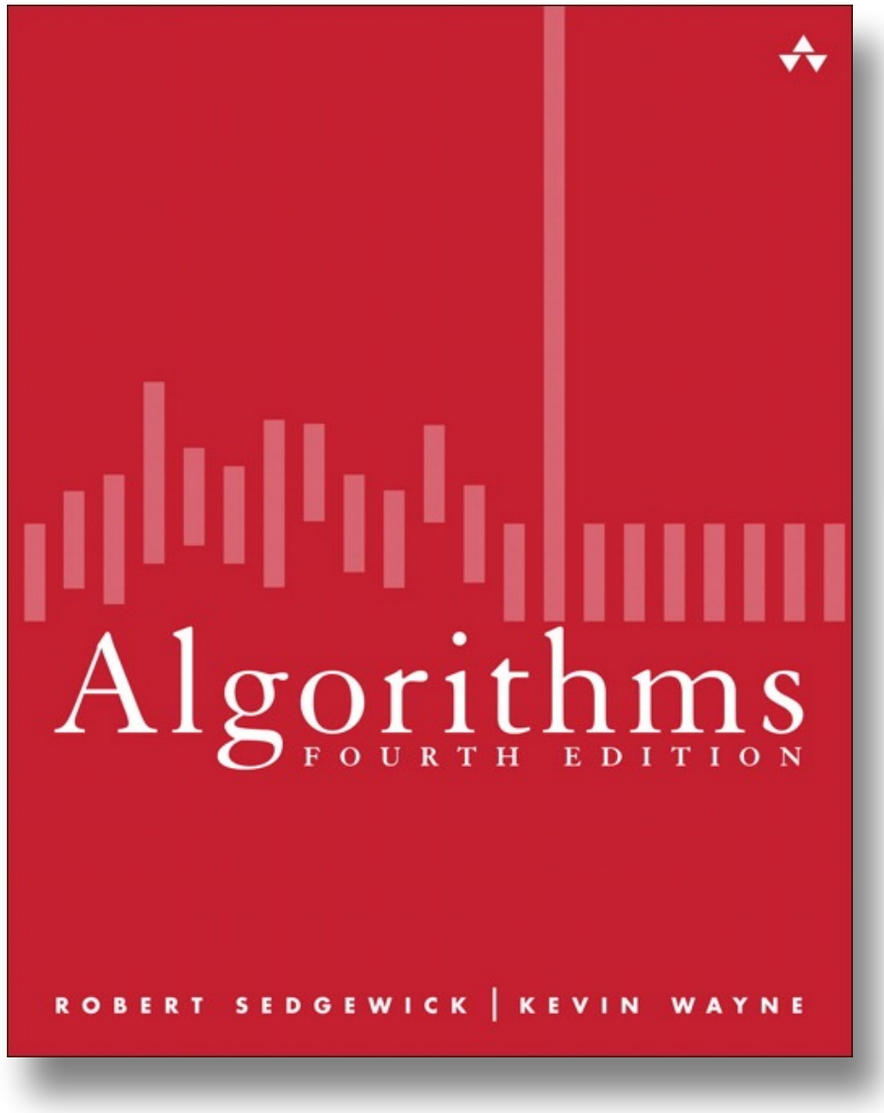
Assignment due next Monday at 11:59 PM

Assignment

```java
/**
 * removeDuplicates returns a new array containing the unique values in the
 * array. There should not be any extra space in the array --- there should
 * be exactly one space for each unique element (Hint: numUnique tells you
 * how big the array should be). You may assume that the list is sorted, as
 * you did for numUnique.
 *
 * Your solution may call numUnique, but should not call any other
 * functions. After the call to numUnique, you must go through the array
 * exactly one more time. Here are some examples (using "==" informally):
 *
 * <pre>
 *
 * double noDuplicates[] = removeDuplicates (new double[] { 11, 11, 11, 11, 22, 33, 44, 44, 44, 44, 44, 55, 55, 66, 77, 88, 88 });
 *
 * </pre>
 */

public static double[] removeDuplicates (double[] list) {
    int numUnique = numUnique (list);
    if (numUnique == list.length) {
        return list;
    }
    // At this point it must be the case that list.length >= 2
    double[] result = new double[numUnique];
    result[0] = list[0];
    int i = 1; // index into list
    int j = 1; // index into result
    while (i < list.length) {
        //System.err.format("i=%2d j=%2d\n", i, j);
        if (result[j - 1] != list[i]) {
            result[j] = list[i];
            j++;
        }
        i++;
    }
    return result;
}
```

