CSC 402

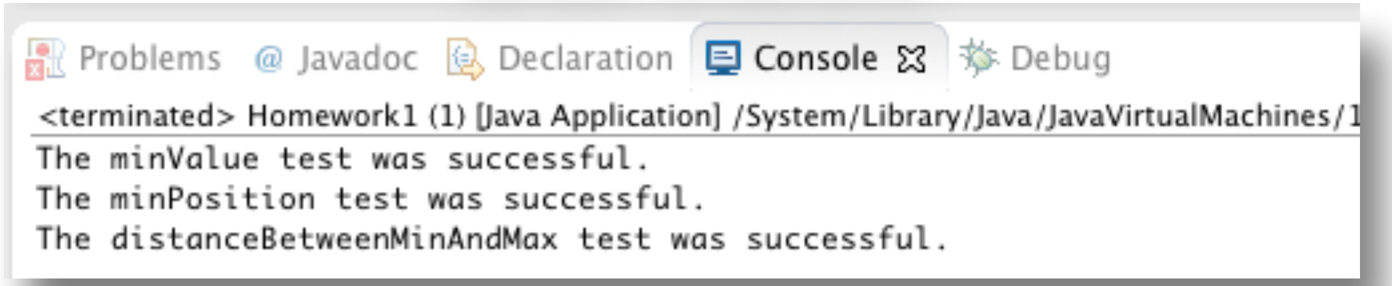
Data Structures I
Lecture 2

Assignment

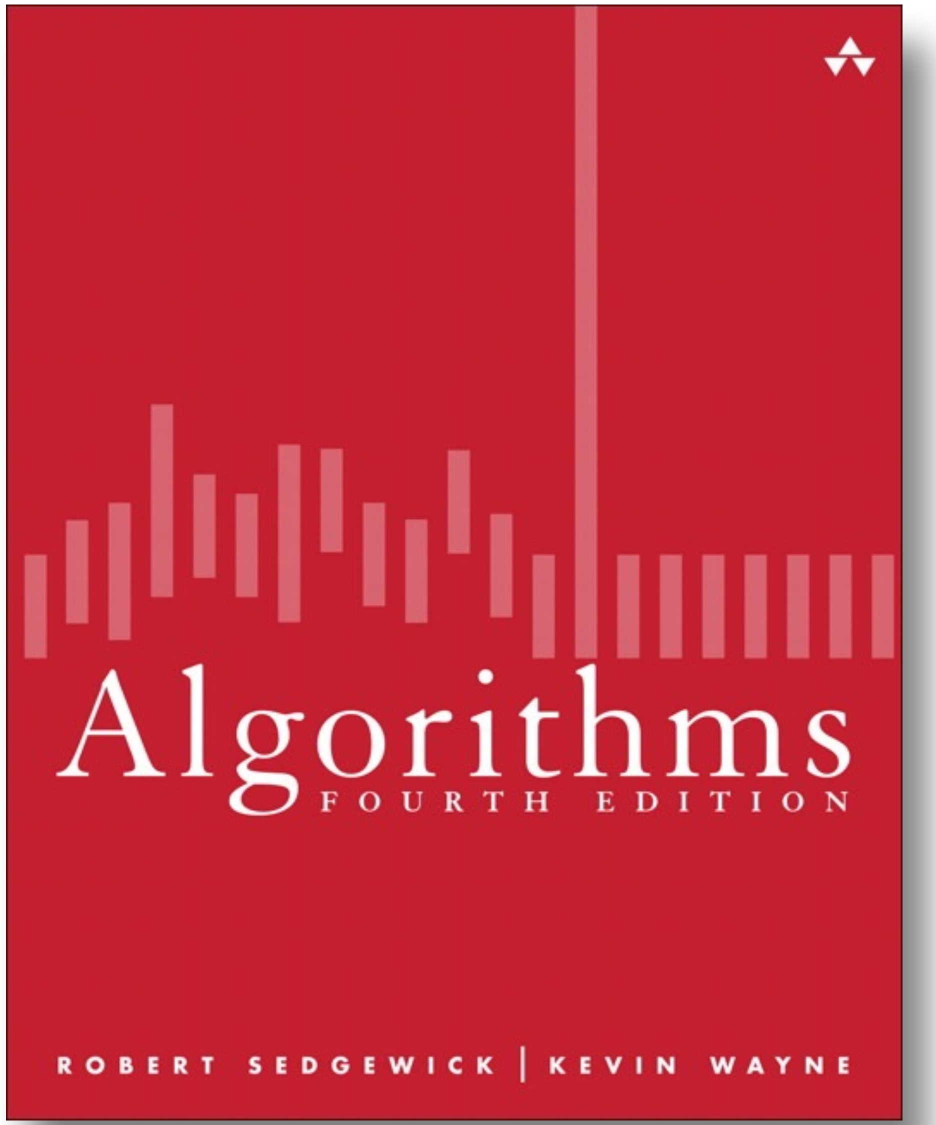
Modify the Homework1.java application to provide valid logic for the following methods:

- minValue
- minPosition
- distanceBetweenMinAndMax

Sample Screenshot:



```
<terminated> Homework1 (1) [Java Application] /System/Library/Java/JavaVirtualMachines/1
The minValue test was successful.
The minPosition test was successful.
The distanceBetweenMinAndMax test was successful.
```



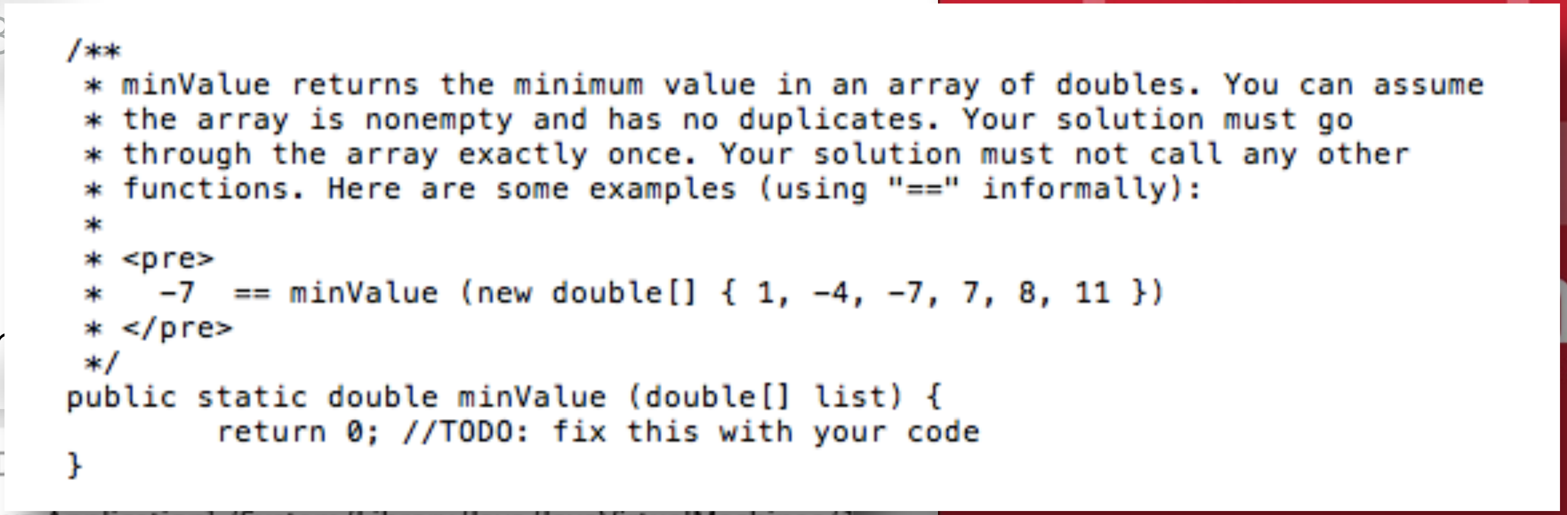
Assignment due next Monday at 11:59 PM

Assignment

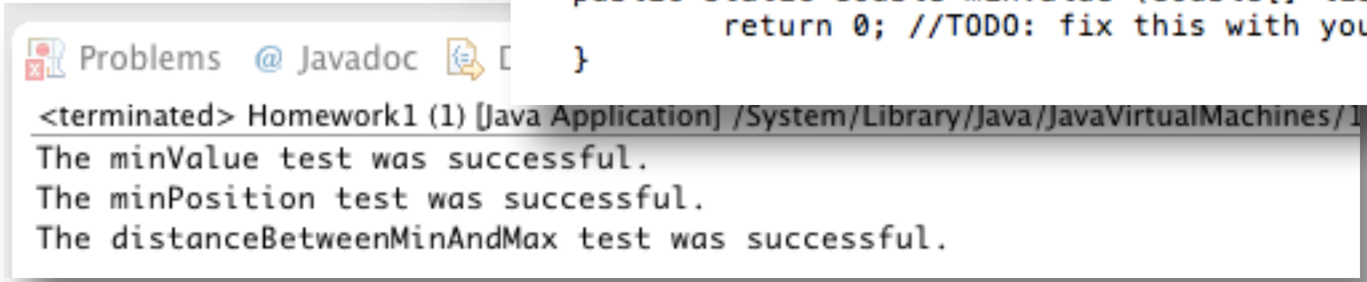
Modify the Homework1.java application to provide valid logic for the following methods:

- minValue
- minPosition
- distanceB

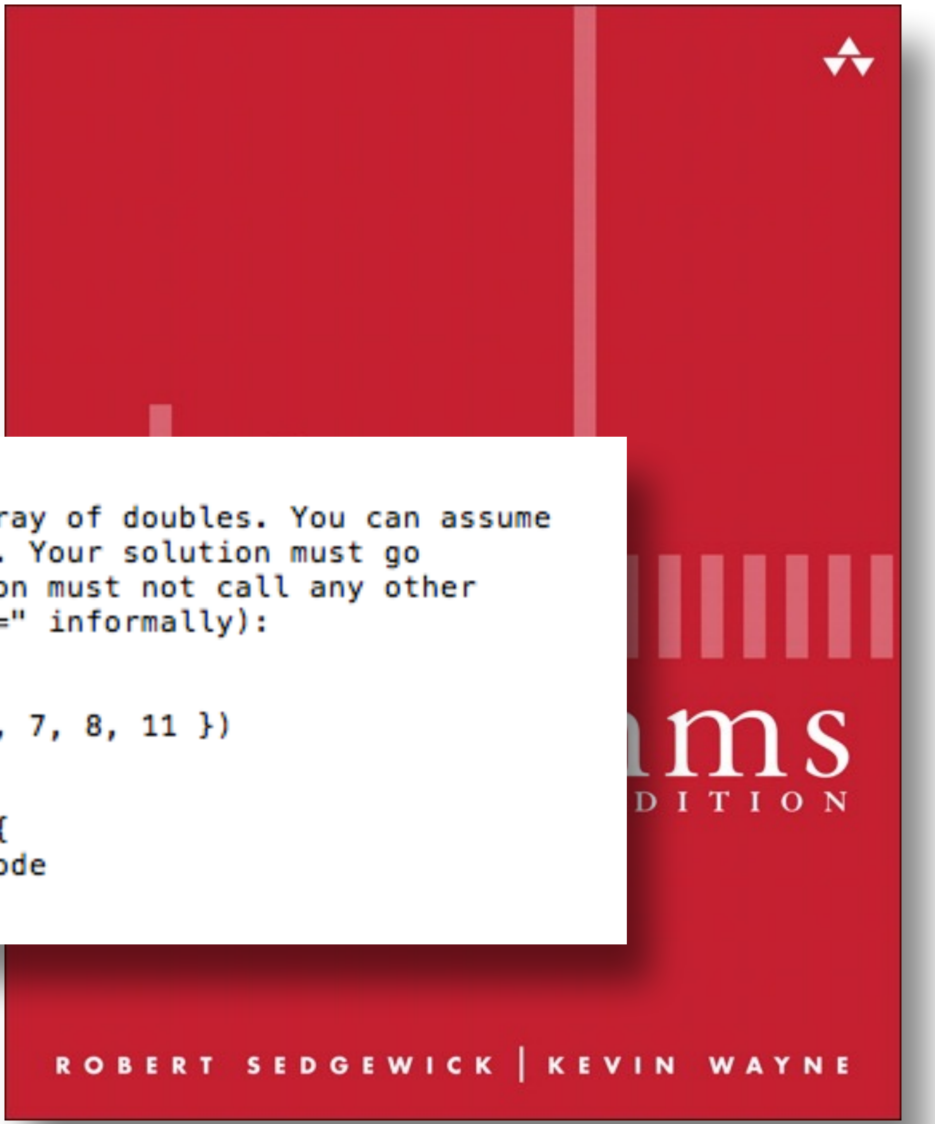
Sar



```
/**
 * minValue returns the minimum value in an array of doubles. You can assume
 * the array is nonempty and has no duplicates. Your solution must go
 * through the array exactly once. Your solution must not call any other
 * functions. Here are some examples (using "==" informally):
 *
 * <pre>
 *   -7  == minValue (new double[] { 1, -4, -7, 7, 8, 11 })
 * </pre>
 */
public static double minValue (double[] list) {
        return 0; //TODO: fix this with your code
}
```



```
<terminated> Homework1 (1) [Java Application] /System/Library/Java/JavaVirtualMachines/1
The minValue test was successful.
The minPosition test was successful.
The distanceBetweenMinAndMax test was successful.
```



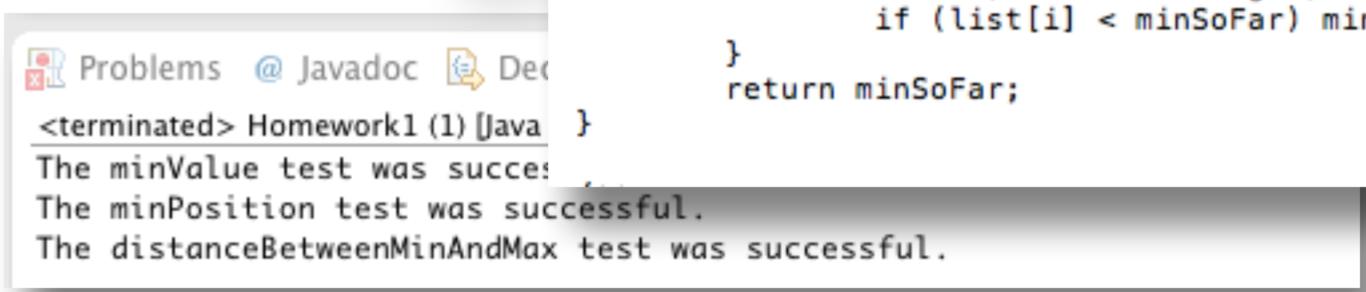
Assignment due next Monday at 11:59 PM

Assignment

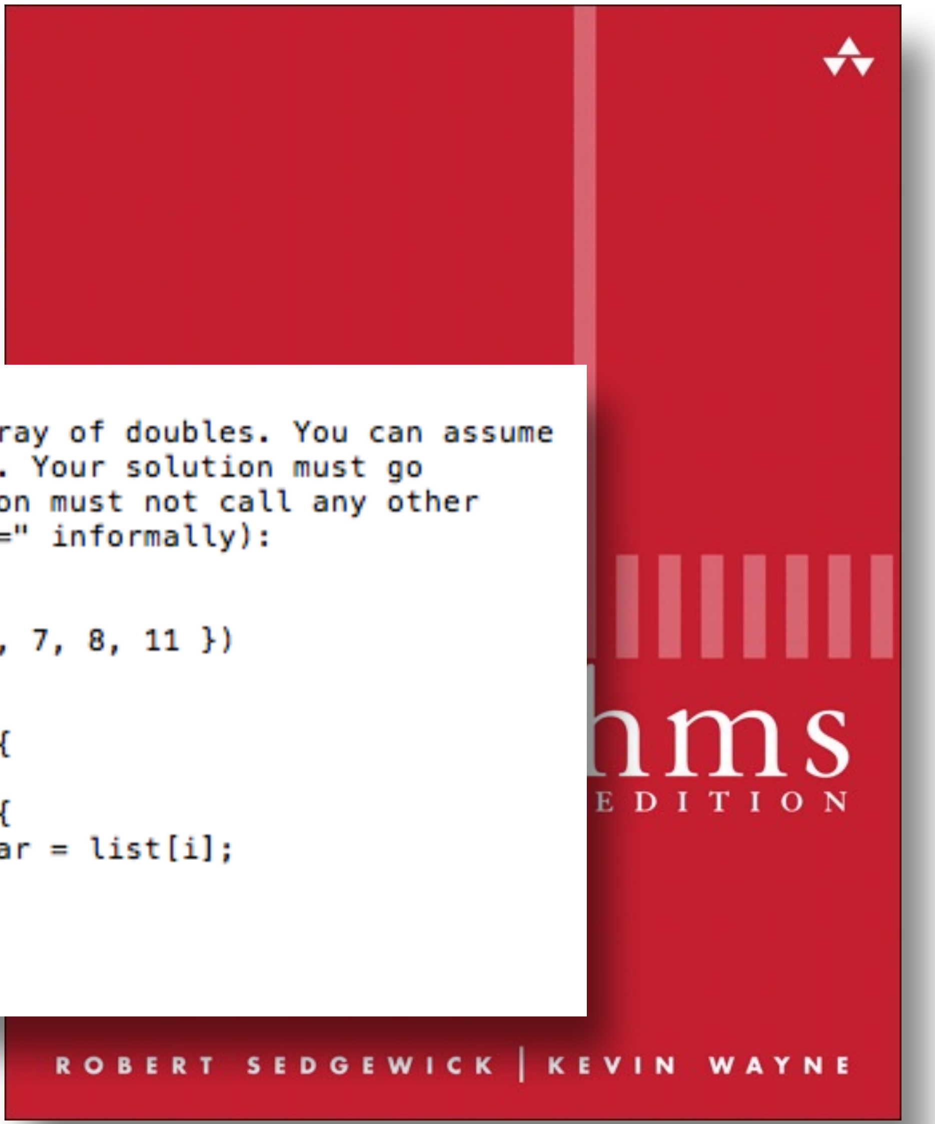
Modify the Homework1.java application to provide valid logic for the following methods:

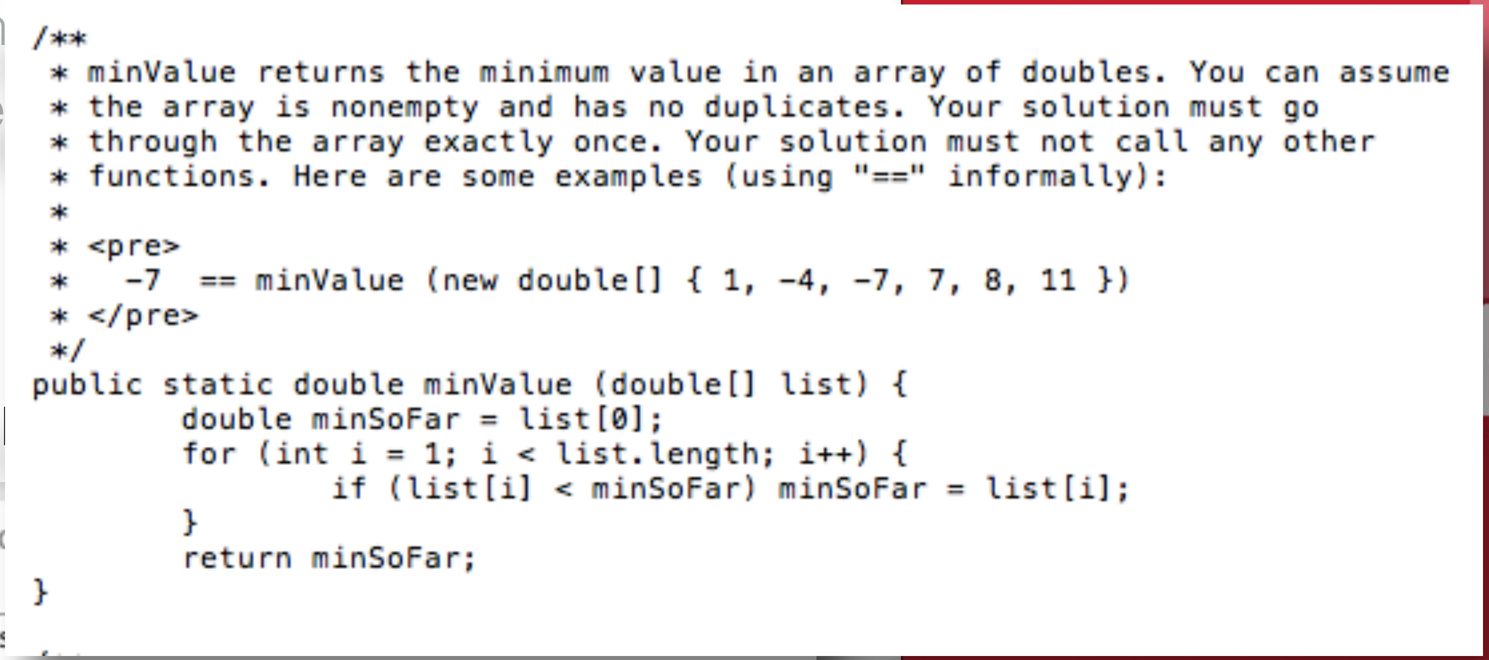
- minValue
- minPosition
- distanceBe

Samp



```
Problems  @ Javadoc  Dec
<terminated> Homework1 (1) [Java
The minValue test was succe
The minPosition test was successful.
The distanceBetweenMinAndMax test was successful.
```





```
/**
 * minValue returns the minimum value in an array of doubles. You can assume
 * the array is nonempty and has no duplicates. Your solution must go
 * through the array exactly once. Your solution must not call any other
 * functions. Here are some examples (using "==" informally):
 *
 * <pre>
 *   -7  == minValue (new double[] { 1, -4, -7, 7, 8, 11 })
 * </pre>
 */
public static double minValue (double[] list) {
        double minSoFar = list[0];
        for (int i = 1; i < list.length; i++) {
                if (list[i] < minSoFar) minSoFar = list[i];
        }
        return minSoFar;
}
```