Assignment due next Monday at 11:59 PM

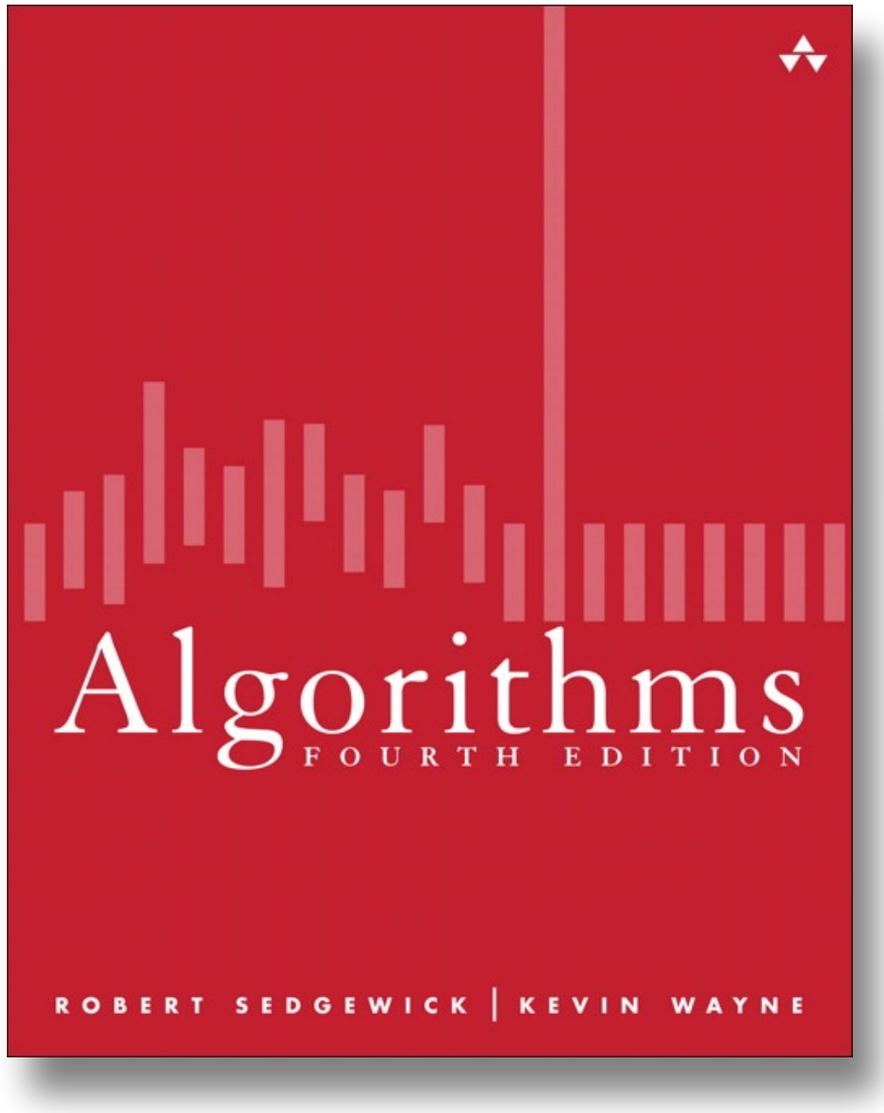
Lecture Overview



Recursion

- Process
- Recursive Helper Methods
- Efficiency

Lecture Overview



Recursion

- Process
- Recursive Helper Methods
- Efficiency

Recursion

- A recursive computation solves a problem by using the solution of the same problem with simpler values
- For recursion to terminate, there must be special cases for the simplest inputs
- To complete our Triangle example, we must handle width <= 0:

```
if (width <= 0)  return 0;
```

- Two key requirements for recursion success:
  - Every recursive call must simplify the computation in some way
  - There must be special cases to handle the simplest computations directly

Other Ways to Compute Triangle Numbers

- The area of a triangle equals the sum:

```
1 + 2 + 3 + ... + width
```

- Using a simple loop:

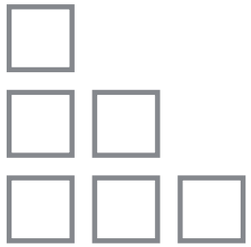
```
double area = 0;
for (int i = 1; i <= width; i++)
  area = area + i;
```

- Using math:

```
1 + 2 + ... + n = n × (n + 1)/2
  => width * (width + 1) / 2
```

Triangle Numbers

- Compute the area of a triangle of width n
- Assume each □ square has an area of 1
- Also called the nth triangle number
- The third triangle number is 6



Process

Triangle Class

```
public class Triangle {
    private int width;
    public Triangle(int aWidth) {
        width = aWidth;
    }

    public int getArea() {
        ...
    }
}
```

Process

Triangle Class

```
public class Triangle {
    private int width;
    public Triangle(int aWidth) {
        width = aWidth;
    }

    public int getArea() {
        ...
    }
}
```

Handling Triangle of Width 1

- The triangle consists of a single square
- Its area is 1

Process

Triangle Class

```
public class Triangle {
    private int width;
    public Triangle(int aWidth) {
        width = aWidth;
    }

    public int getArea() {
        ...
    }
}
```

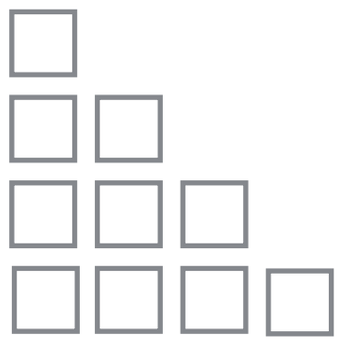
Handling Triangle of Width 1

- The triangle consists of a single square
- Its area is 1
- Add the code to ***getArea*** method for width 1

```
public class Triangle {
    private int width;
    public Triangle(int aWidth) {
        width = aWidth;
    }

    public int getArea() {
        if (width == 1) {
            return 1;
        }
    }
}
```

Handling the General Case

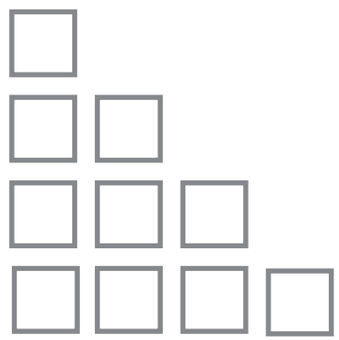
- Assume we know the area of the smaller, colored triangle:



- Area of larger triangle can be calculated as: smallerArea + width

Handling the General Case

- Assume we know the area of the smaller, colored triangle:



- Area of larger triangle can be calculated as: smallerArea + width

- To get the area of the smaller triangle
  - *Make a smaller triangle and ask it for its area:*

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```

```
public class Triangle {
     private int width;
     public Triangle(int aWidth) {
         width = aWidth;
     }

     public int getArea() {
         if (width == 1) {
             return 1;
         }
         Triangle smallerTriangle = new Triangle(width - 1);
         int smallerArea = smallerTriangle.getArea();

         return smallerArea + width;
     }
 }
```

Process

Computing the area of a triangle with width 4

- ***getArea*** method makes a smaller triangle of width 3
- It calls ***getArea*** on that triangle
  - That method makes a smaller triangle of width 2
  - It calls ***getArea*** on that triangle
    - That method makes a smaller triangle of width 1
      - It calls ***getArea*** on that triangle
        - That method returns 1
      - The method returns ***smallerArea + width = 1 + 2 = 3***
  - The method returns ***smallerArea + width = 3 + 3 = 6***
- The method returns ***smallerArea + width = 6 + 4 = 10***

Process

Computing the area of a triangle with width 4

- ***getArea*** method makes a smaller triangle of width 3
- It calls ***getArea*** on that triangle
  - That method makes a smaller triangle of width 2
  - It calls ***getArea*** on that triangle
    - That method makes a smaller triangle of width 1
    - It calls ***getArea*** on that triangle
      - That method returns 1
    - The method returns ***smallerArea + width = 1 + 2 = 3***
  - The method returns ***smallerArea + width = 3 + 3 = 6***
- The method returns ***smallerArea + width = 6 + 4 = 10***

```
public class Triangle {
     private int width;
     public Triangle(int aWidth) {
         width = aWidth;
     }

     public int getArea() {
         if (width == 1) {
             return 1;
         }
         Triangle smallerTriangle = new Triangle(width - 1);
         int smallerArea = smallerTriangle.getArea();

         return smallerArea + width;
     }
}
```

Process

Example

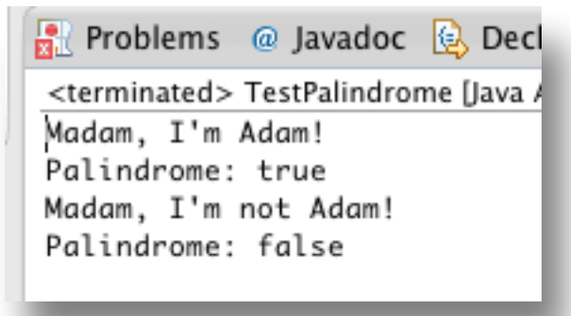
```
package recursion.triangle;

/**
 * A triangular shape composed of stacked unit squares like this:

   []
   [][]
   [][][]

 * . . .
 */
public class Triangle {
    private int width;

    /**
     * Constructs a triangular shape.
     *
     * @param aWidth
     *            the width (and height) of the triangle
     */
    public Triangle(int aWidth) {
     width = aWidth;
    }

    /**
     * Computes the area of the triangle.
     *
     * @return the area
     */
    public int getArea() {
     if (width <= 0) {
           return 0;
     } else if (width == 1) {
           return 1;
     } else {
           Triangle smallerTriangle = new Triangle(width - 1);
           int smallerArea = smallerTriangle.getArea();
           return smallerArea + width;
     }
    }

}
```

```
package recursion.triangle;

public class TriangleTester {

    public static void main(String[] args) {
     Triangle t = new Triangle(10);
     int area = t.getArea();
     System.out.println("Area: " + area);
     System.out.println("Expected: 55");
    }

}
```

Process

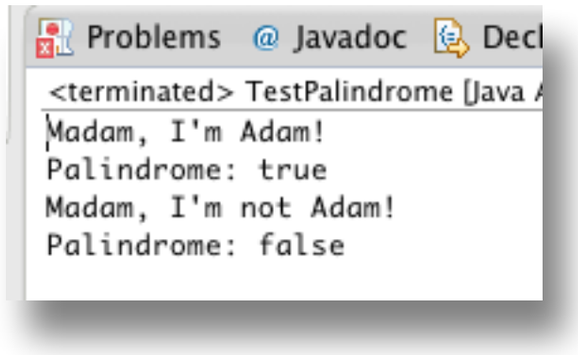
Palindromes - How can we do it recursively

- Remove the first character
- Remove the last character
- Remove both the first and last characters
- Remove a character from the middle
- Cut the string into two halves



Palindromes - Simplify

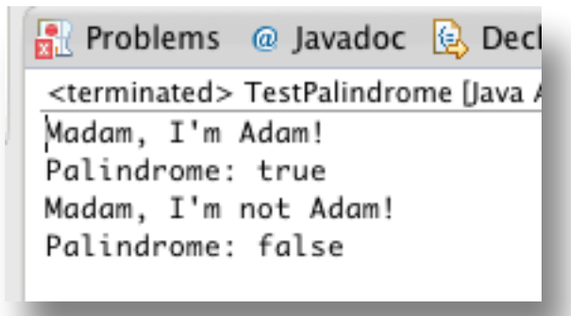
- Most promising simplification: Remove first and last characters
  "adam, I'm Ada" is a palindrome too!
- Thus, a word is a palindrome if
  - The first and last letters match, and
  - Word obtained by removing the first and last letters is a palindrome
- What if first or last character is not a letter? Ignore it
  - If the first and last characters are letters, check whether they match;
    if so, remove both and test shorter string
  - If last character isn't a letter, remove it and test shorter string
  - If first character isn't a letter, remove it and test shorter string