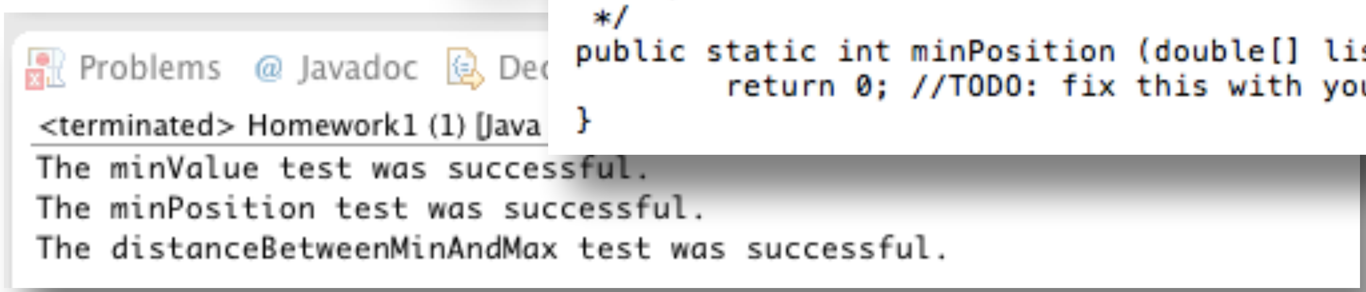
Assignment due next Monday at 11:59 PM

Assignment

Modify the Homework1.java application to provide valid logic for the following methods:

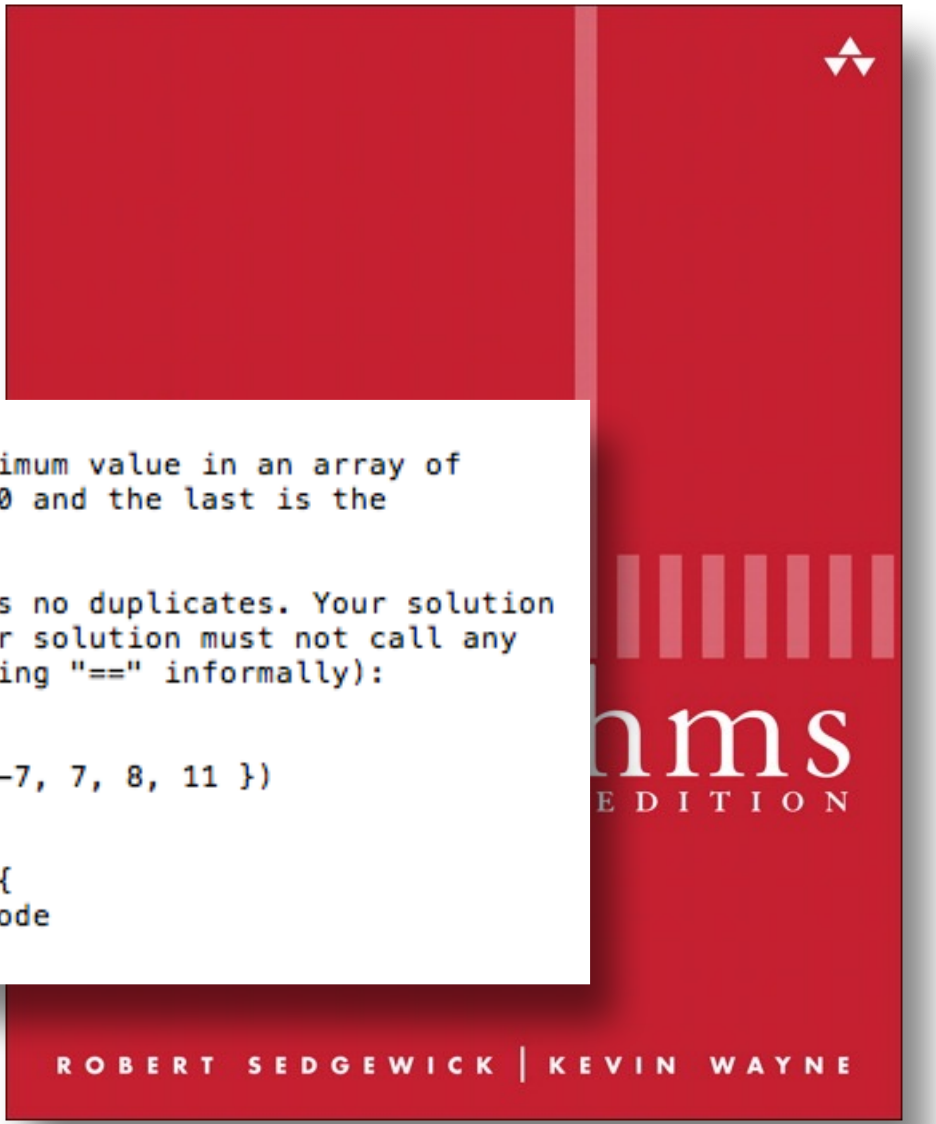
- minValue
- minPosition
- distanceBe

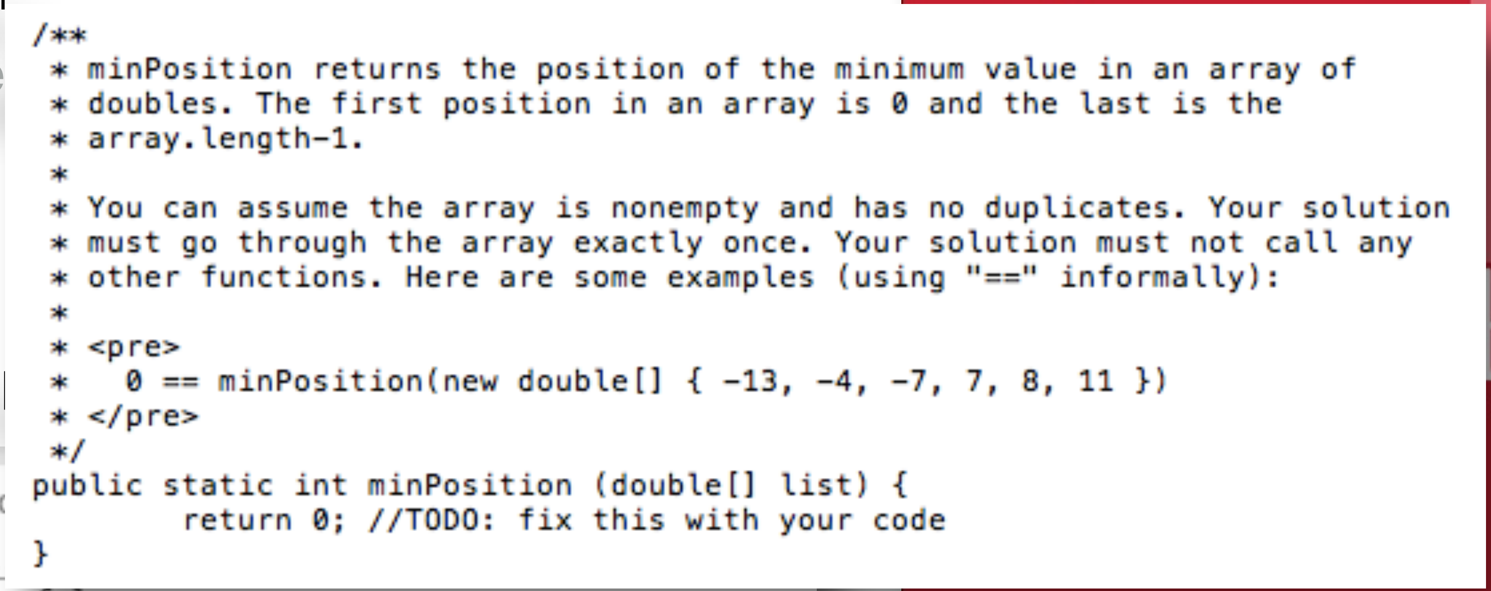
Samp



Problems @ Javadoc Dec

```
<terminated> Homework1 (1) [Java
The minValue test was successful.
The minPosition test was successful.
The distanceBetweenMinAndMax test was successful.
```





```
/**
 * minPosition returns the position of the minimum value in an array of
 * doubles. The first position in an array is 0 and the last is the
 * array.length-1.
 *
 * You can assume the array is nonempty and has no duplicates. Your solution
 * must go through the array exactly once. Your solution must not call any
 * other functions. Here are some examples (using "==" informally):
 *
 * <pre>
 *   0 == minPosition(new double[] { -13, -4, -7, 7, 8, 11 })
 * </pre>
 */
public static int minPosition (double[] list) {
        return 0; //TODO: fix this with your code
}
```

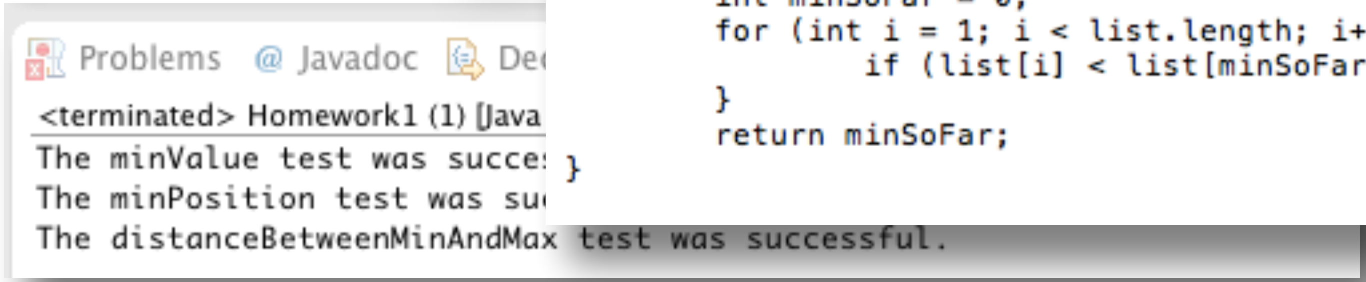
Assignment due next Monday at 11:59 PM

Assignment

Modify the Homework1.java application to provide valid logic for the following methods:

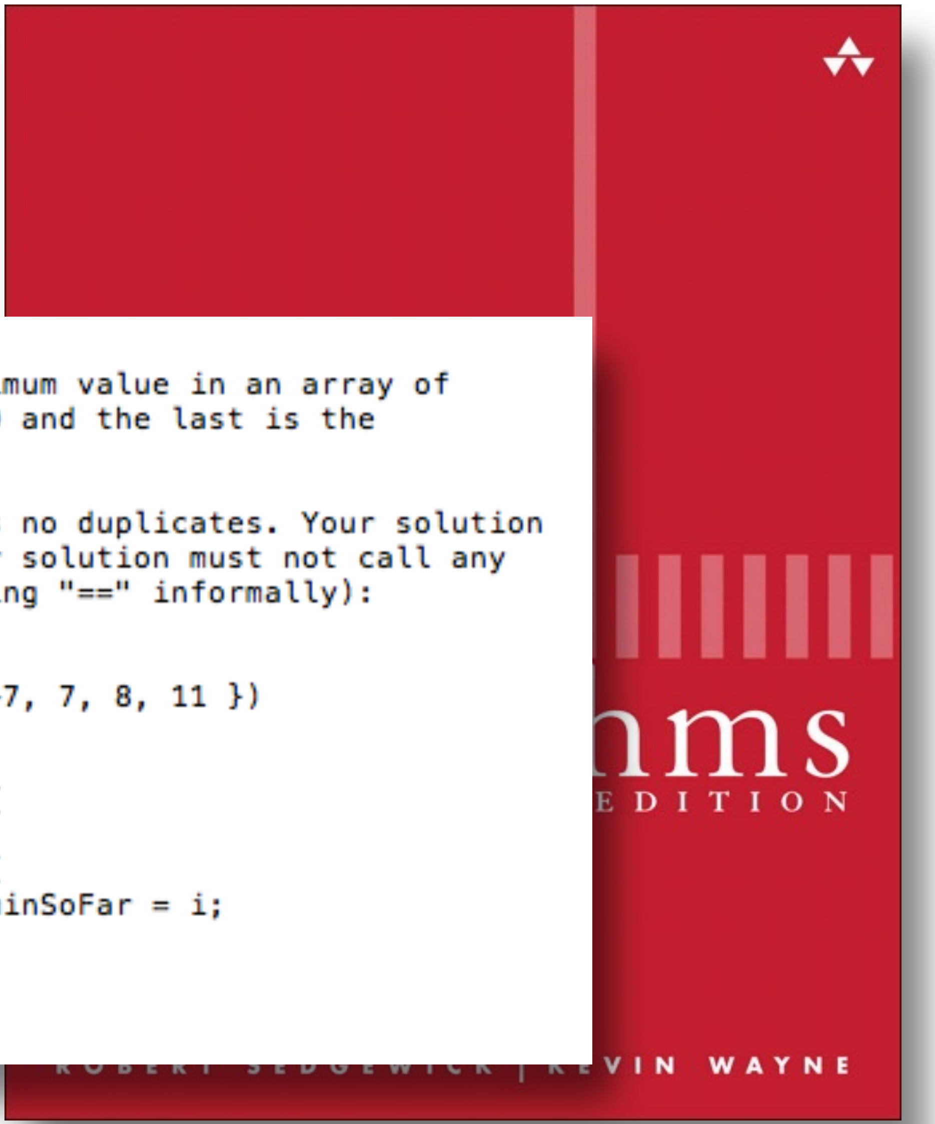
- minValue
- minPosition
- distanceBe

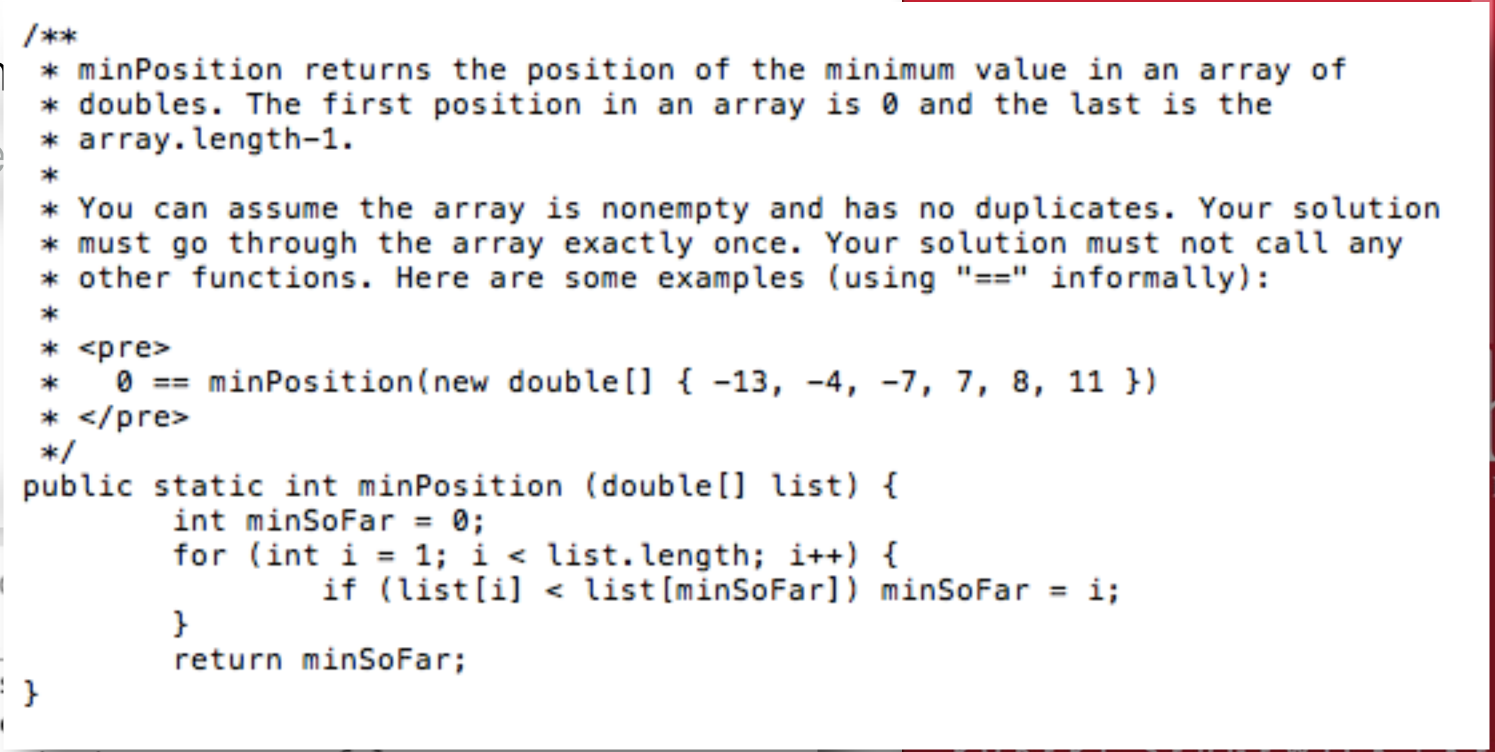
Sam



Problems @ Javadoc De

```
<terminated> Homework1 (1) [Java
The minValue test was succe
The minPosition test was su
The distanceBetweenMinAndMax test was successful.
```





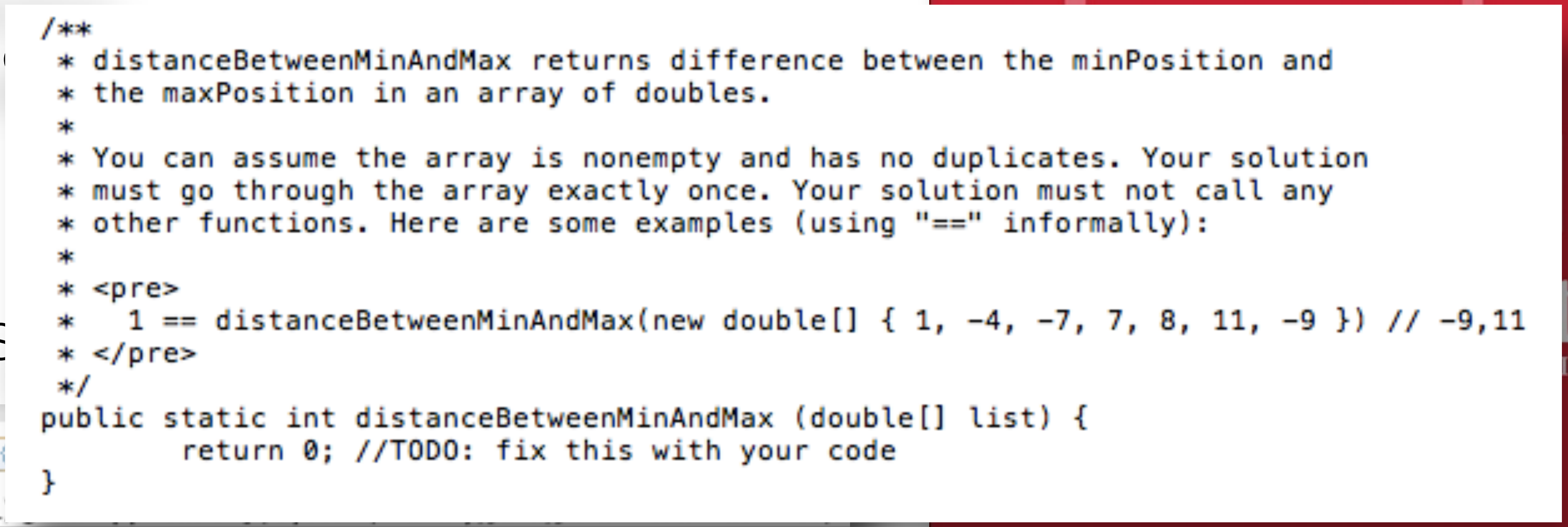
```
/**
 * minPosition returns the position of the minimum value in an array of
 * doubles. The first position in an array is 0 and the last is the
 * array.length-1.
 *
 * You can assume the array is nonempty and has no duplicates. Your solution
 * must go through the array exactly once. Your solution must not call any
 * other functions. Here are some examples (using "==" informally):
 *
 * <pre>
 *   0 == minPosition(new double[] { -13, -4, -7, 7, 8, 11 })
 * </pre>
 */
public static int minPosition (double[] list) {
        int minSoFar = 0;
        for (int i = 1; i < list.length; i++) {
                if (list[i] < list[minSoFar]) minSoFar = i;
        }
        return minSoFar;
}
```