Process

Palindromes - Find solutions to the simplest inputs

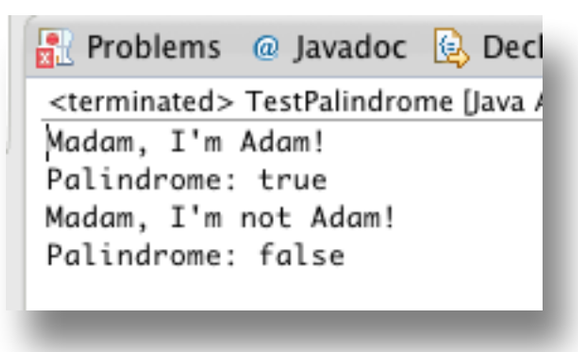
- Strings with two characters
    - No special case required; step two still applies
- Strings with a single character
    - They are palindromes
- The empty string
    - It is a palindrome



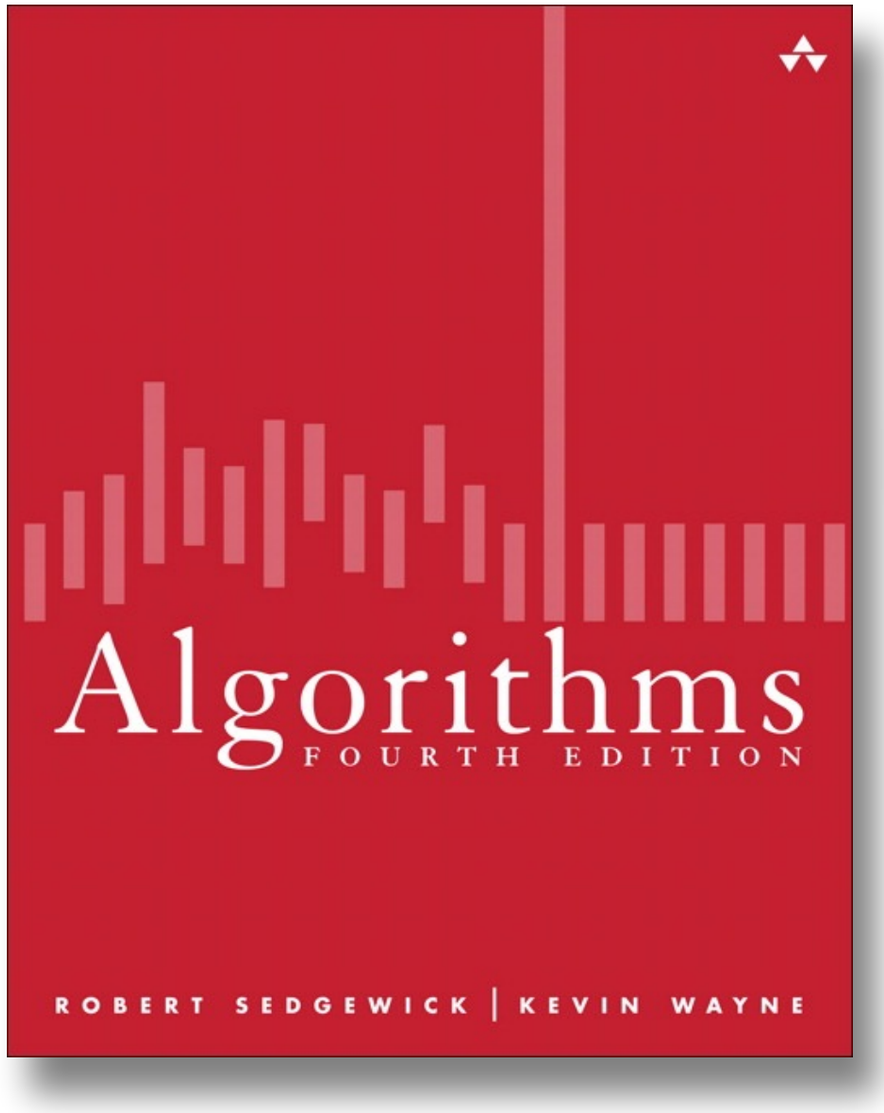
Process

```
package recursion.palindrome;

public class Sentence {

    private String text;

    /**
     * Constructs a sentence.
     * @param aText a string containing all characters of
     * the sentence
     */
    public Sentence(String aText) {
        text = aText;
    }
```

```
    /**
     * Tests whether this sentence is a palindrome.
     * @return true if this sentence is a palindrome, false
     * otherwise
     */
    public boolean isPalindrome() {
        return isPalindromeHelper(0, text.length()-1);
    }

    public boolean isPalindromeHelper(int start, int end) {
        if (start >= end) {
            return true;
        }
        // Get first and last characters, converted to
        // lowercase.
        char first = Character.toLowerCase(text.charAt(start));
        char last = Character.toLowerCase(text.charAt(end));
        if (Character.isLetter(first) && Character.isLetter(last)) {
            // Both are letters.
            if (first == last) {
                // Test substring that doesn't contain the
                // matching letters
                return isPalindromeHelper(start+1, end-1);
            } else {
                return false;
            }
        } else if (!Character.isLetter(last)) {
            // Test substring that doesn't contain the last
            // character.
            return isPalindromeHelper(start, end-1);
        } else {
            // Test substring that doesn't contain the first
            // character
            return isPalindromeHelper(start+1, end);
        }
    }
}
```