Assignment due next Monday at 11:59 PM

Assignment

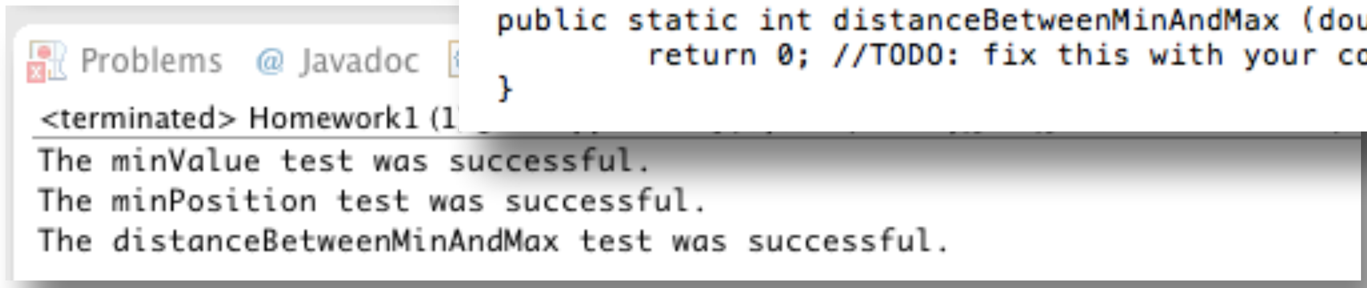
Modify the Homework1.java application to provide valid logic for the following methods:

- minValue
- minPosition
- distanc

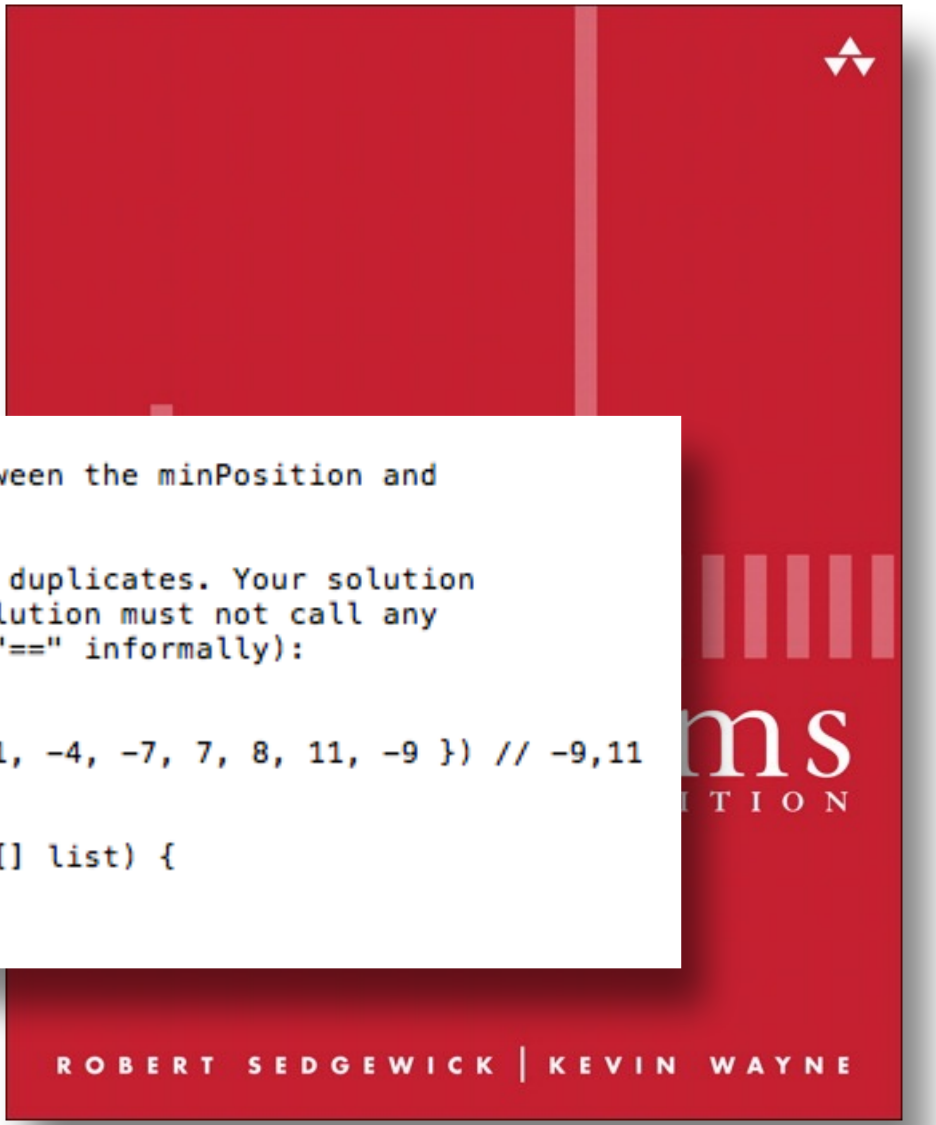


```
/**
 * distanceBetweenMinAndMax returns difference between the minPosition and
 * the maxPosition in an array of doubles.
 *
 * You can assume the array is nonempty and has no duplicates. Your solution
 * must go through the array exactly once. Your solution must not call any
 * other functions. Here are some examples (using "==" informally):
 *
 * <pre>
 *   1 == distanceBetweenMinAndMax(new double[] { 1, -4, -7, 7, 8, 11, -9 }) // -9,11
 * </pre>
 */
public static int distanceBetweenMinAndMax (double[] list) {
        return 0; //TODO: fix this with your code
}
```

S



```
<terminated> Homework1 (1
The minValue test was successful.
The minPosition test was successful.
The distanceBetweenMinAndMax test was successful.
```



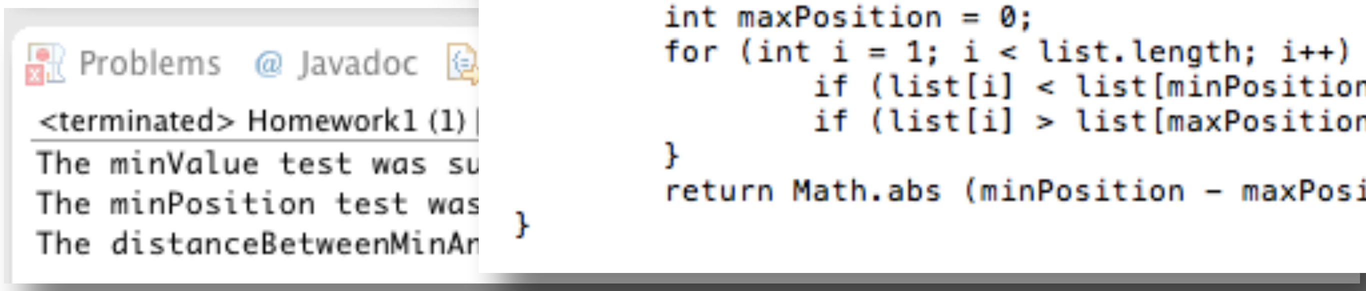
Assignment due next Monday at 11:59 PM

Assignment

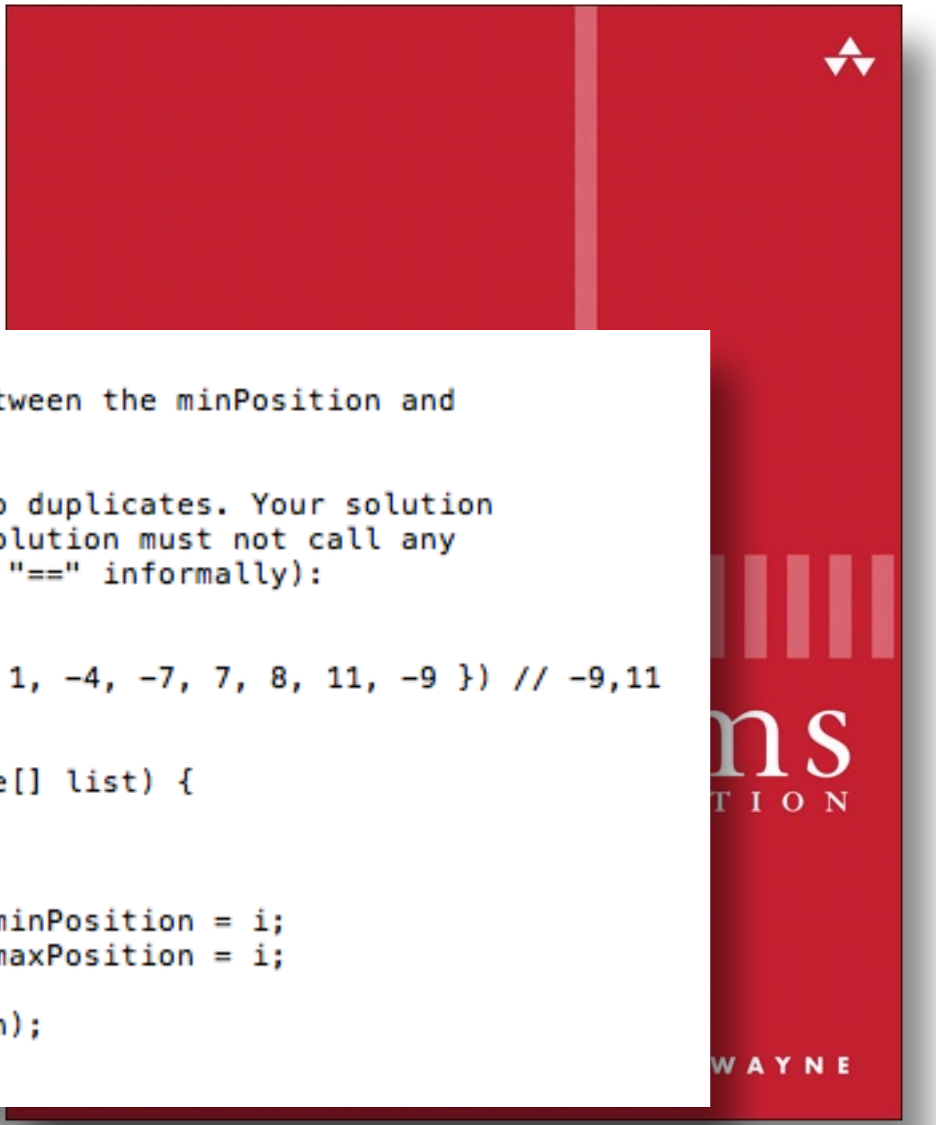
Modify the Homework1.java application to provide valid logic for the following methods:

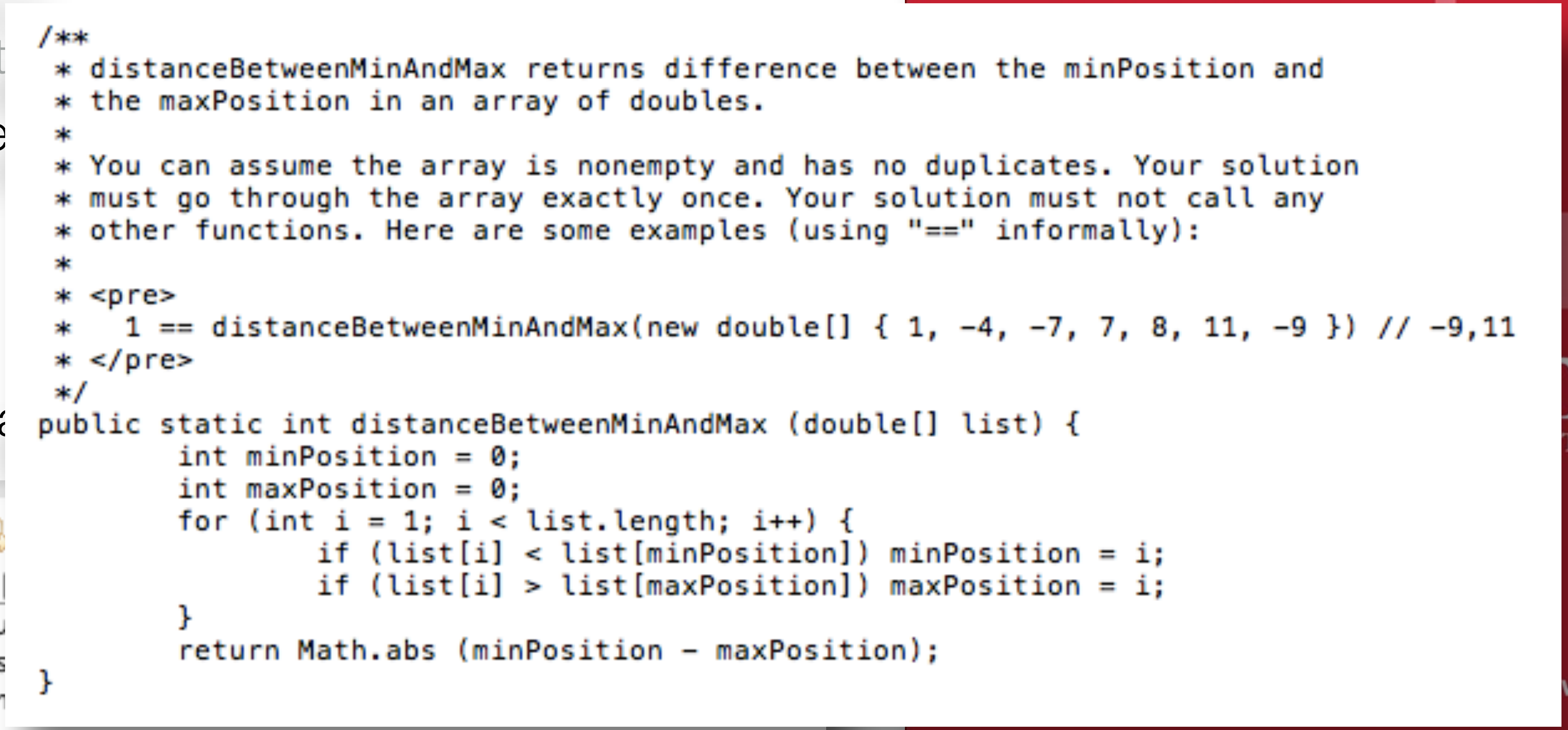
- minValue
- minPosit
- distance

Sa



```
Problems  @ Javadoc
<terminated> Homework1 (1)
The minValue test was su
The minPosition test was
The distanceBetweenMinAn
```





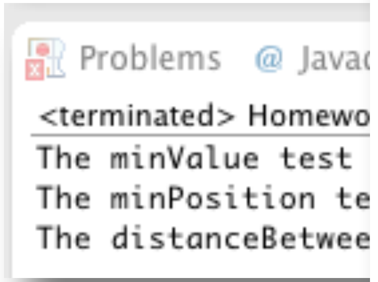
```
/**
 * distanceBetweenMinAndMax returns difference between the minPosition and
 * the maxPosition in an array of doubles.
 *
 * You can assume the array is nonempty and has no duplicates. Your solution
 * must go through the array exactly once. Your solution must not call any
 * other functions. Here are some examples (using "==" informally):
 *
 * <pre>
 *   1 == distanceBetweenMinAndMax(new double[] { 1, -4, -7, 7, 8, 11, -9 }) // -9,11
 * </pre>
 */
public static int distanceBetweenMinAndMax (double[] list) {
        int minPosition = 0;
        int maxPosition = 0;
        for (int i = 1; i < list.length; i++) {
                if (list[i] < list[minPosition]) minPosition = i;
                if (list[i] > list[maxPosition]) maxPosition = i;
        }
        return Math.abs (minPosition - maxPosition);
}
```

Assignment due next Monday at 11:59 PM

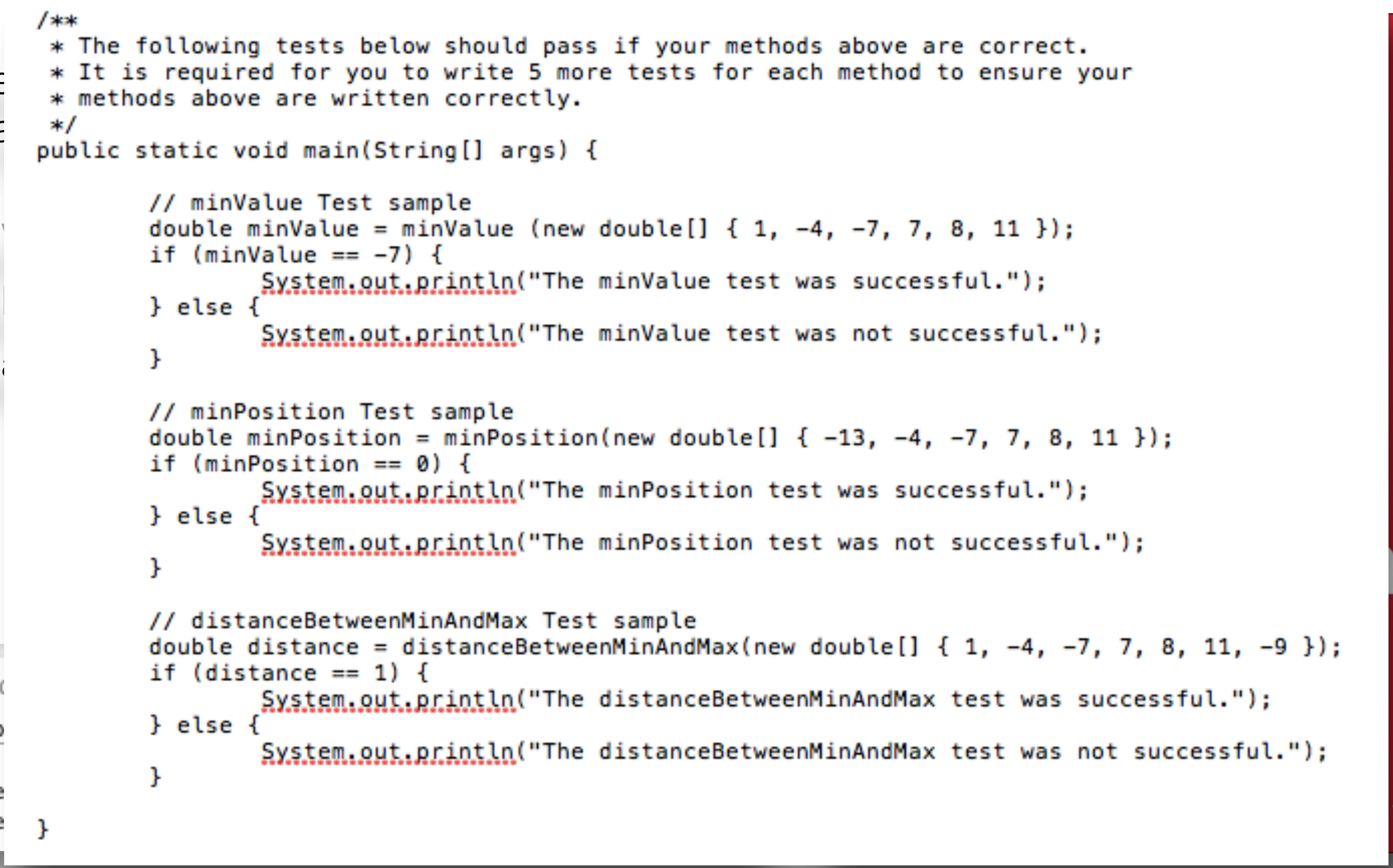
Assignment

Modify the
provide va

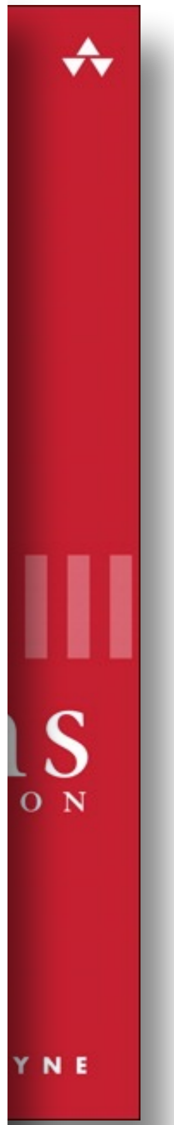
- min
- min
- dist



Problems @ Javad
<terminated> Homewo
The minValue test
The minPosition te
The distanceBetwee

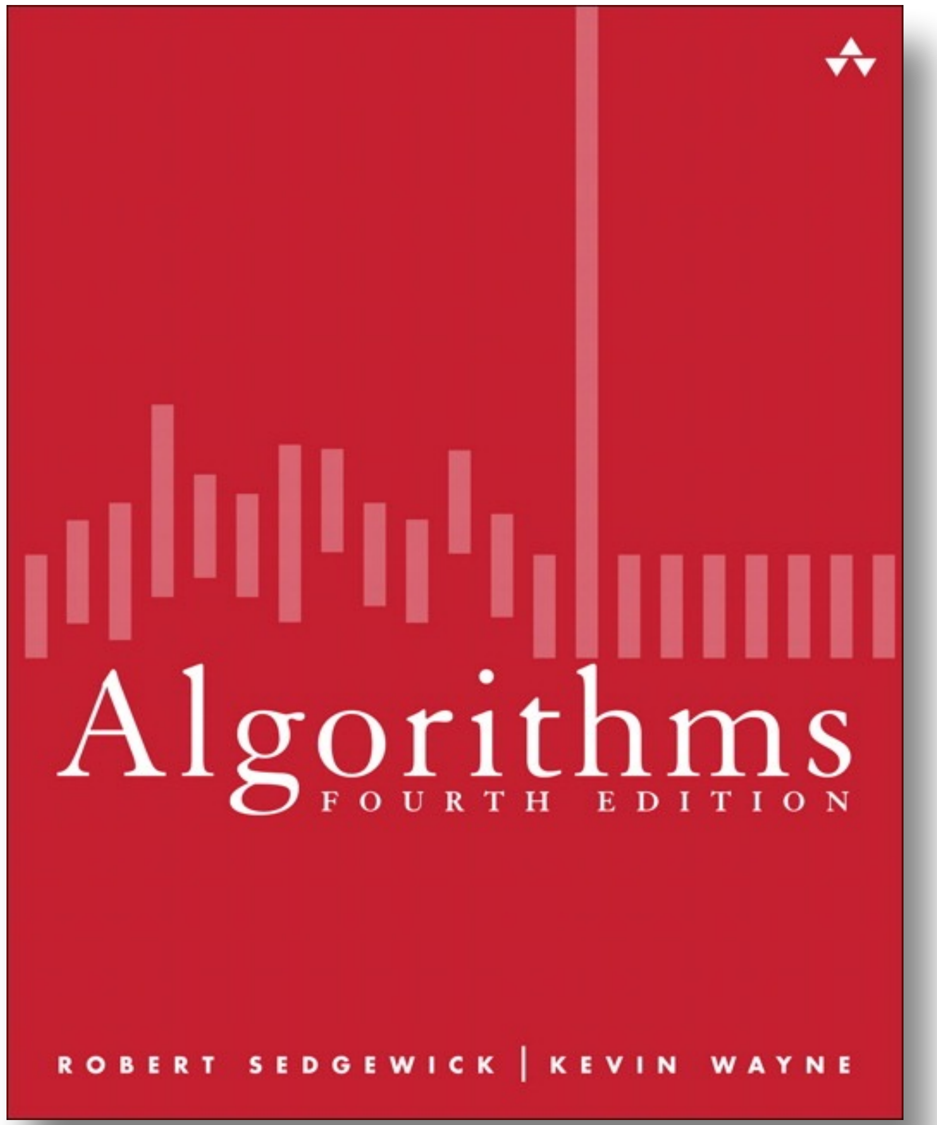


```
/**
 * The following tests below should pass if your methods above are correct.
 * It is required for you to write 5 more tests for each method to ensure your
 * methods above are written correctly.
 */
public static void main(String[] args) {

        // minValue Test sample
        double minValue = minValue (new double[] { 1, -4, -7, 7, 8, 11 });
        if (minValue == -7) {
                System.out.println("The minValue test was successful.");
        } else {
                System.out.println("The minValue test was not successful.");
        }

        // minPosition Test sample
        double minPosition = minPosition(new double[] { -13, -4, -7, 7, 8, 11 });
        if (minPosition == 0) {
                System.out.println("The minPosition test was successful.");
        } else {
                System.out.println("The minPosition test was not successful.");
        }

        // distanceBetweenMinAndMax Test sample
        double distance = distanceBetweenMinAndMax(new double[] { 1, -4, -7, 7, 8, 11, -9 });
        if (distance == 1) {
                System.out.println("The distanceBetweenMinAndMax test was successful.");
        } else {
                System.out.println("The distanceBetweenMinAndMax test was not successful.");
        }

}
```



Assignment due next Monday at 11:59 PM

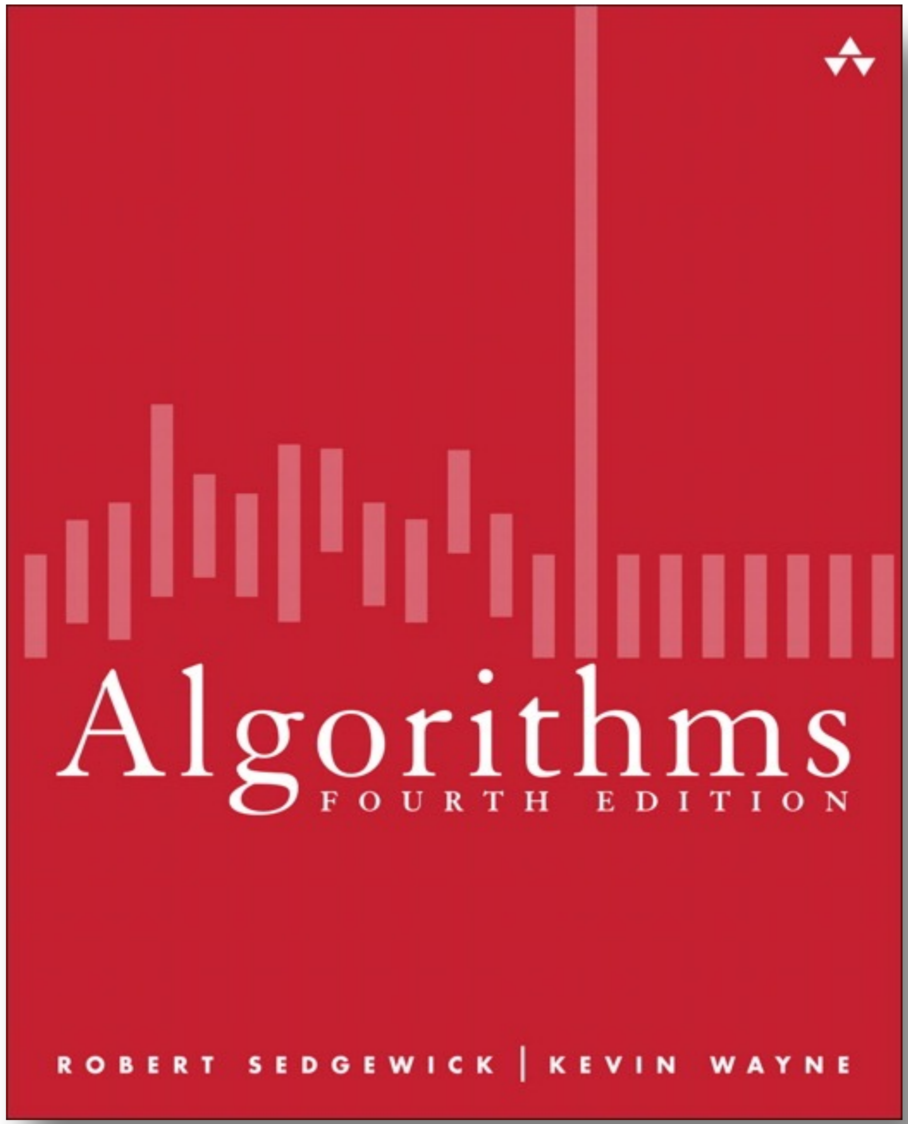
Lecture Overview



1.3 Bags,Queues, and Stacks

- APIs
- Array and resizing array implementations of collections
- Linked lists
- Linked-list implementations of collections
- Iteration

Lecture Overview

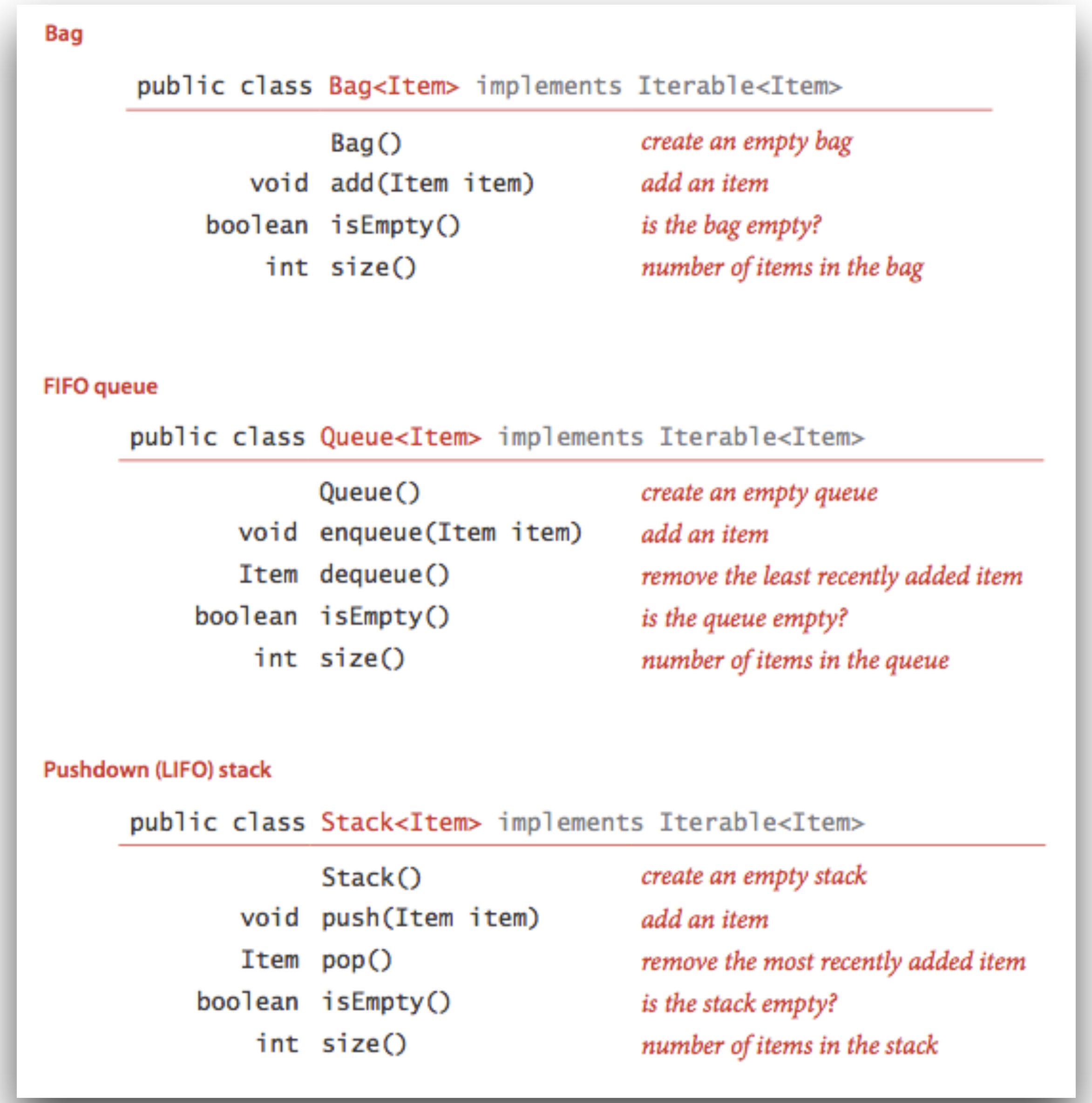


1.3 Bags,Queues, and Stacks

- APIs
- Array and resizing array implementations of collections
- Linked lists
- Linked-list implementations of collections
- Iteration

API's

We define the APIs for bags, queues, and stacks. Beyond the basics, these APIs reflect two Java features: generics and iterable collections.



Bag

`public class Bag<Item> implements Iterable<Item>`

	`Bag()`	*create an empty bag*
`void`	`add(Item item)`	*add an item*
`boolean`	`isEmpty()`	*is the bag empty?*
`int`	`size()`	*number of items in the bag*

FIFO queue

`public class Queue<Item> implements Iterable<Item>`

	`Queue()`	*create an empty queue*
`void`	`enqueue(Item item)`	*add an item*
`Item`	`dequeue()`	*remove the least recently added item*
`boolean`	`isEmpty()`	*is the queue empty?*
`int`	`size()`	*number of items in the queue*

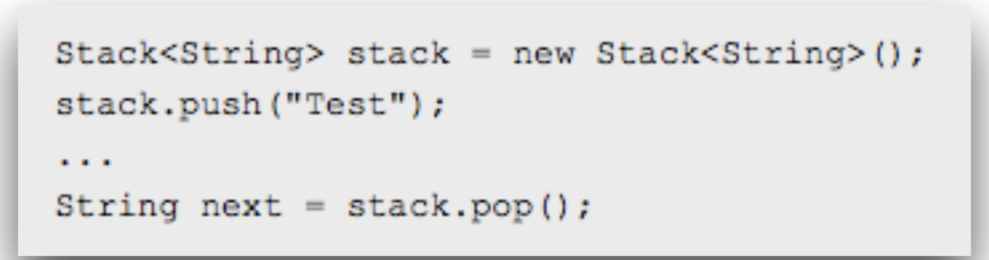
Pushdown (LIFO) stack

`public class Stack<Item> implements Iterable<Item>`

	`Stack()`	*create an empty stack*
`void`	`push(Item item)`	*add an item*
`Item`	`pop()`	*remove the most recently added item*
`boolean`	`isEmpty()`	*is the stack empty?*
`int`	`size()`	*number of items in the stack*

API’s

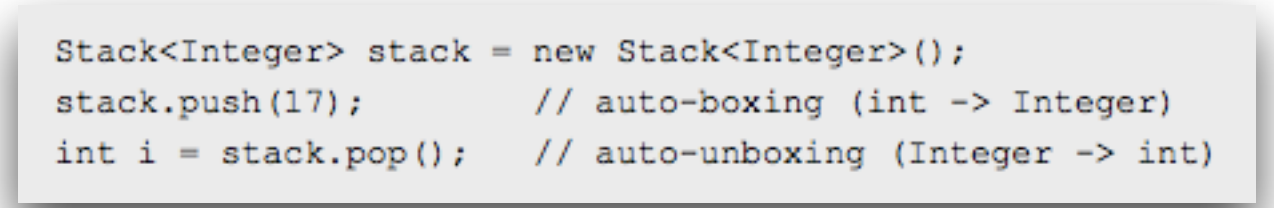
Generics. An essential characteristic of collection ADTs is that we should be able to use them for any type of data. A specific Java mechanism known as generics enables this capability. The notation <Item> after the class name in each of our APIs defines the name Item as a type parameter, a symbolic placeholder for some concrete type to be used by the client. You can read Stack<Item> as "stack of items." For example, you can write code such as:



```
Stack<String> stack = new Stack<String>();
stack.push("Test");
...
String next = stack.pop();
```

API’s

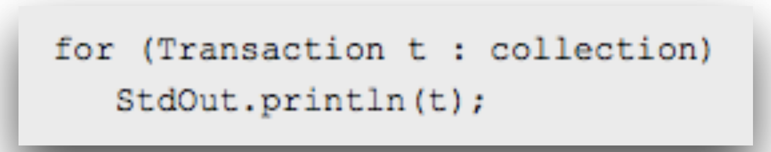
Autoboxing: Type parameters have to be instantiated as reference types, so Java automatically converts between a primitive type and its corresponding wrapper type in assignments, method arguments, and arithmetic/logic expressions. Automatically casting a primitive type to a wrapper type is known as autoboxing, and automatically casting a wrapper type to a primitive type is known as auto-unboxing. This conversion enables us to use generics with primitive types, as in the following code:



```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);         // auto-boxing (int -> Integer)
int i = stack.pop();    // auto-unboxing (Integer -> int)
```

API's

Iterable collections. For many applications, the client's requirement is just to process each of the items in some way, or to iterate through the items in the collection. Java's foreach statement supports this paradigm. For example, suppose that collection is a Queue<Transaction>. Then, if the collection is iterable, the client can print a transaction list with a single statement:



```
for (Transaction t : collection)
   StdOut.println(t);
```

API’s

Bags. A bag is a collection where removing items is not supported—its purpose is to provide clients with the ability to collect items and then to iterate through the collected items. Stats.java is a bag client that reads a sequence of real numbers from standard input and prints out their mean and standard deviation.

FIFO queues. A FIFO queue is a collection that is based on the first-in-first-out (FIFO) policy. The policy of doing tasks in the same order that they arrive server is one that we encounter frequently in everyday life: from people waiting in line at a theater, to cars waiting in line at a toll booth, to tasks waiting to be serviced by an application on your computer.

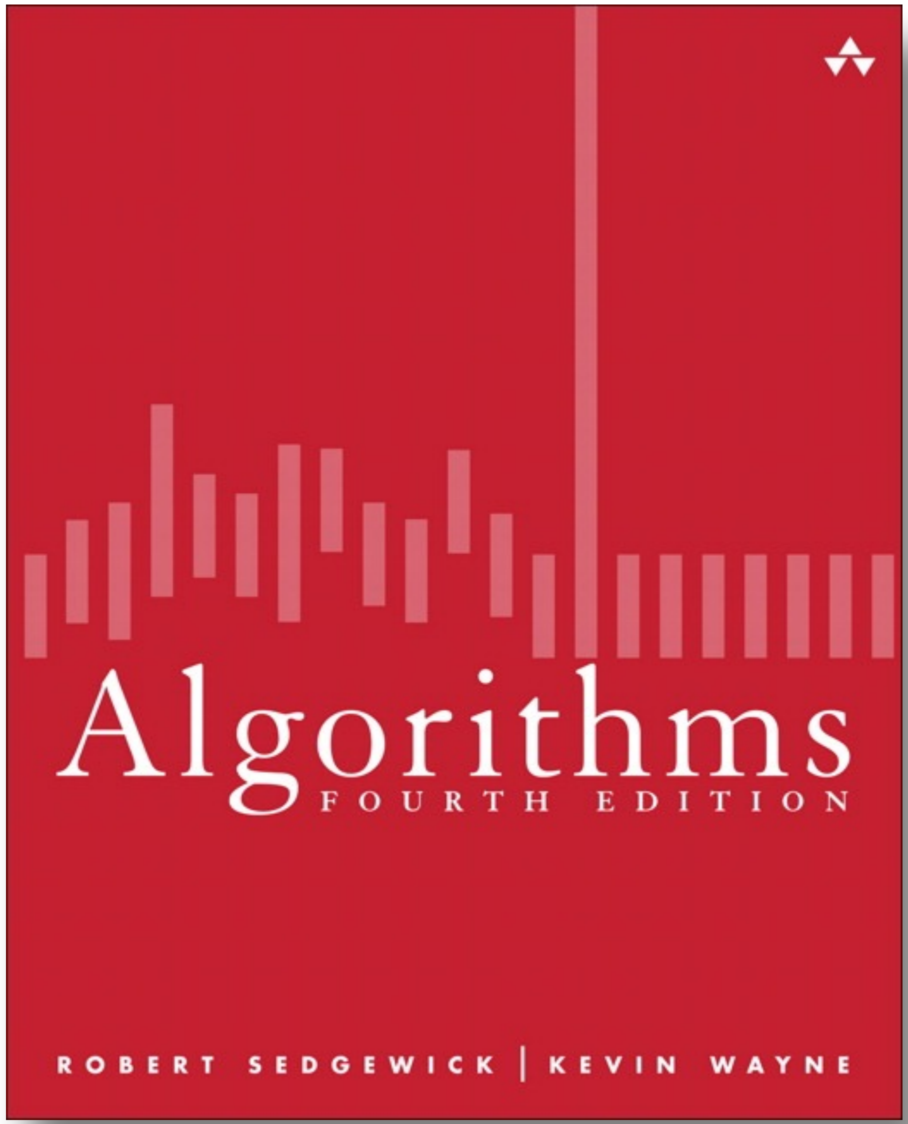
Pushdown stack. A pushdown stack is a collection that is based on the last-in-first-out (LIFO) policy. When you click a hyperlink, your browser displays the new page (and pushes onto a stack). You can keep clicking on hyperlinks to visit new pages, but you can always revisit the previous page by clicking the back button (popping it from the stack). Reverse.java is a stack client that reads a sequence of integers from standard input and prints them in reverse order.