Lecture Overview



Recursion

- Process
- Recursive Helper Methods
- Efficiency

Palindromes - Use Recursive Helper Methods

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem
- Consider the palindrome test of previous slide
- It is a bit inefficient to construct new Sentence objects in every step
- Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:
- Then, simply call the helper method with positions that test the entire string:

```
public boolean isPalindromeHelper(int start, int end) {
    if (start >= end) {
        return true;
    }
    // Get first and last characters, converted to
    // lowercase.
    char first = Character.toLowerCase(text.charAt(start));
    char last = Character.toLowerCase(text.charAt(end));
    if (Character.isLetter(first) && Character.isLetter(last)) {
        // Both are letters.
        if (first == last) {
            // Test substring that doesn't contain the
            // matching letters
            return isPalindromeHelper(start+1, end-1);
        } else {
            return false;
        }
    } else if (!Character.isLetter(last)) {
        // Test substring that doesn't contain the last
        // character.
        return isPalindromeHelper(start, end-1);
    } else {
        // Test substring that doesn't contain the first
        // character
        return isPalindromeHelper(start+1, end);
    }
}
```

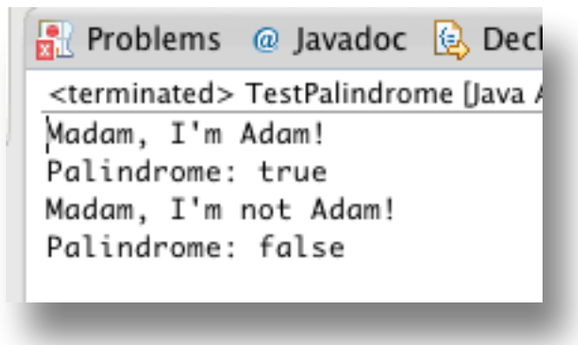
Process

Testing for Palindromes

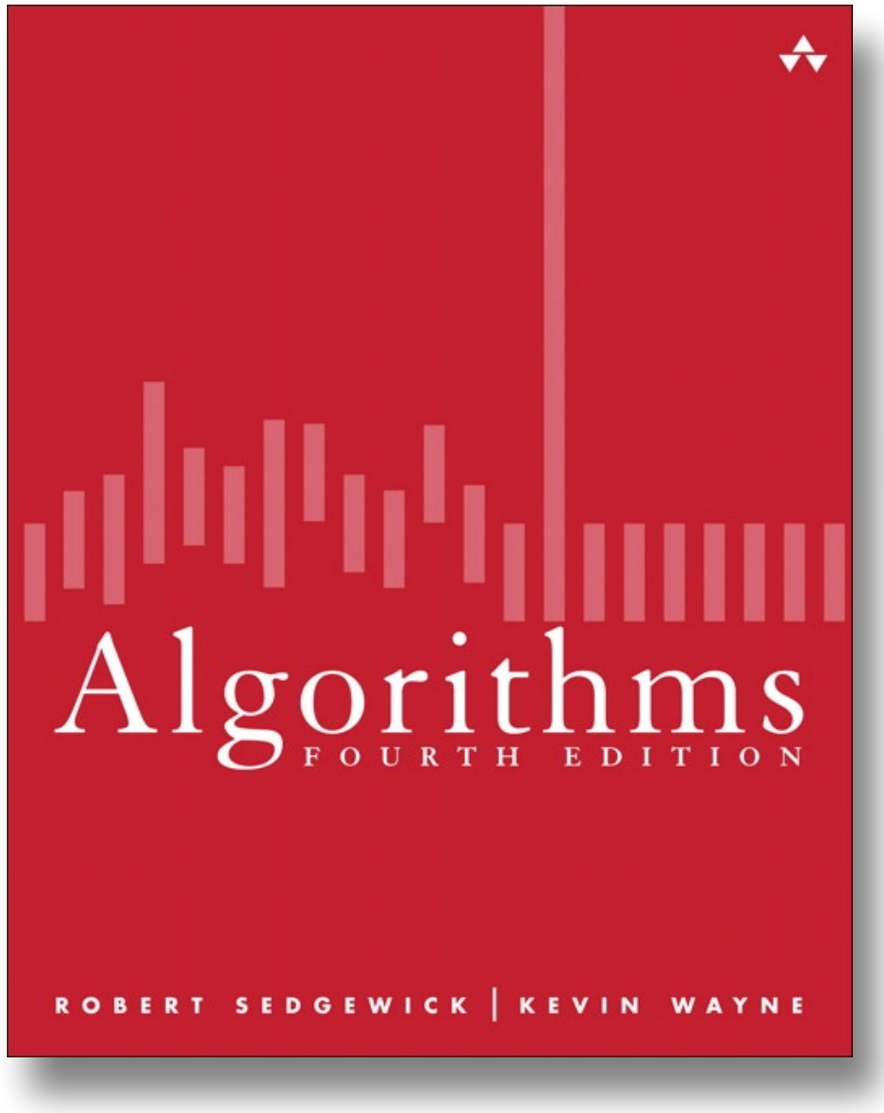
- Palindrome: A string that is equal to itself when you reverse all characters