API’s

Arithmetic expression evaluation. Evaluate.java is a stack client that evaluates fully parenthesized arithmetic expressions. It uses Dijkstra's 2-stack algorithm:

- Push operands onto the operand stack.
- Push operators onto the operator stack.
- Ignore left parentheses.
- On encountering a right parenthesis, pop an operator, pop the requisite number of operands, and push onto the operand stack the result of applying that operator to those operands.

Lecture Overview



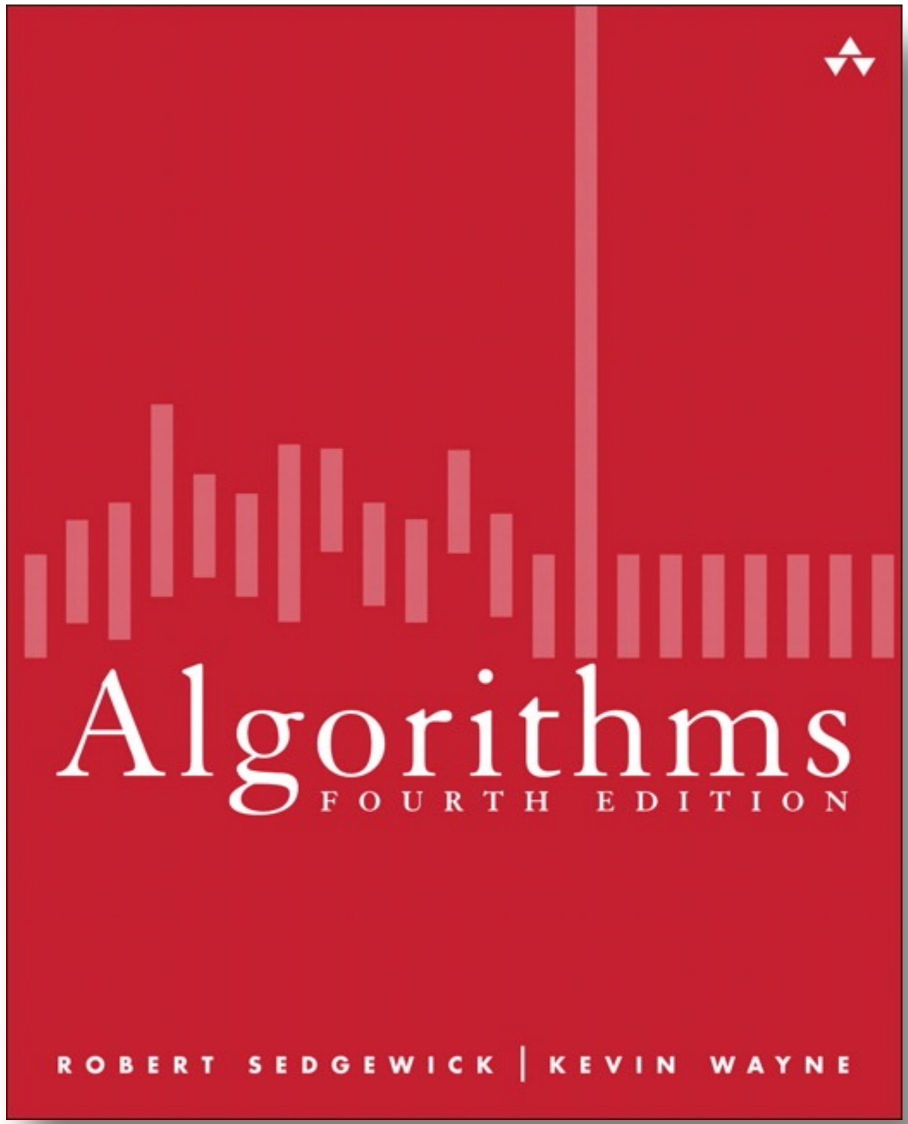
1.3 Bags,Queues, and Stacks

- APIs
- Array and resizing array implementations of collections
- Linked lists
- Linked-list implementations of collections
- Iteration

Array and resizing array implementations of collections

- Fixed-capacity stack of strings. FixedCapacityStackOfString.java implements a fixed-capacity stack of strings using an array.
- Fixed-capacity generic stack. FixedCapacityStack.java implements a generic fixed-capacity stack.
- Array resizing stack. ResizingArrayStack.java implements a generic stack using a resizing array. With a resizing array, we dynamically adjust the size of the array so that it is both sufficiently large to hold all of the items and not so large as to waste an excessive amount of space. We double the size of the array in push() if it is full; we halve the size of the array in pop() if it is less than one-quarter full.
- Array resizing queue. ResizingArrayQueue.java implements the queue API with a resizing array.

Lecture Overview

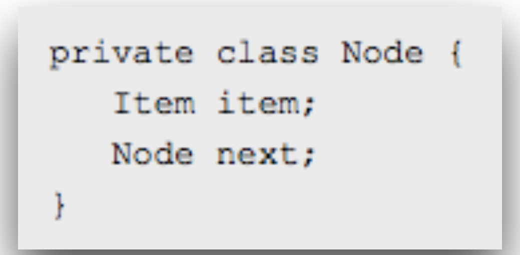


1.3 Bags,Queues, and Stacks

- APIs
- Array and resizing array implementations of collections
- Linked lists
- Linked-list implementations of collections
- Iteration

Linked Lists

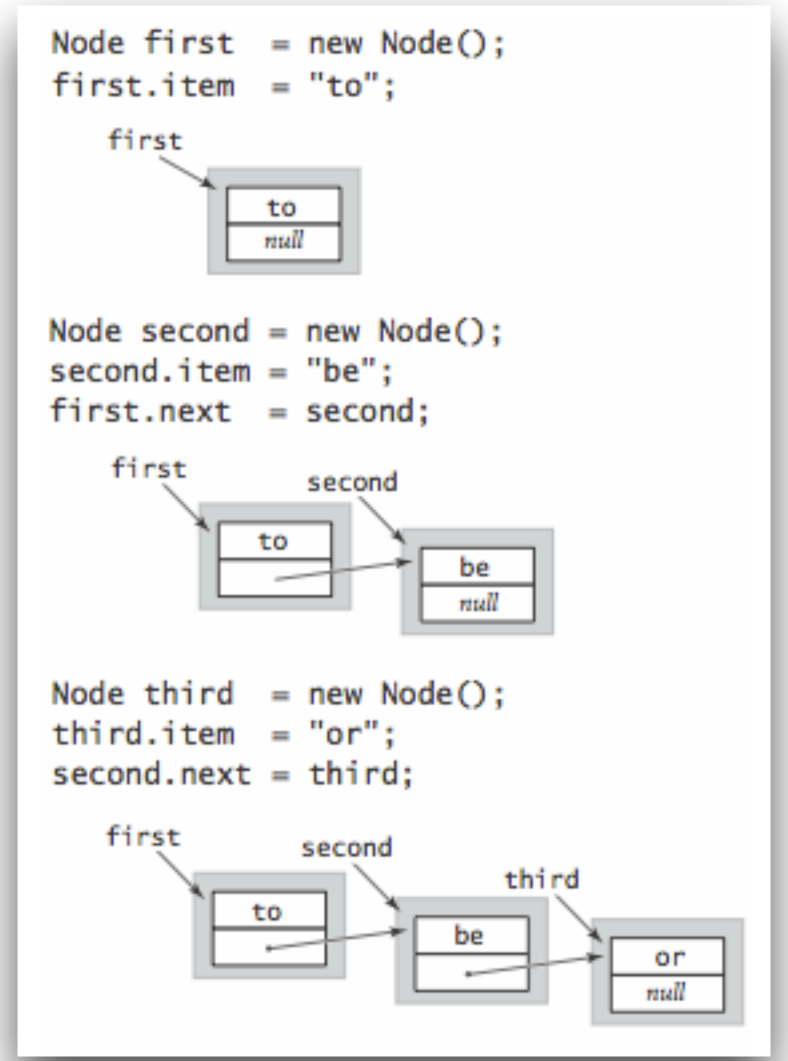
Linked lists. A linked list is a recursive data structure that is either empty (null) or a reference to a node having a generic item and a reference to a linked list. To implement a linked list, we start with a nested class that defines the node abstraction



```
private class Node {
   Item item;
   Node next;
}
```

Linked Lists

Building a linked list. To build a linked list that contains the items to, be, and or, we create a Node for each item, set the item field in each of the nodes to the desired value, and set the next fields to build the linked list.



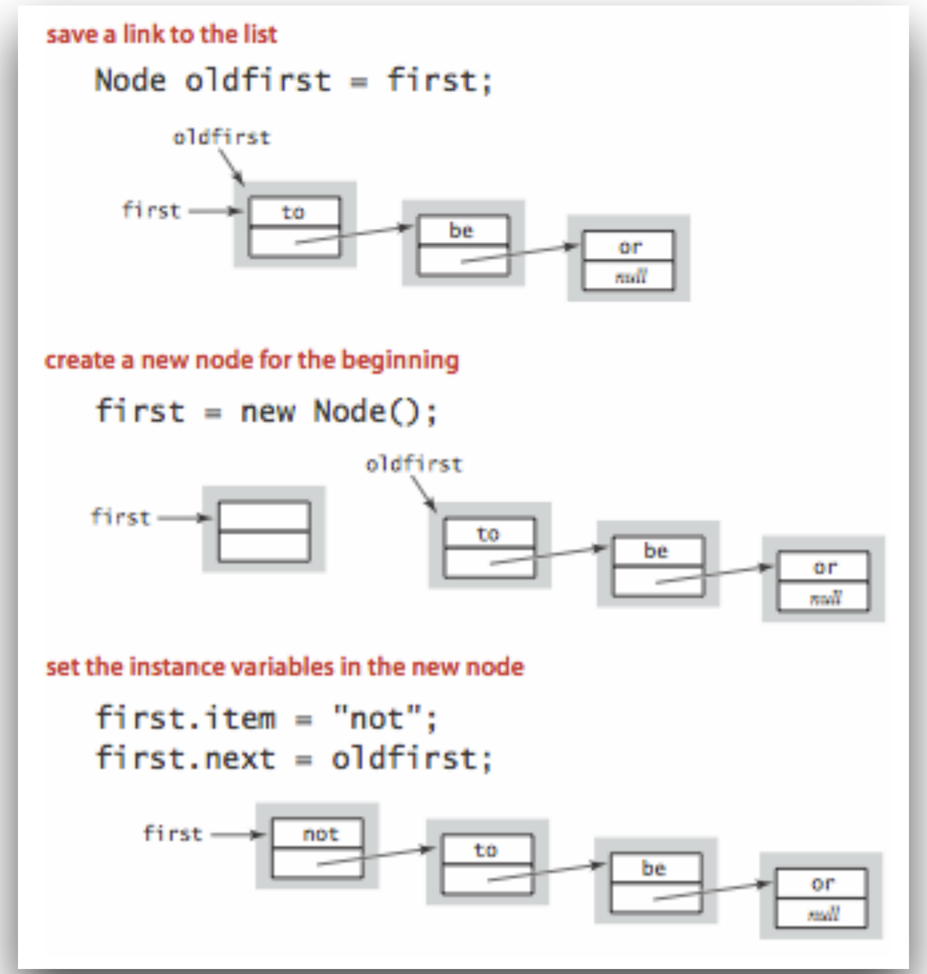
```
Node first  = new Node();
first.item  = "to";
```

```
Node second = new Node();
second.item = "be";
first.next  = second;
```

```
Node third  = new Node();
third.item  = "or";
second.next = third;
```

Linked Lists

Insert at the beginning. The easiest place to insert a new node in a linked list is at the beginning.



save a link to the list

```
Node oldfirst = first;
```

create a new node for the beginning

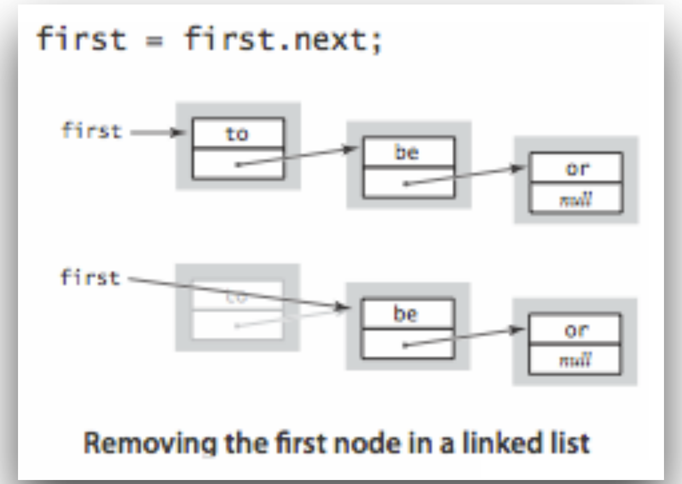
```
first = new Node();
```

set the instance variables in the new node

```
first.item = "not";
first.next = oldfirst;
```

Linked Lists

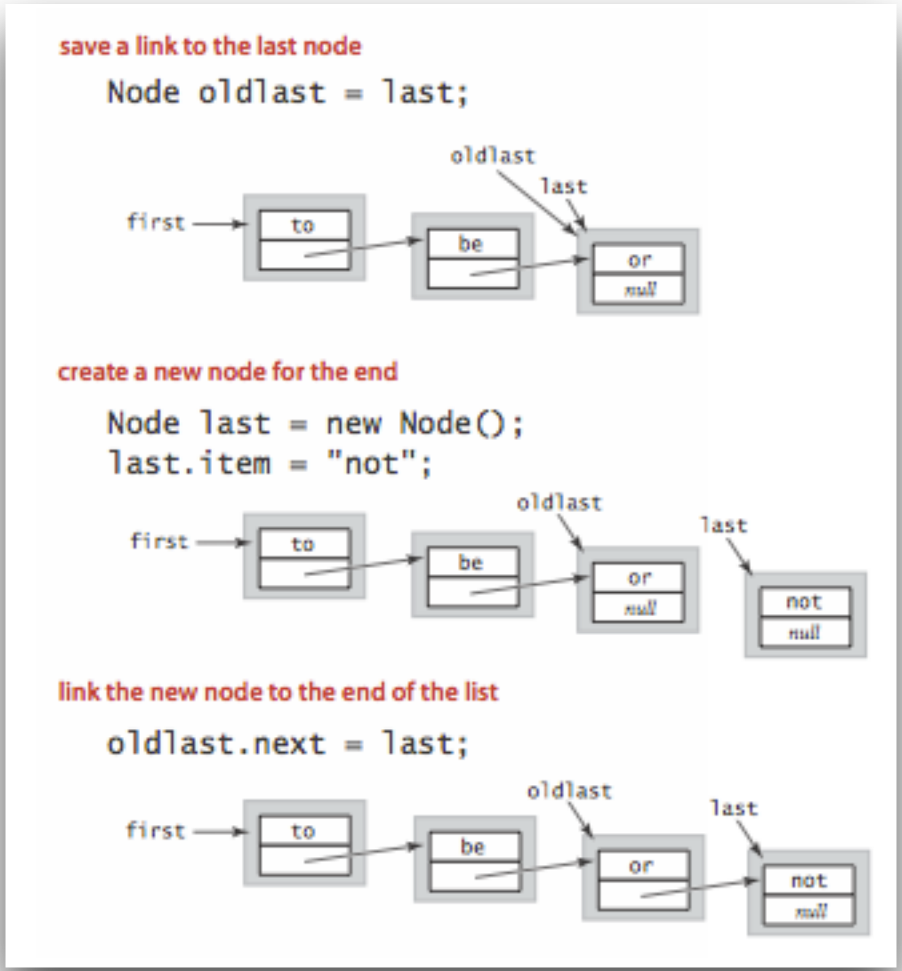
Remove from the beginning. Removing the first node in a linked list is also easy.



Removing the first node in a linked list

Linked Lists

Insert at the end. To insert a node at the end of a linked list, we maintain a link to the last node in the list.


save a link to the last node

```
Node oldlast = last;
```

create a new node for the end

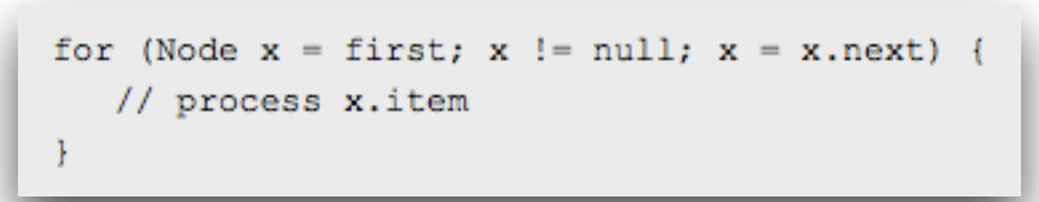
```
Node last = new Node();
last.item = "not";
```

link the new node to the end of the list

```
oldlast.next = last;
```

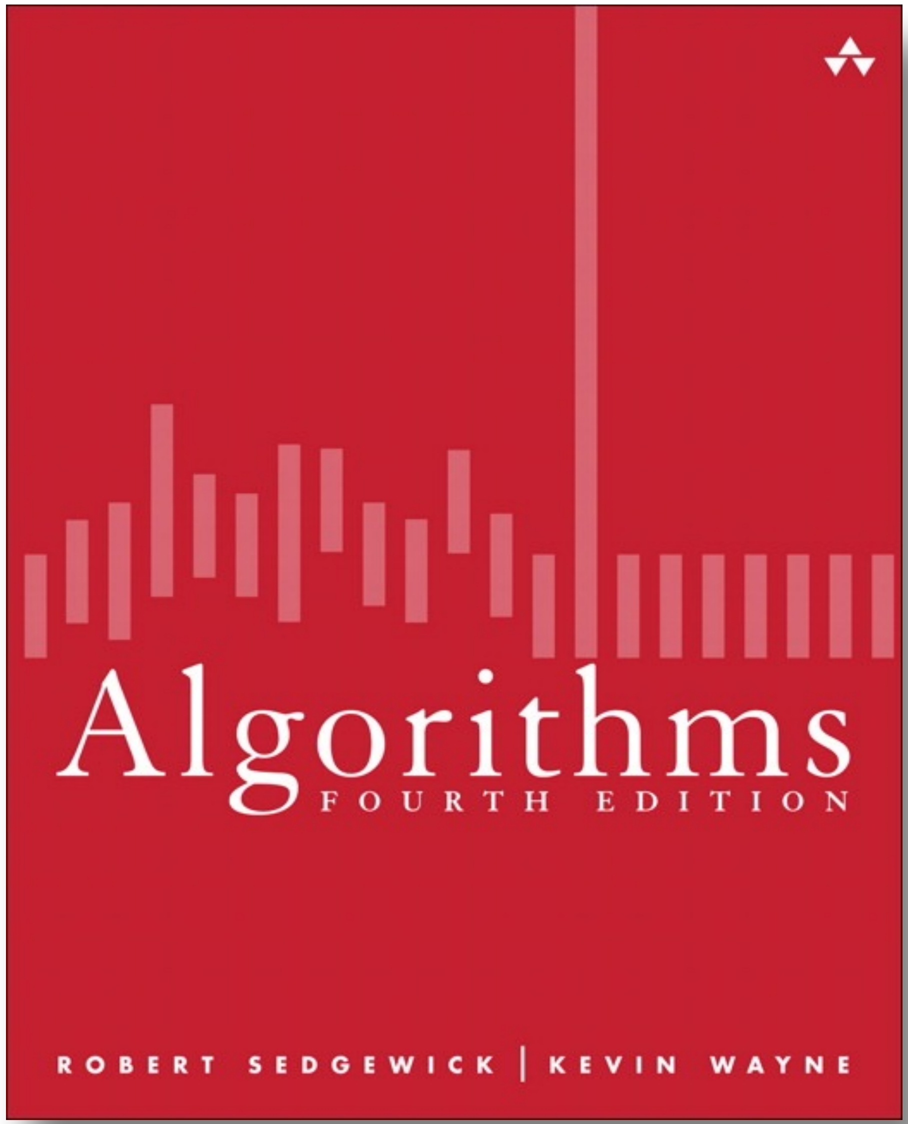
Linked Lists

Traversal. The following is the idiom for traversing the nodes in a linked list.



```java
for (Node x = first; x != null; x = x.next) {
   // process x.item
}
```

Lecture Overview



1.3 Bags,Queues, and Stacks

- APIs
- Array and resizing array implementations of collections
- Linked lists
- Linked-list implementations of collections
- Iteration

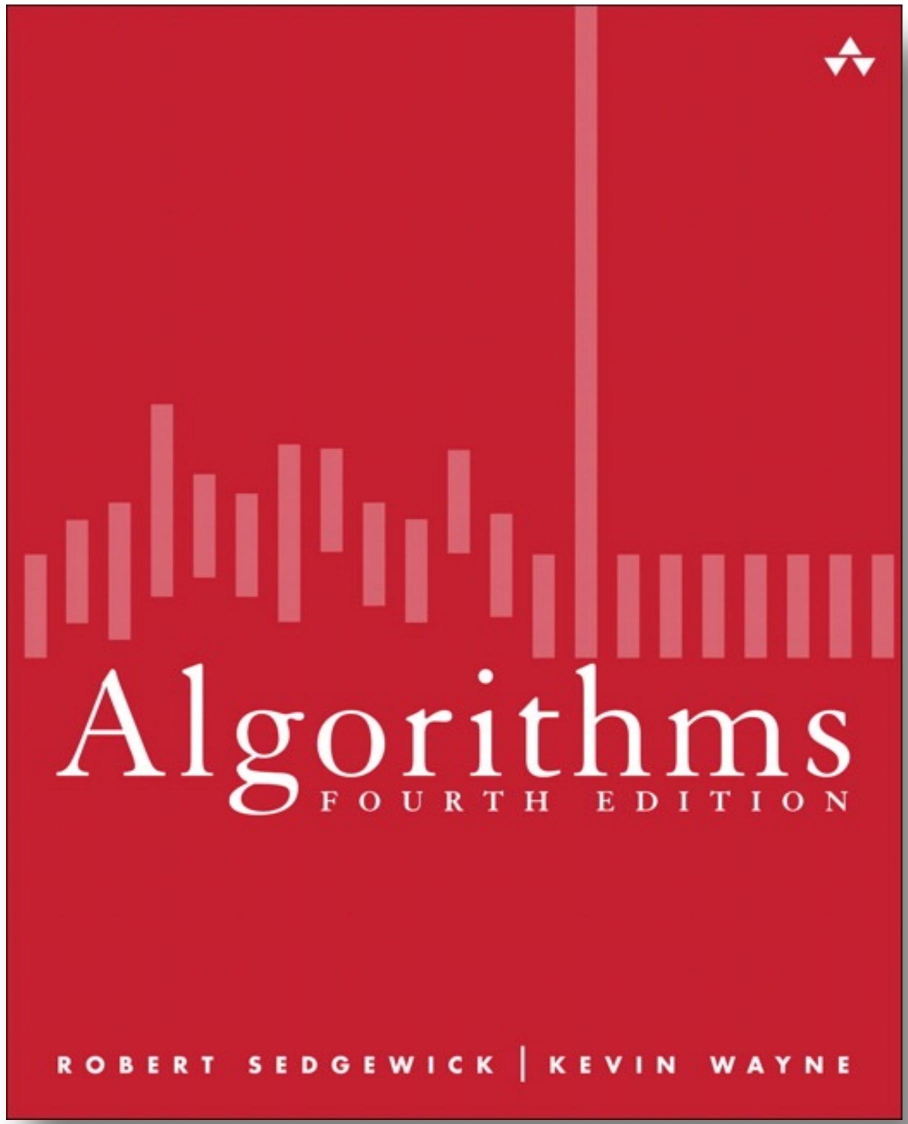
Linked-list implementations of collections

Linked list implementation of a stack. Stack.java implements a generic stack using a linked list. It maintains the stack as a linked list, with the top of the stack at the beginning, referenced by an instance variable first. To push() an item, we add it to the beginning of the list; to pop() an item, we remove it from the beginning of the list.

Linked list implementation of a queue. Program Queue.java implements a generic FIFO queue using a linked list. It maintains the queue as a linked list in order from least recently to most recently added items, with the beginning of the queue referenced by an instance variable first and the end of the queue referenced by an instance variable last. To enqueue() an item, we add it to the end of the list; to dequeue() an item, we remove it from the beginning of the list.

Linked list implementation of a bag. Program Bag.java implements a generic bag using a linked list. The implementation is the same as Stack.java except for changing the name of push() to add() and removing pop().

Lecture Overview



1.3 Bags,Queues, and Stacks

- APIs
- Array and resizing array implementations of collections
- Linked lists
- Linked-list implementations of collections
- Iteration

Iteration

Iteration. To consider the task of implementing iteration, we start with a snippet of client code that prints all of the items in a collection of strings, one per line:

```
Stack<String> collection = new Stack<String>();
...
for (String s : collection)
   StdOut.println(s);
...
```

This foreach statement is shorthand for the following while statement:

```
Iterator<String> i = collection.iterator();
while (i.hasNext()) {
   String s = i.next();
   StdOut.println(s);
}
```

Iteration

To implement iteration in a collection:

- Include the following import statement so that our code can refer to Java's java.util.Iterator interface:

```
import java.util.Iterator;
```

- Add the following to the class declaration, a promise to provide an iterator() method, as specified in the java.lang.Iterable interface:

```
implements Iterable<Item>
```

- Implement a method iterator() that returns an object from a class that implements the Iterator interface:.

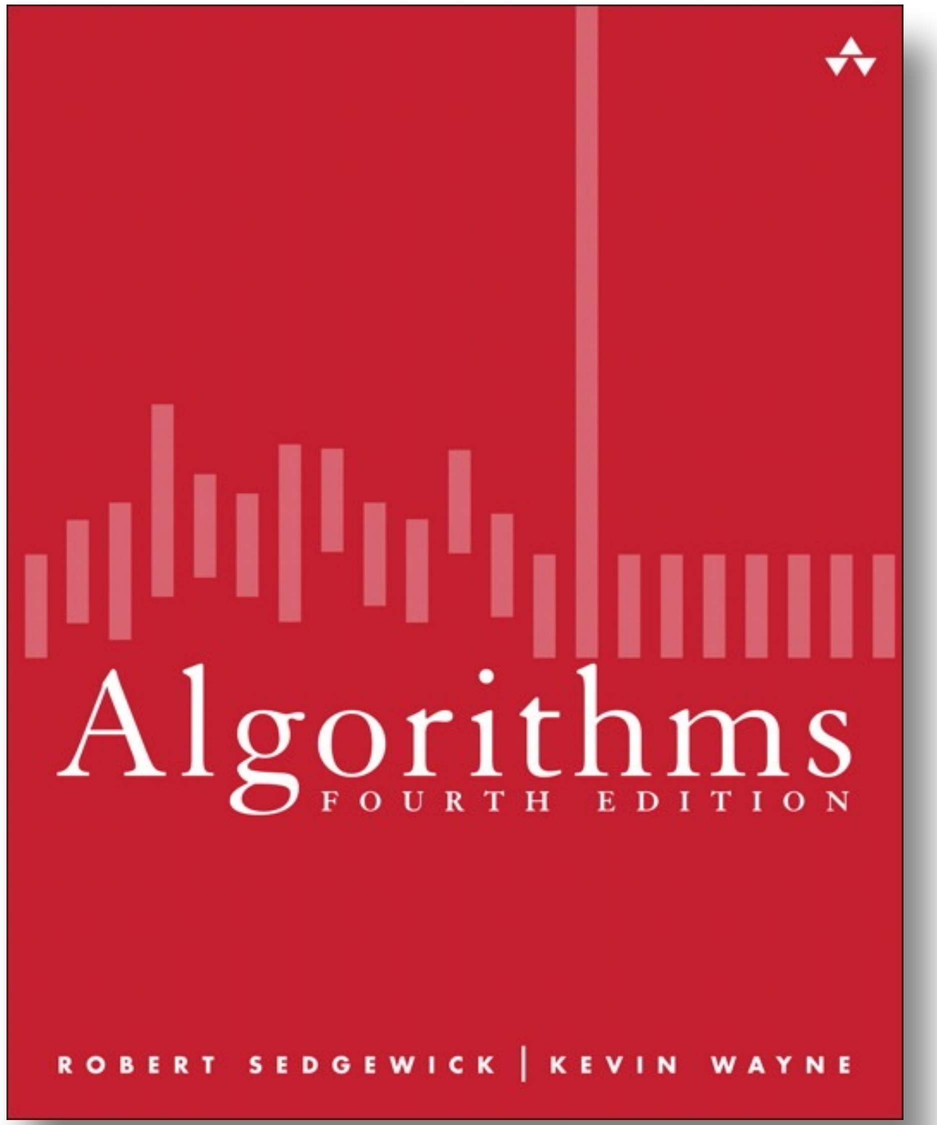
```
public Iterator<Item> iterator() {
    return new ListIterator();
}
```

Iteration

Implement a nested class that implements the Iterator interface by including the methods hasNext(), next(), and remove(). We always use an empty method for the optional remove() method because interleaving iteration with operations that modify the data structure is best avoided.

- The nested class ListIterator in Bag.java illustrates how to implement a class that implements the Iterator interface when the underlying data structure is a linked list.
- The nested class ArrayIterator in ResizingArrayBag.java does the same when the underlying data structure is an array.

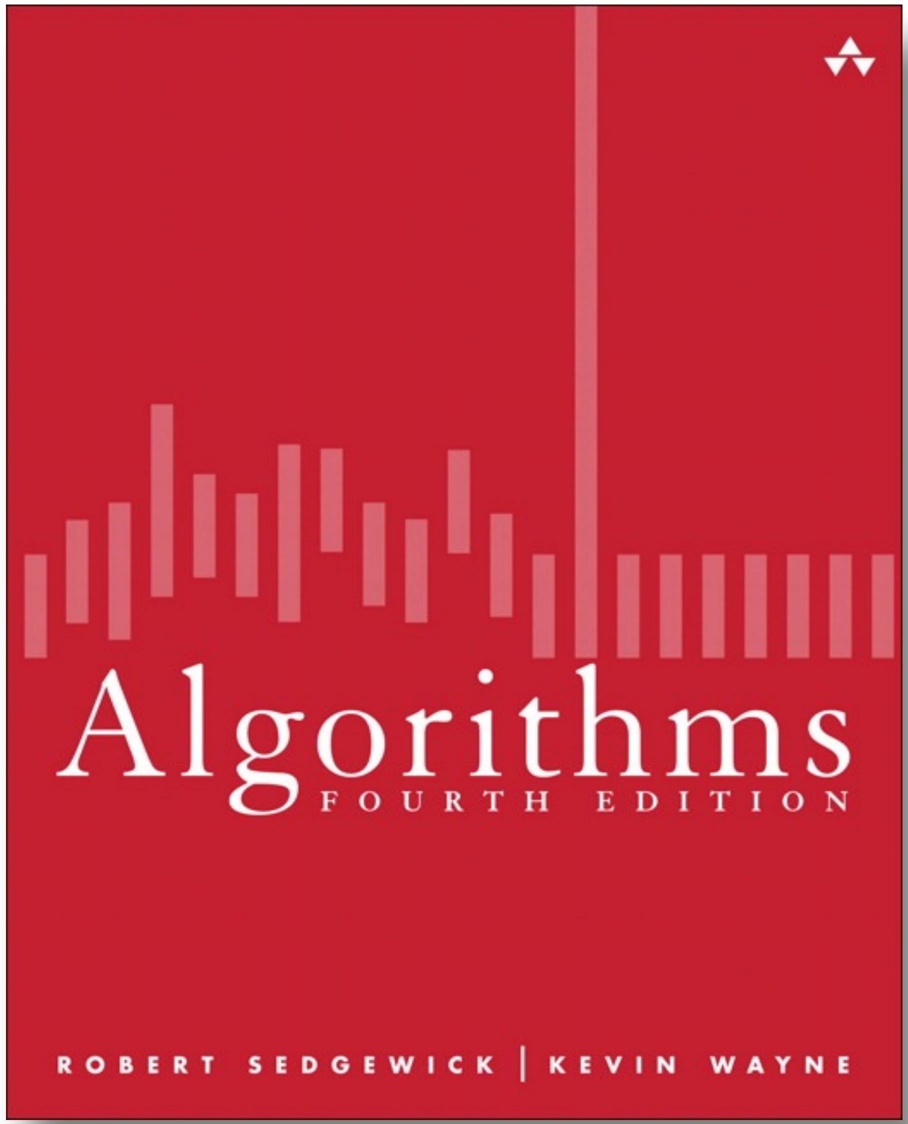
Lecture Overview



1.4 Analysis of Algorithms

- Scientific method
- Observations
- Mathematical models
- Coping with dependence on inputs
- Memory usage

Lecture Overview



1.4 Analysis of Algorithms

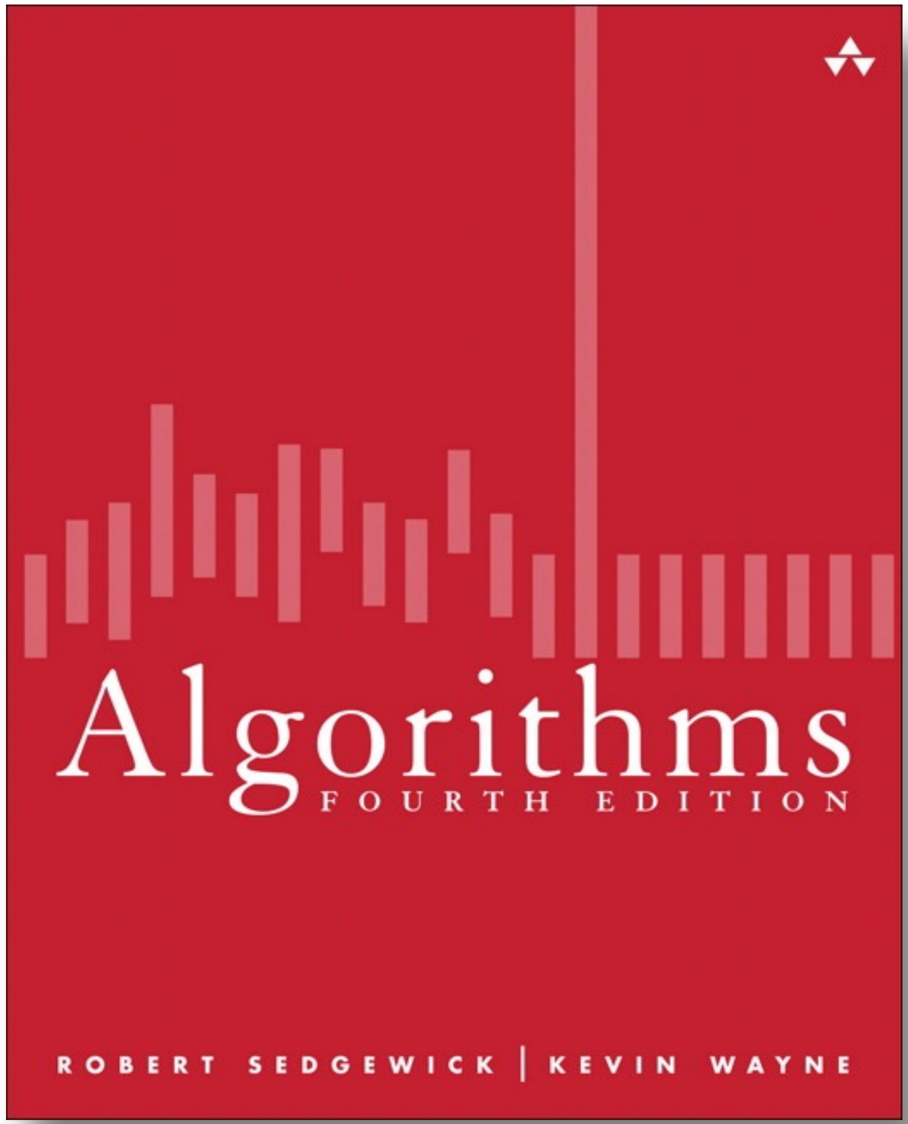
- Scientific method
- Observations
- Mathematical models
- Coping with dependence on inputs
- Memory usage

Scientific method

Scientific method. The very same approach that scientists use to understand the natural world is effective for studying the running time of programs. The experiments we design must be reproducible and the hypotheses that we formulate must be falsifiable:

- Observe some feature of the natural world, generally with precise measurements.
- Hypothesize a model that is consistent with the observations.
- Predict events using the hypothesis.
- Verify the predictions by making further observations.
- Validate by repeating until the hypothesis and observations agree..

Lecture Overview

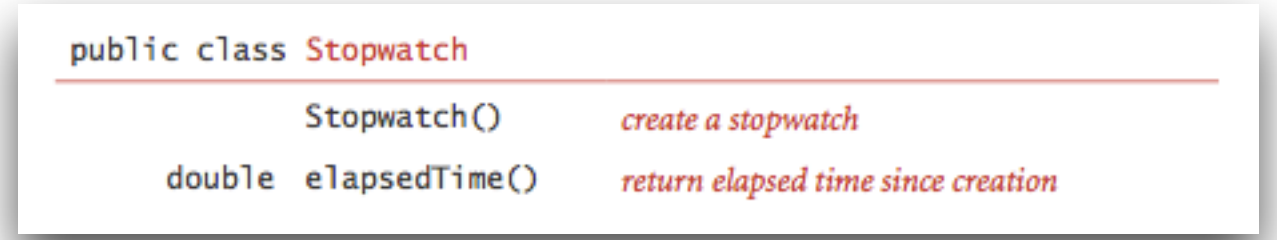


1.4 Analysis of Algorithms

- Scientific method
- Observations
- Mathematical models
- Coping with dependence on inputs
- Memory usage

Observations

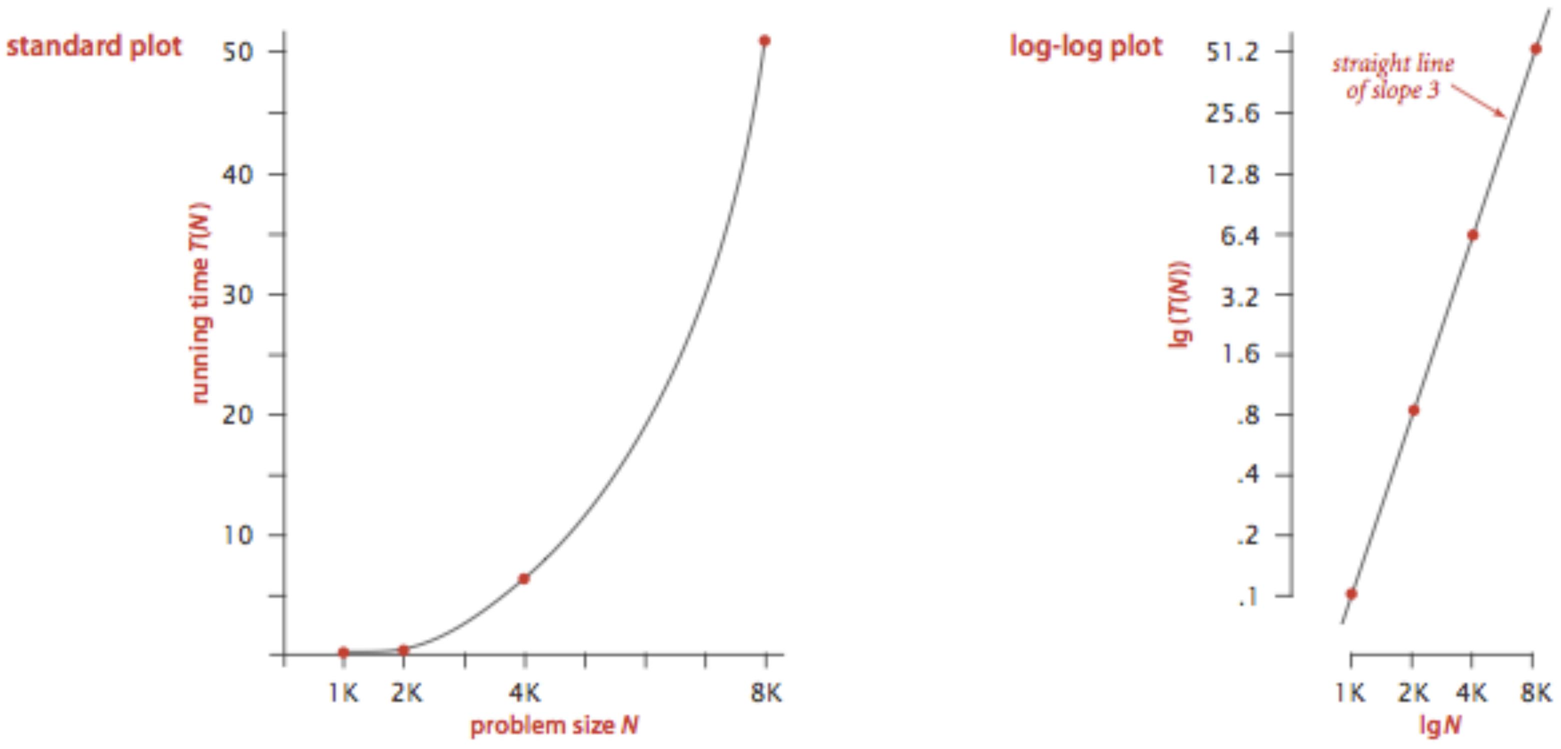
Our first challenge is to determine how to make quantitative measurements of the running time of our programs. Stopwatch.java is a data type that measures the elapsed running time of a program.



```
public class Stopwatch
-------------------------------------------------------------
                Stopwatch()       create a stopwatch
        double  elapsedTime()     return elapsed time since creation
```