  Madam Im Adam

```
package recursion.palindrome;

public class TestPalindrome {

    /**
     * @param args
     */
    public static void main(String[] args) {
        String text = "Madam, I'm Adam!";
        System.out.println(text);
        Sentence sentence = new Sentence(text);
        System.out.println("Palindrome: " + sentence.isPalindrome());

        String text2 = "Madam, I'm not Adam!";
        System.out.println(text2);

        Sentence sentence2 = new Sentence(text2);
        System.out.println("Palindrome: " + sentence2.isPalindrome());
    }

}
```



Lecture Overview



Recursion

- Process
- Recursive Helper Methods
- Efficiency

The Efficiency of Recursion

- Occasionally, a recursive solution runs much slower than its iterative counterpart
- In most cases, the recursive solution is only slightly slower
- The iterative isPalindrome performs only slightly better than recursive solution
    - Each recursive method call takes a certain amount of processor time
    - Smart compilers can avoid recursive method calls if they follow simple patterns
- Most compilers don't do that
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution

Efficiency

Iterative *isPalindrome* Method

```
public boolean isPalindrome() {

    int start = 0;
    int end = text.length() - 1;

    while (start < end) {
        char first = Character.toLowerCase(text.charAt(start));
        char last = Character.toLowerCase(text.charAt(end));
        if (Character.isLetter(first) && Character.isLetter(last)) {
            if (first == last) {
                start++;
                end--;
            } else {
                return false;
            }
        }
        if (!Character.isLetter(last)) {
            end--;
        }
        if (!Character.isLetter(first)) {
            start++;
        }
    }
   return true
}
```