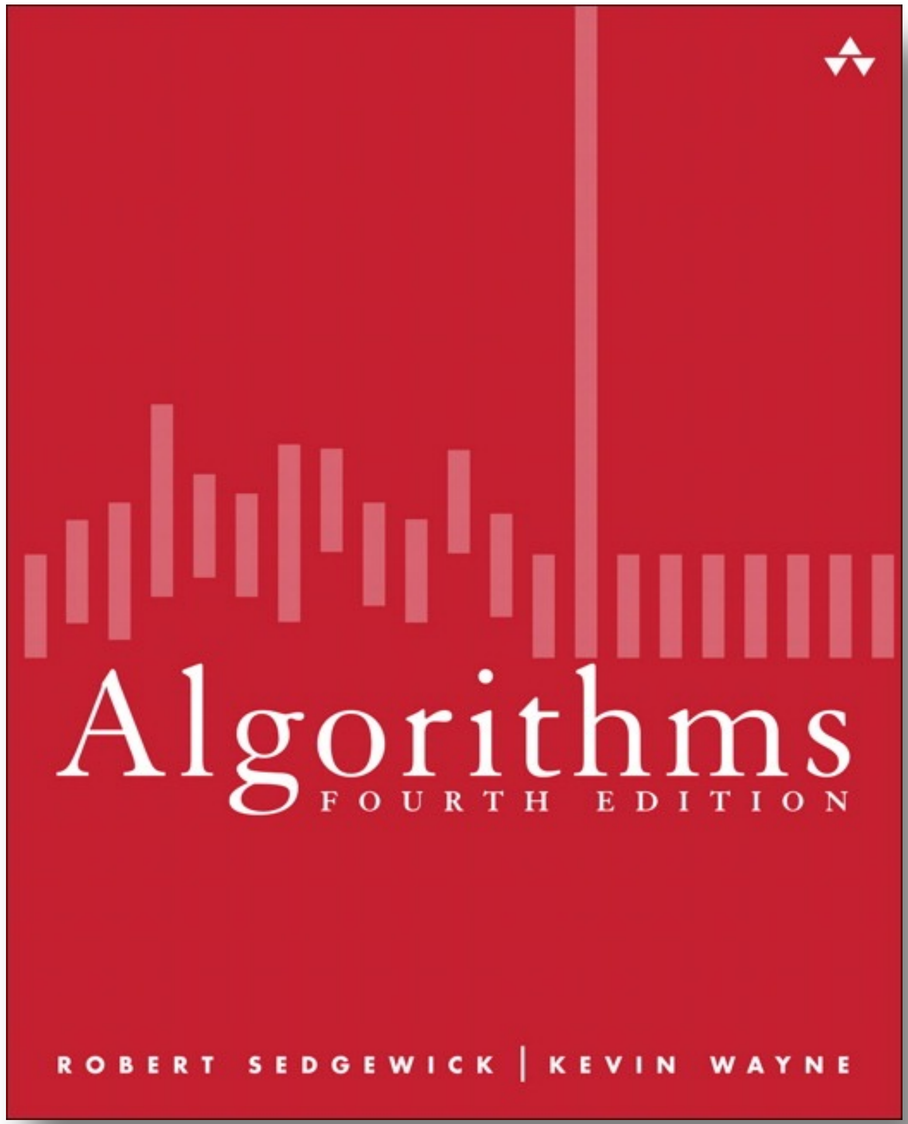
Observations

ThreeSum.java counts the number of triples in a file of N integers that sums to 0 (ignoring integer overflow). DoublingTest.java generates a sequence of random input arrays, doubling the array size at each step, and prints the running times of ThreeSum.count() for each input size. DoublingRatio.java is similar but also output the ratios in running times from one size to the next.



Analysis of experimental data (the running time of ThreeSum.count())

Lecture Overview



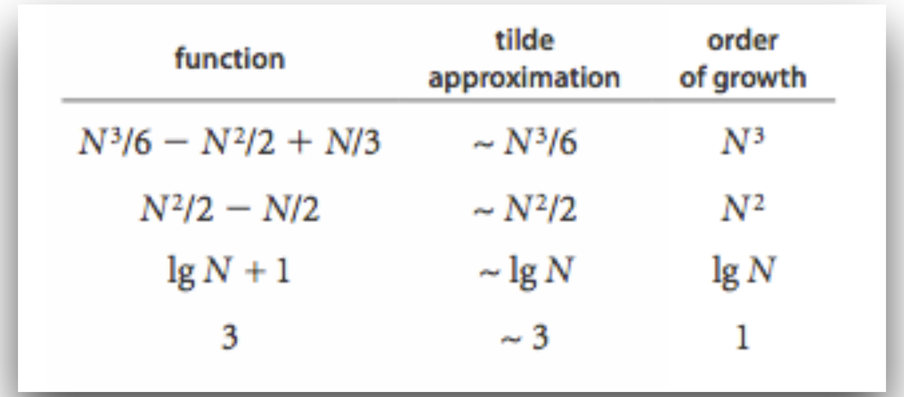
1.4 Analysis of Algorithms

- Scientific method
- Observations
- Mathematical models
- Coping with dependence on inputs
- Memory usage

Mathematical models

The total running time of a program is determined by two primary factors: the cost of executing each statement and the frequency of execution of each statement.

Tilde approximations. We use tilde approximations, where we throw away low-order terms that complicate formulas. We write ~ f(N) to represent any function that when divided by f(N) approaches 1 as N grows. We write g(N) ~ f(N) to indicate that g(N) / f(N) approaches 1 as N grows.

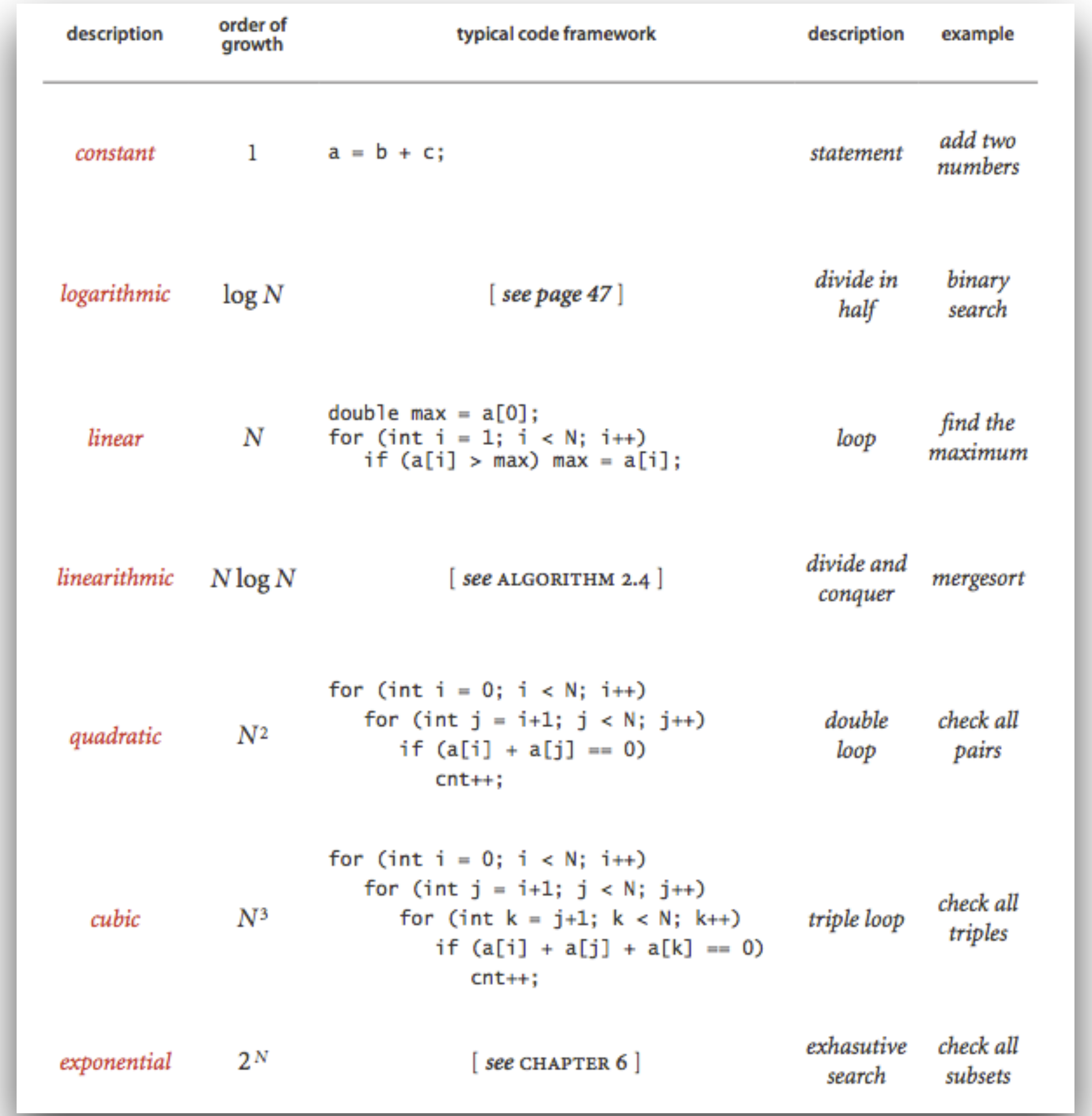


function	tilde approximation	order of growth
$N^3/6 - N^2/2 + N/3$	$\sim N^3/6$	N^3
$N^2/2 - N/2$	$\sim N^2/2$	N^2
$\lg N + 1$	$\sim \lg N$	$\lg N$
3	~ 3	1

Cost model. We focus attention on properties of algorithms by articulating a cost model that defines the basic operations. For example, an appropriate cost model for the 3-sum problem is the number of times we access an array entry, for read or write

Mathematical models

Order-of-growth classifications. Most often, we work with tilde approximations of the form g(N) ~ a f(N) where f(N) = N^b log^c N and refer to f(N) as the The order of growth of g(N). We use just a few structural primitives (statements, conditionals, loops, nesting, and method calls) to implement algorithms, so very often the order of growth of the cost is one of just a few functions of the problem size N.



description	order of growth	typical code framework	description	example
constant	1	`a = b + c;`	*statement*	*add two numbers*
logarithmic	$\log N$	[*see page 47*]	*divide in half*	*binary search*

linear	N	`double max = a[0];`
		`for (int i = 1; i < N; i++)`
		`   if (a[i] > max) max = a[i];`
			loop	*find the maximum*

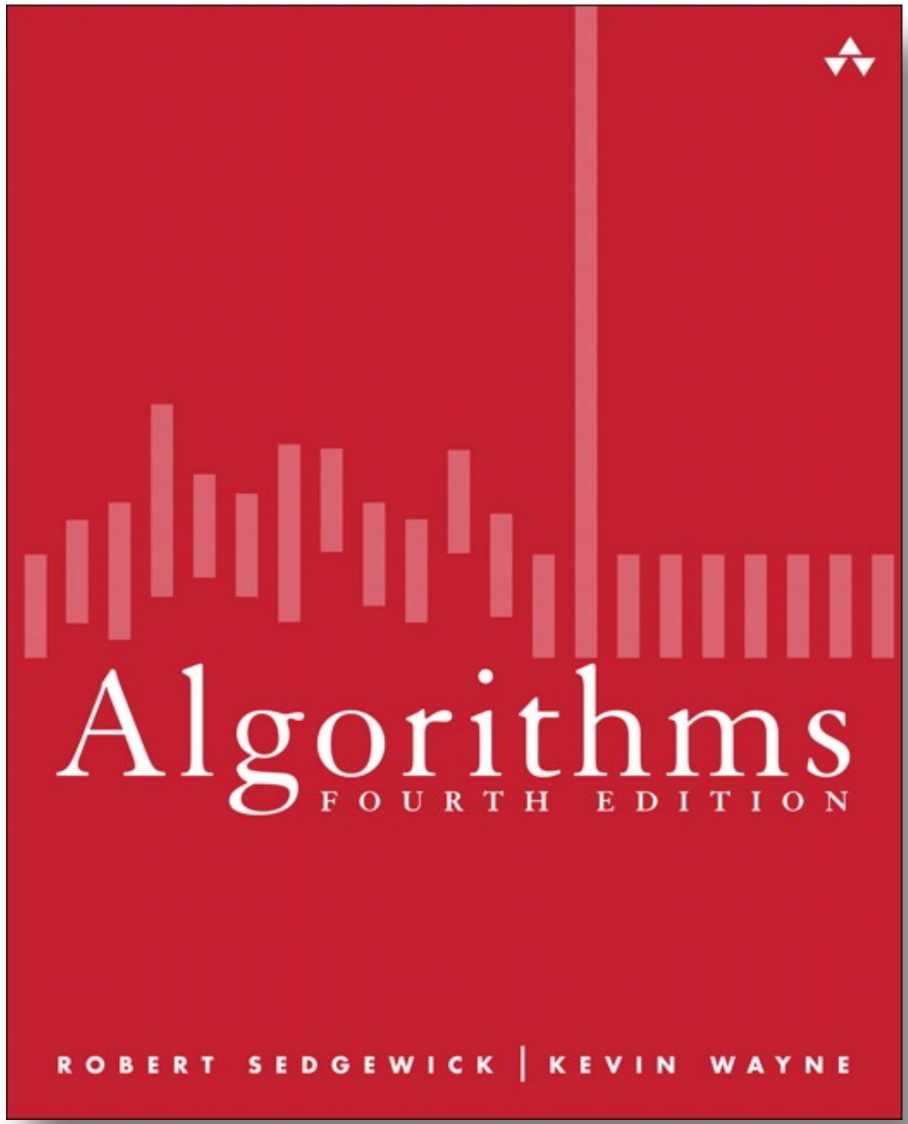
linearithmic	$N \log N$	[*see* ALGORITHM 2.4]	*divide and conquer*	*mergesort*

quadratic	N^2	`for (int i = 0; i < N; i++)`
		`   for (int j = i+1; j < N; j++)`
		`      if (a[i] + a[j] == 0)`
		`         cnt++;`
			double loop	*check all pairs*

cubic	N^3	`for (int i = 0; i < N; i++)`
		`   for (int j = i+1; j < N; j++)`
		`      for (int k = j+1; k < N; k++)`
		`         if (a[i] + a[j] + a[k] == 0)`
		`            cnt++;`
			triple loop	*check all triples*

exponential	2^N	[*see* CHAPTER 6]	*exhasutive search*	*check all subsets*

Lecture Overview



1.4 Analysis of Algorithms

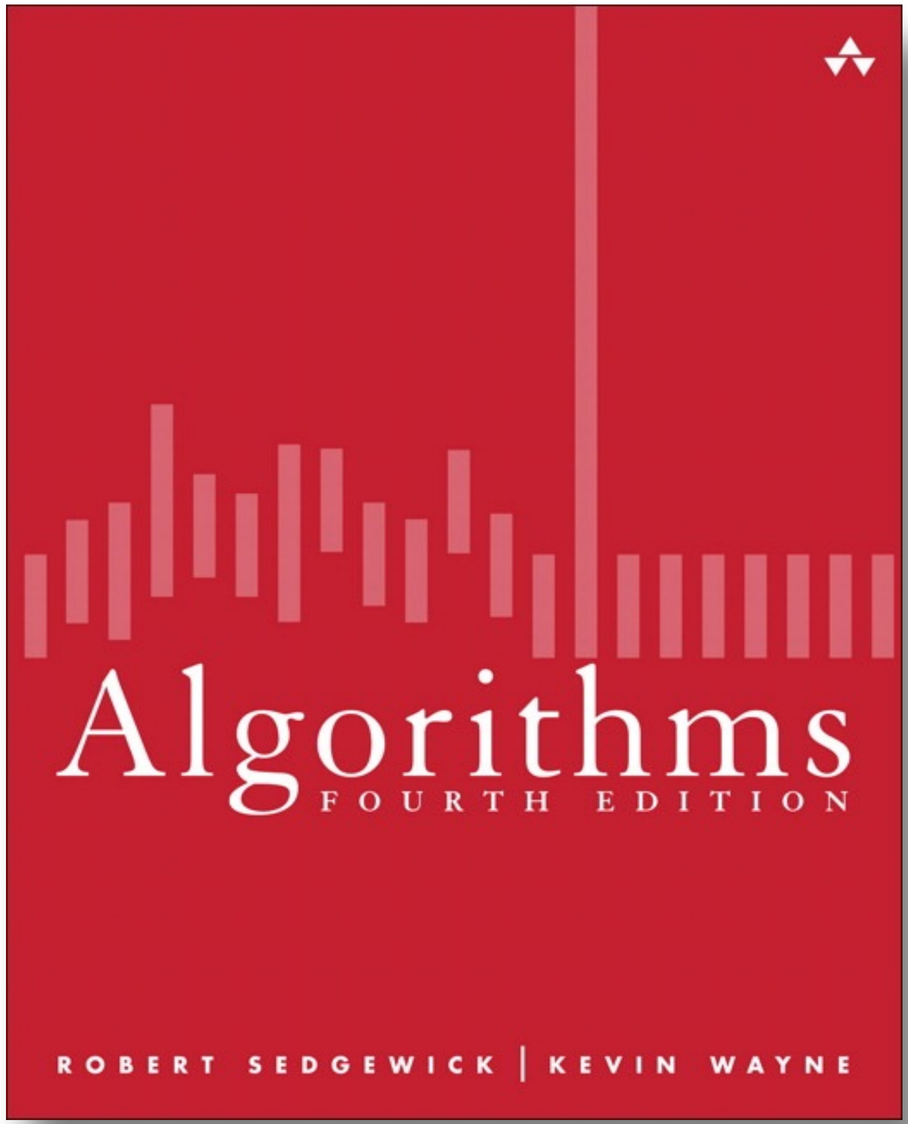
- Scientific method
- Observations
- Mathematical models
- Coping with dependence on inputs
- Memory usage

Coping with dependence on inputs

For many problems, the running time can vary widely depending on the input.

- Input models. We can carefully model the kind of input to be processed. This approach is challenging because the model may be unrealistic.
- Worst-case performance guarantees. Running time of a program is less than a certain bound (as a function of the input size), no matter what the input. Such a conservative approach might be appropriate for the software that runs a nuclear reactor or a pacemaker or the brakes in your car.
- Randomized algorithms. One way to provide a performance guarantee is to introduce randomness, e.g., quicksort and hashing. Every time you run the algorithm, it will take a different amount of time. These guarantees are not absolute, but the chance that they are invalid is less than the chance your computer will be struck by lightning. Thus, such guarantees are as useful in practice as worst-case guarantees.
- Amortized analysis. For many applications, the algorithm input might be not just data, but the sequence of operations performed by the client. Amortized analysis provides a worst-case performance guarantee on a sequence of operations.

Lecture Overview



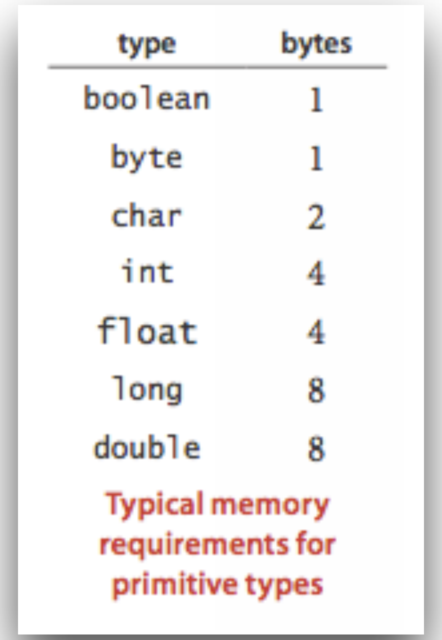
1.4 Analysis of Algorithms

- Scientific method
- Observations
- Mathematical models
- Coping with dependence on inputs
- Memory usage

Memory Usage

To estimate how much memory our program uses, we can count up the number of variables and weight them by the number of bytes according to their type. For a typical 64-bit machine,

- Primitive types. the following table gives the memory requirements for primitive types

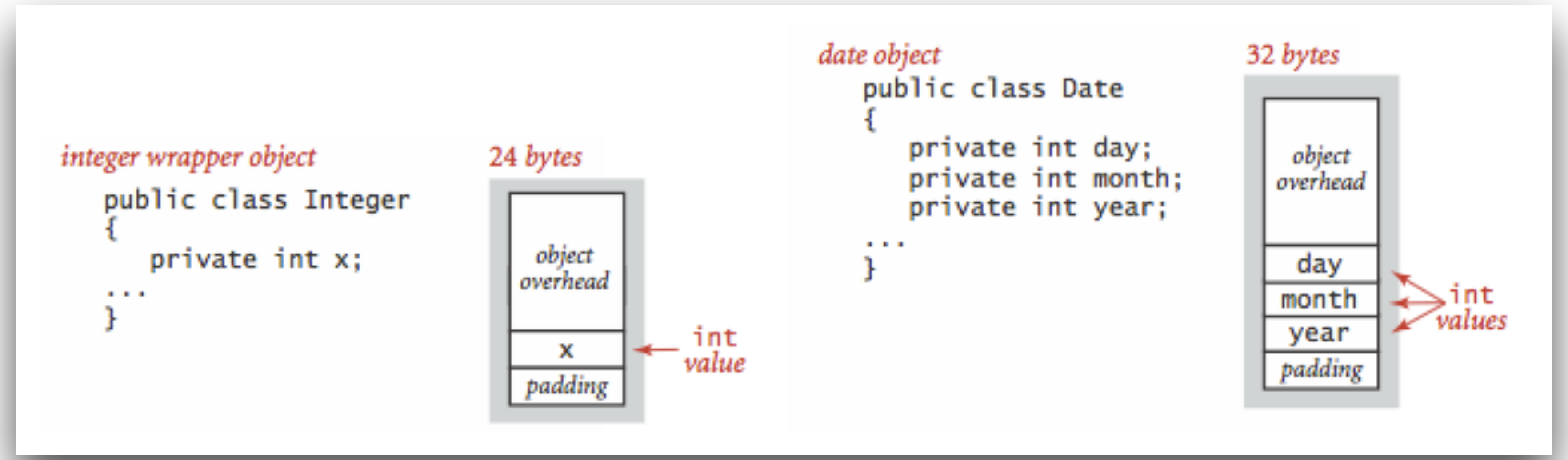


type	bytes
`boolean`	1
`byte`	1
`char`	2
`int`	4
`float`	4
`long`	8
`double`	8

Typical memory requirements for primitive types

Memory Usage

Objects. To determine the memory usage of an object, we add the amount of memory used by each instance variable to the overhead associated with each object, typically 16 bytes. Moreover, the memory usage is typically padded to be a multiple of 8 bytes (on a 64-bit machine).



integer wrapper object

```
public class Integer
{
   private int x;
...
}
```

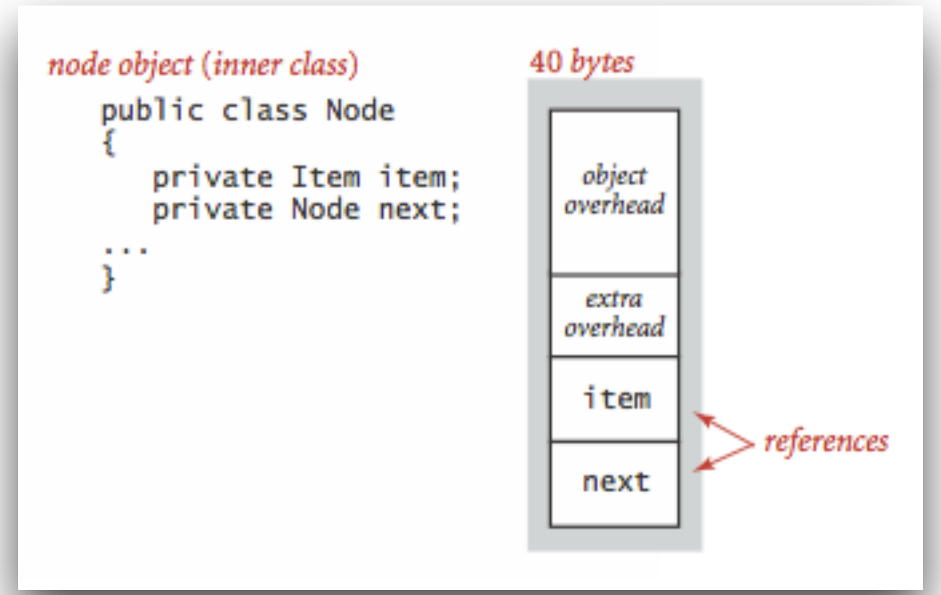
date object

```
public class Date
{
   private int day;
   private int month;
   private int year;
...
}
```

Memory Usage

References. A reference to an object typically is a memory address and thus uses 8 bytes of memory (on a 64-bit machine).

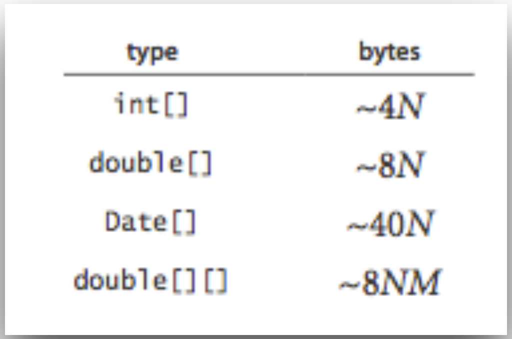
Linked lists. A nested non-static (inner) class such as our Node class requires an extra 8 bytes of overhead (for a reference to the enclosing instance).



```
public class Node
{
   private Item item;
   private Node next;
...
}
```

Memory Usage

Arrays. Arrays in Java are implemented as objects, typically with extra overhead for the length. An array of primitive-type values typically requires 24 bytes of header information (16 bytes of object overhead, 4 bytes for the length, and 4 bytes of padding) plus the memory needed to store the values.

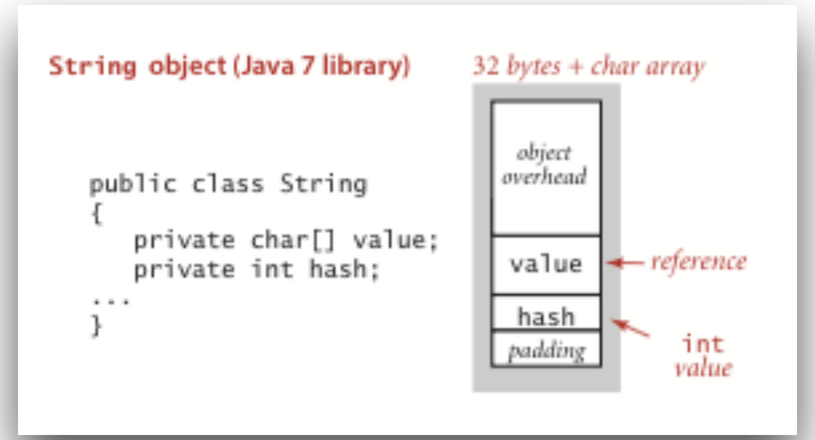


type	bytes
`int[]`	$\sim 4N$
`double[]`	$\sim 8N$
`Date[]`	$\sim 40N$
`double[][]`	$\sim 8NM$

Memory Usage

Strings. A Java 7 string of length N typically uses 32 bytes (for the String object) plus 24 + 2N bytes (for the array that contains the characters) for a total of 56 + 2N bytes.

Depending on context, we may or may not count the memory references by an object (recursively). For example, we count the memory for the char[] array in the memory for a String object because this memory is allocated when the string is created. But, we would not ordinarily count the memory for the String objects in a StackOfStrings object because the String objects are created by the client.

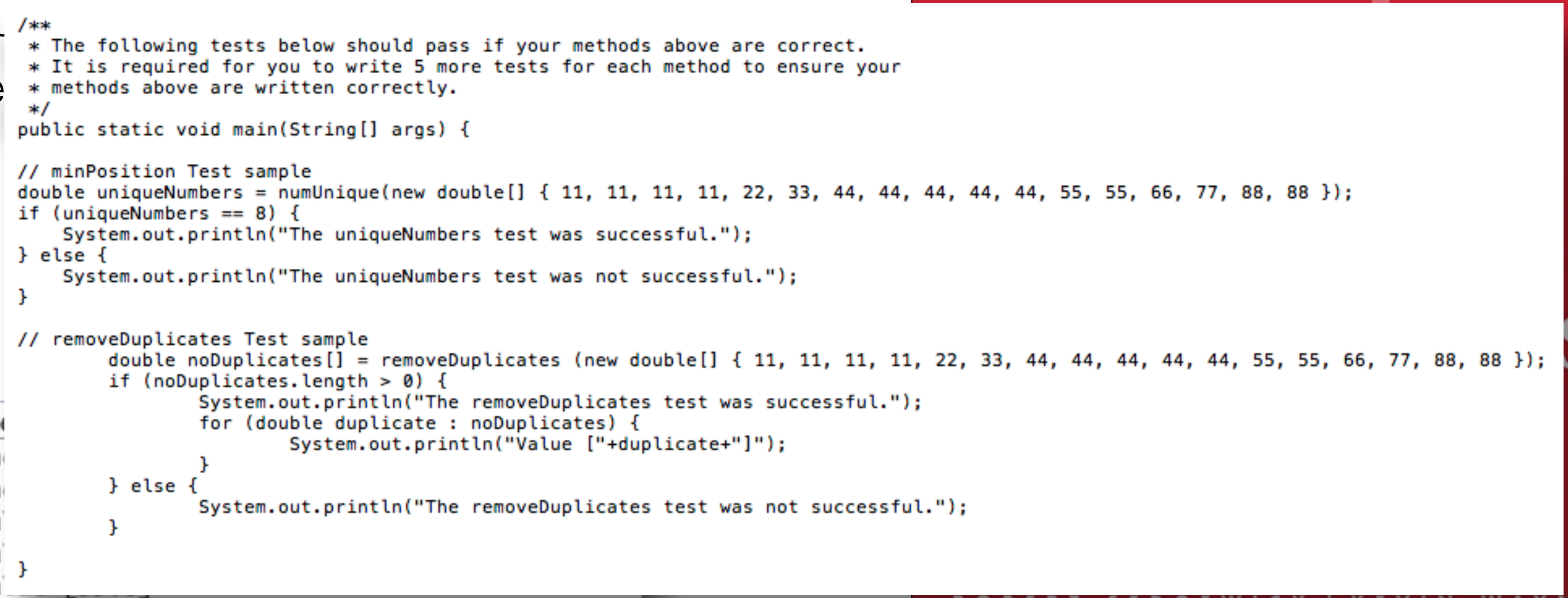


```
public class String
{
   private char[] value;
   private int hash;
...
}
```

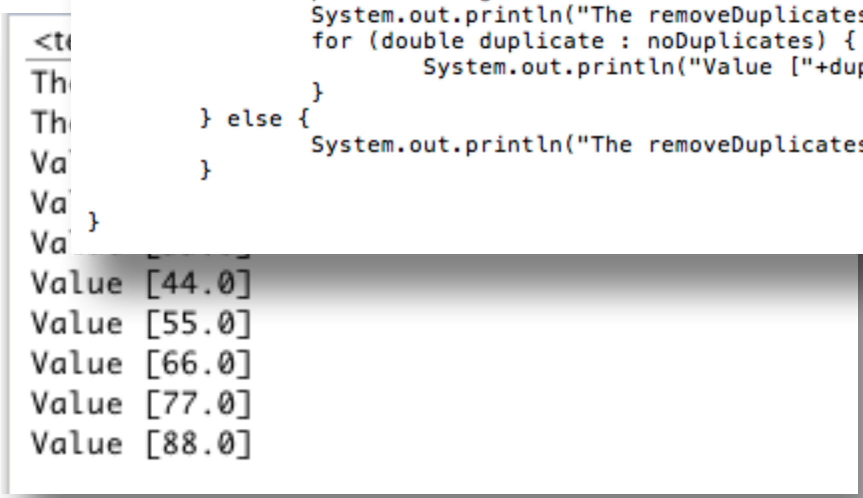
Assignment

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

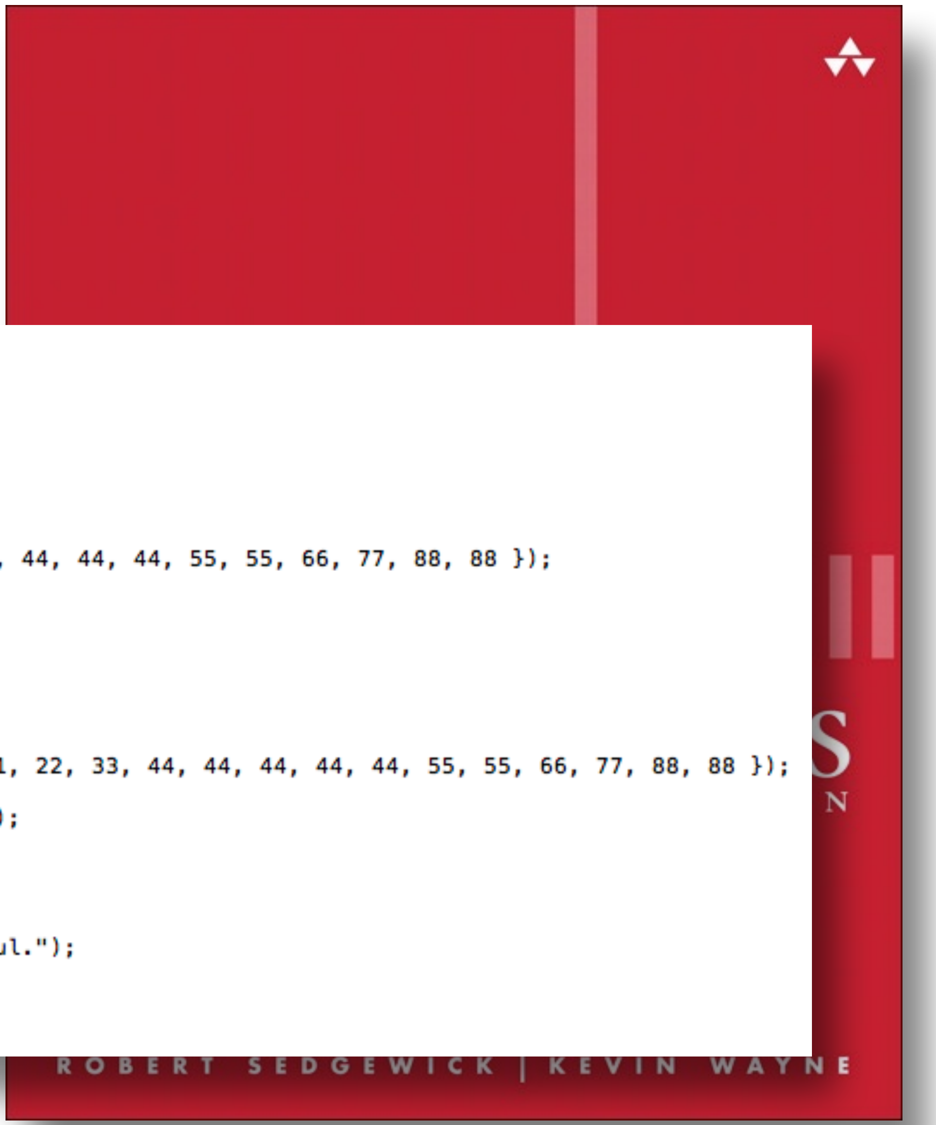
- nu
- re



```
/**
 * The following tests below should pass if your methods above are correct.
 * It is required for you to write 5 more tests for each method to ensure your
 * methods above are written correctly.
 */
public static void main(String[] args) {

// minPosition Test sample
double uniqueNumbers = numUnique(new double[] { 11, 11, 11, 11, 22, 33, 44, 44, 44, 44, 44, 55, 55, 66, 77, 88, 88 });
if (uniqueNumbers == 8) {
    System.out.println("The uniqueNumbers test was successful.");
} else {
    System.out.println("The uniqueNumbers test was not successful.");
}

// removeDuplicates Test sample
        double noDuplicates[] = removeDuplicates (new double[] { 11, 11, 11, 11, 22, 33, 44, 44, 44, 44, 44, 55, 55, 66, 77, 88, 88 });
        if (noDuplicates.length > 0) {
                System.out.println("The removeDuplicates test was successful.");
                for (double duplicate : noDuplicates) {
                        System.out.println("Value ["+duplicate+"]");
                }
        } else {
                System.out.println("The removeDuplicates test was not successful.");
        }

}
```



```
<te
Th
Th
Va
Va
Va
Value [44.0]
Value [55.0]
Value [66.0]
Value [77.0]
Value [88.0]
```



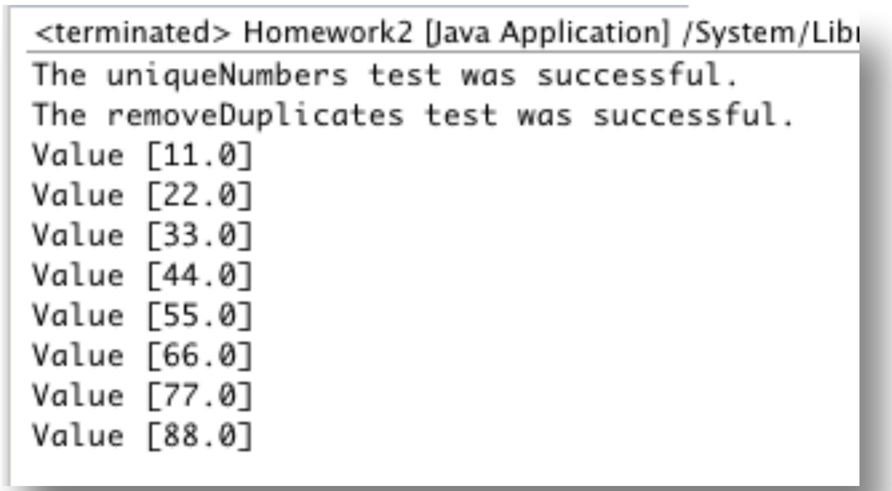
Assignment due next Monday at 11:59 PM

Assignment

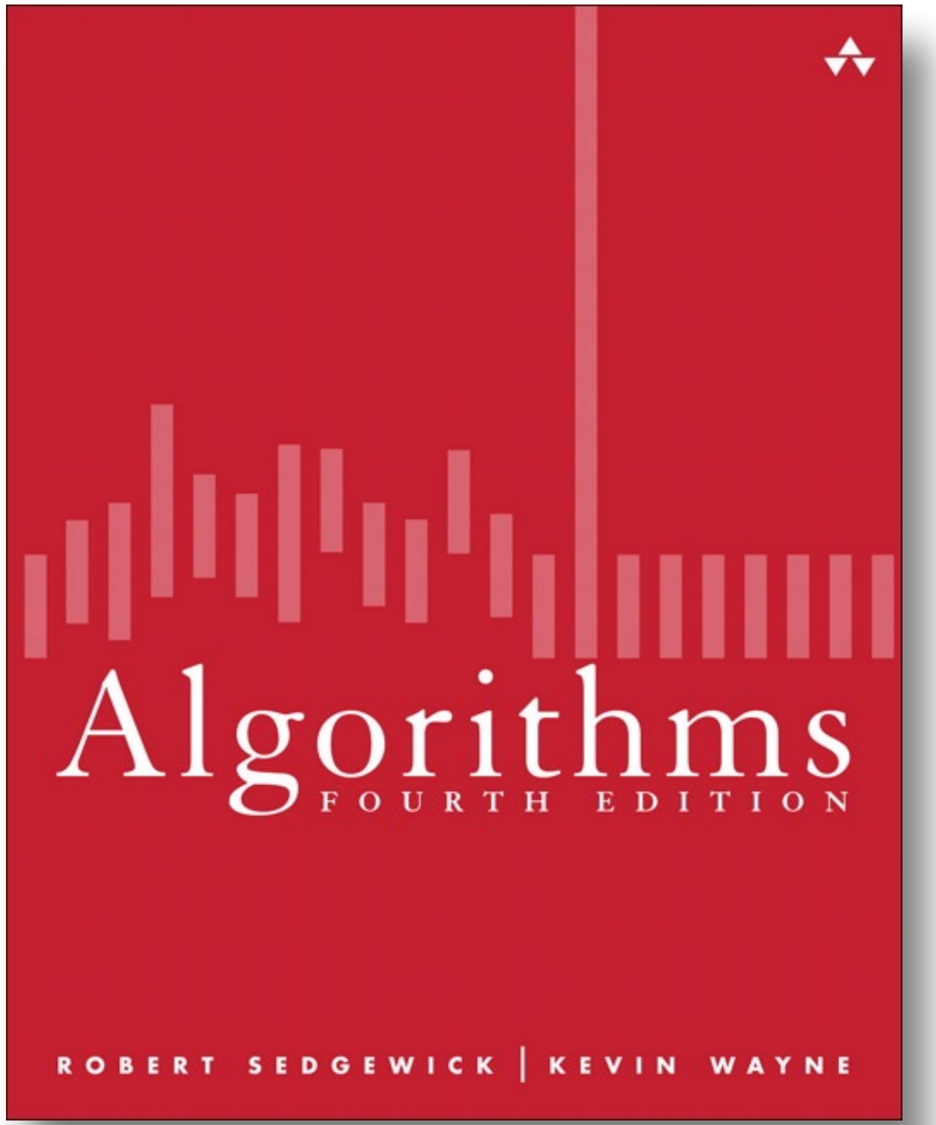
Modify the Homework2.java application to provide valid logic for the following methods:

- numUnique
- removeDuplicates

Sample Screenshot:



```
<terminated> Homework2 [Java Application] /System/Libr
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