Efficiency

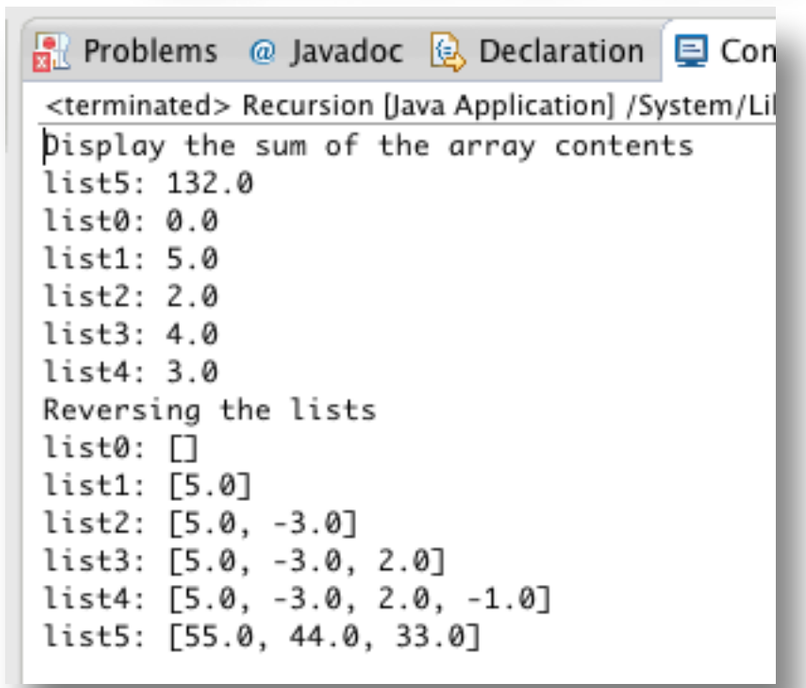
Recursive *isPalindrome* Helper Method

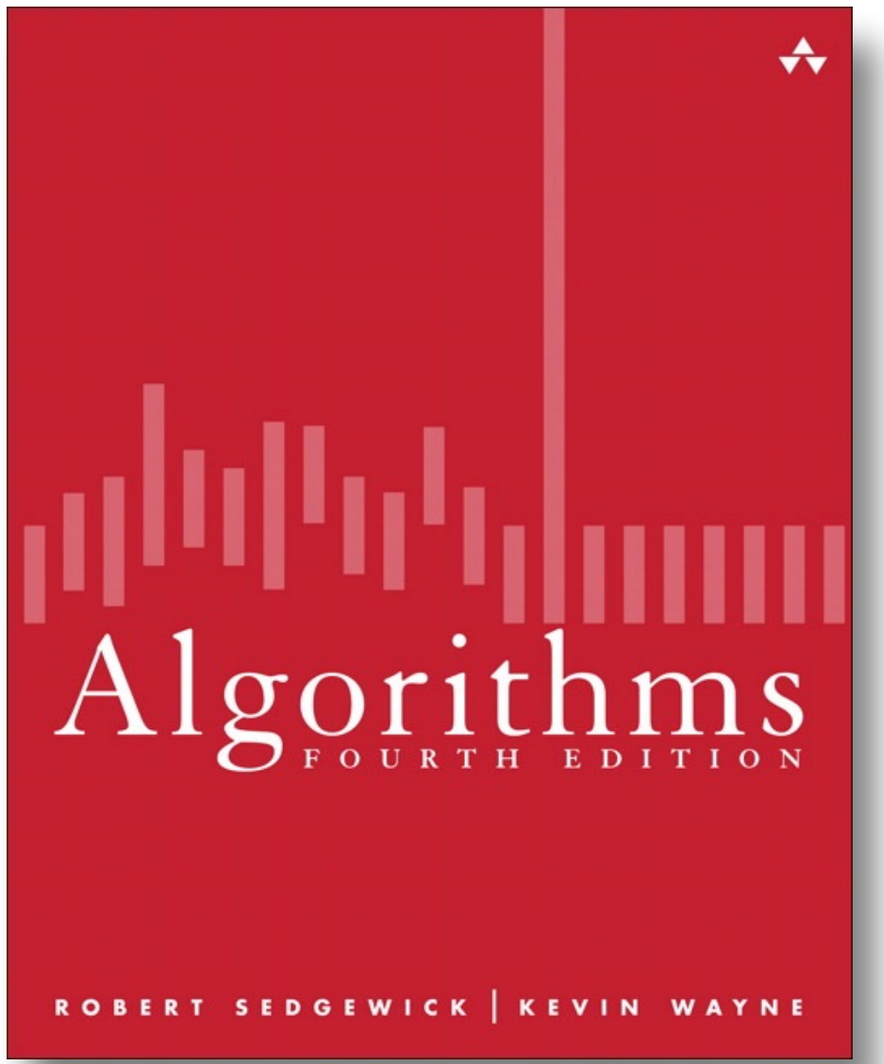
```
public boolean isPalindromeHelper(int start, int end) {
    if (start >= end) {
        return true;
    }
    // Get first and last characters, converted to
    // lowercase.
    char first = Character.toLowerCase(text.charAt(start));
    char last = Character.toLowerCase(text.charAt(end));
    if (Character.isLetter(first) && Character.isLetter(last)) {
        // Both are letters.
        if (first == last) {
            // Test substring that doesn't contain the
            // matching letters
            return isPalindromeHelper(start+1, end-1);
        } else {
            return false;
        }
    } else if (!Character.isLetter(last)) {
        // Test substring that doesn't contain the last
        // character.
        return isPalindromeHelper(start, end-1);
    } else {
        // Test substring that doesn't contain the first
        // character
        return isPalindromeHelper(start+1, end);
    }
}
```

Assignment

Modify the Recursion.java application to provide valid logic for the following methods:

- sum (Modify the existing method)
  - **add** a sumHelper method
- reverse (Modify the existing method)
  - **add** a reverseHelper method





Assignment due next Monday at 11:59 PM

Assignment

Modify the Recursion.java application to provide valid logic for the following methods:

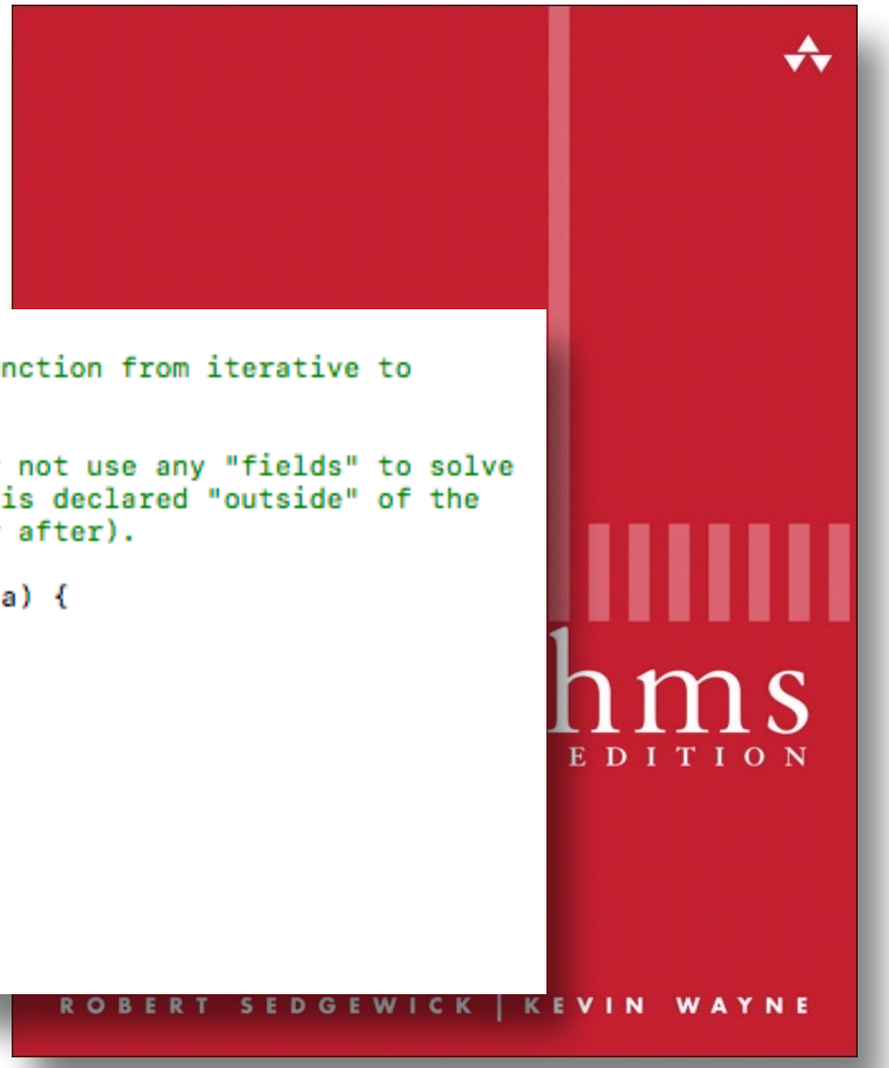
- sum (Modify the existing method)
  - **add** a su
- reverse (Mo
  - **add** a re

```
/**
 * PROBLEM 1: Translate the following sum function from iterative to
 * recursive.
 *
 * You should write a helper method. You may not use any "fields" to solve
 * this problem (a field is a variable that is declared "outside" of the
 * function declaration --- either before or after).
 */
public static double sumIterative (double[] a) {
    double result = 0.0;
    int i = 0;
    while (i < a.length) {
        result = result + a[i];
        i = i + 1;
    }
    return result;
}
public static double sum (double[] a) {
    return 0; // TODO
}
```

Problems @ Ja

<terminated> Recur

```
Display the sum
list5: 132.0
list0: 0.0
list1: 5.0
list2: 2.0
list3: 4.0
list4: 3.0
Reversing the li
list0: []
list1: [5.0]
list2: [5.0, -3.0]
list3: [5.0, -3.0, 2.0]
list4: [5.0, -3.0, 2.0, -1.0]
list5: [55.0, 44.0, 33.0]
```



Assignment due next Monday at 11:59 PM

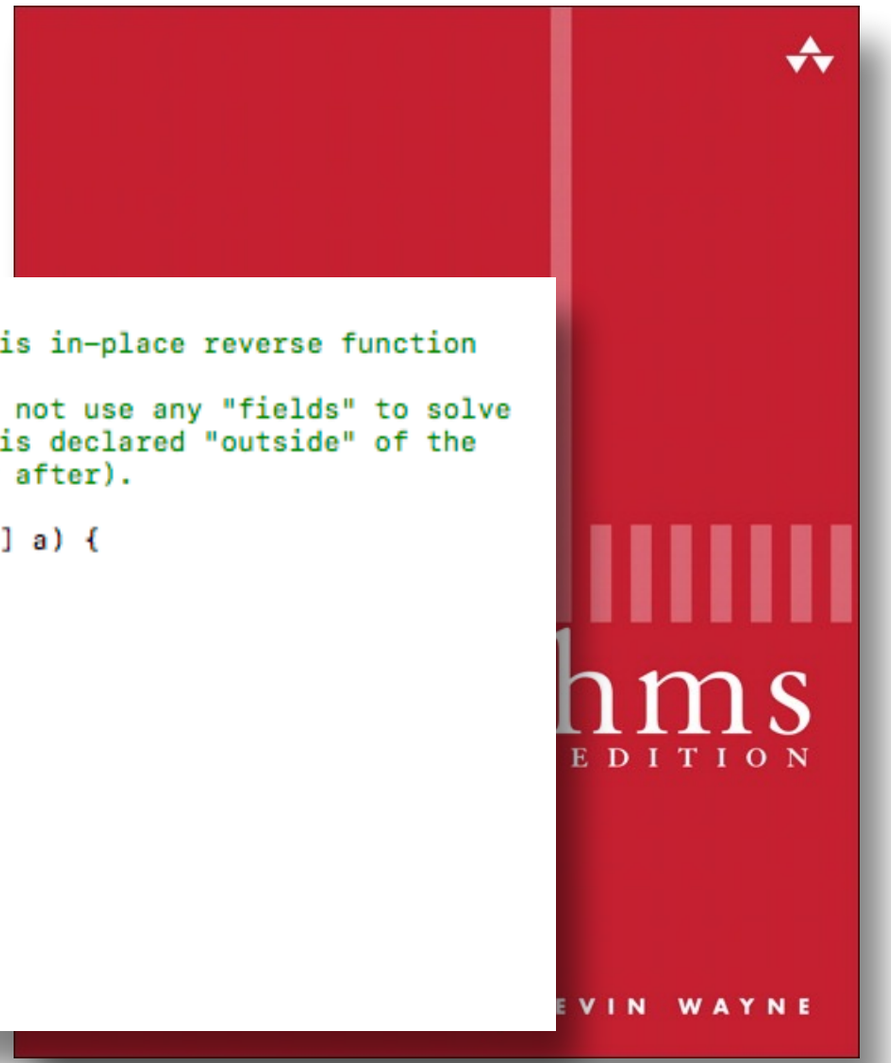
Assignment

Modify the Recursion.java application to provide valid logic for the following methods:

- sum (Modify the existing method)
  - **add** a su
- reverse (Mo
  - **add** a re

```
/**
 * PROBLEM 2: Do the same translation for this in-place reverse function
 *
 * You should write a helper method. You may not use any "fields" to solve
 * this problem (a field is a variable that is declared "outside" of the
 * function declaration --- either before or after).
 */
public static void reverseIterative (double[] a) {
    int hi = a.length - 1;
    int lo = 0;
    while (lo < hi) {
        double loVal = a[lo];
        double hiVal = a[hi];
        a[hi] = loVal;
        a[lo] = hiVal;
        lo = lo + 1;
        hi = hi - 1;
    }
}
public static void reverse (double[] a) {
    // TODO
}
```

```
Problems @ Ja
<terminated> Recu
Display the sum
list5: 132.0
list0: 0.0
list1: 5.0
list2: 2.0
list3: 4.0
list4: 3.0
Reversing the l
list0: []
list1: [5.0]
list2: [5.0, -3.0]
list3: [5.0, -3.0, 2.0]
list4: [5.0, -3.0, 2.0, -1.0]
list5: [55.0, 44.0, 33.0]
```



Assignment due next Monday at 11:59 PM