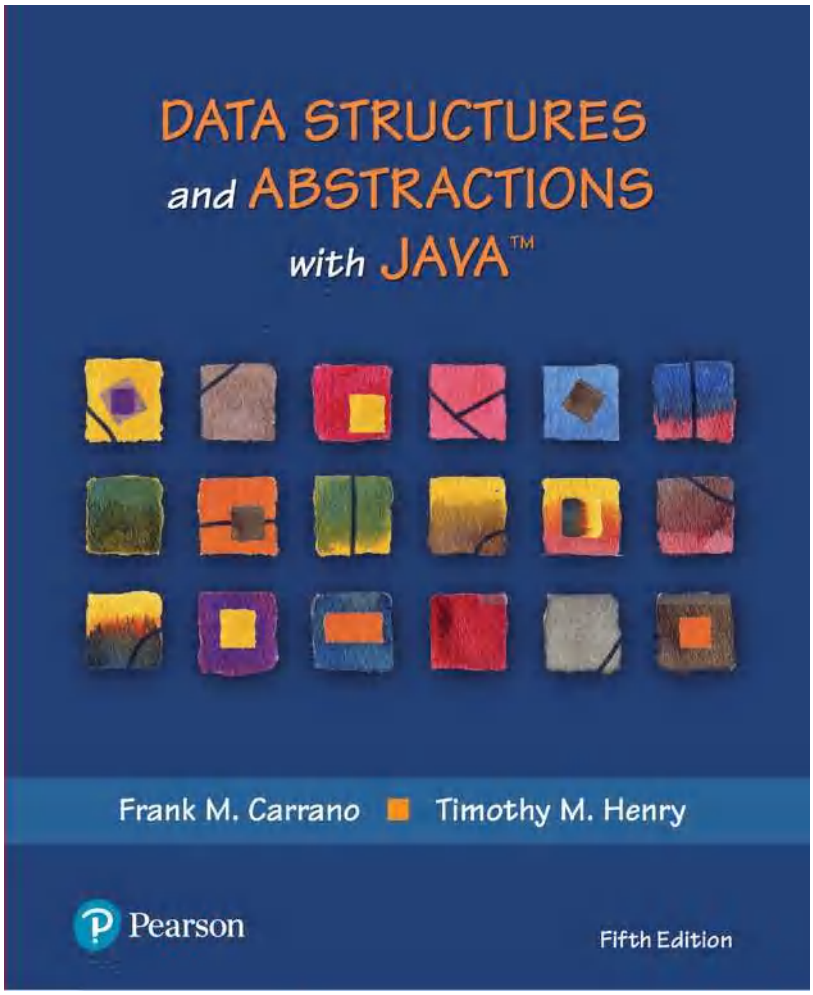
Solutions for Selected Exercises



Frank M. Carrano
University of Rhode Island

Timothy M. Henry
New England Institute of Technology

Charles Hoot
Oklahoma City University

Please send comments or errors to carrano@acm.org or timhenry@acm.org

Contents

(*Click on any entry below to locate the solutions for that chapter.*)

Prelude: Designing Classes 3
Chapter 1: Bags 5
Chapter 2: Bag Implementations That Use Arrays 9
Chapter 3: A Bag Implementation That Links Data 17
Chapter 4: The Efficiency of Algorithms 26
Chapter 5: Stacks 33
Chapter 6: Stack Implementations 37
Chapter 7: Queues, Deques, and Priority Queues 42
Chapter 8: Queue, Deque, and Priority Queue Implementations 49
Chapter 9: Recursion 56
Chapter 10: Lists 71
Chapter 11: List Implementations That Use Arrays 76
Chapter 12: A List Implementation That Links Data 83
Chapter 13: Iterators for the ADT List 94
Chapter 14: Problem Solving with Recursion 102
Chapter 15: An Introduction to Sorting 107
Chapter 16: Faster Sorting Methods 116
Chapter 17: Sorted Lists 122
Chapter 18: Inheritance and Lists 130
Chapter 19: Searching 133
Chapter 20: Dictionaries 141
Chapter 21: Dictionary Implementations 151
Chapter 22: Introducing Hashing 163
Chapter 23: Hashing as a Dictionary Implementation 168
Chapter 24: Trees 172
Chapter 25: Tree Implementations 180
Chapter 26: A Binary Search Tree Implementation 191
Chapter 27: A Heap Implementation 201
Chapter 28: Balanced Search Trees 206
Chapter 29: Graphs 214
Chapter 30: Graph Implementations 220

Prelude: Designing Classes

1. Consider the interface `NameInterface` defined in Segment P.13. We provided comments for only two of the methods. Write comments in `javadoc` style for each of the other methods.

```
/** Sets the first and last names.
    @param firstName  A string that is the desired first name.
    @param lastName   A string that is the desired last name. */
public void setName(String firstName, String lastName);

/** Gets the full name.
    @return  A string containing the first and last names. */
public String getName();

/** Sets the first name.
    @param firstName  A string that is the desired first name. */
public void setFirst(String firstName);

/** Gets the first name.
    @return  A string containing the first name. */
public String getFirst();

/** Sets the last name.
    @param lastName  A string that is the desired last name. */
public void setLast(String lastName);

/** Gets the last name.
    @return  A string containing the last name. */
public String getLast();

/** Changes the last name of the given Name object to the last name of this Name object.
    @param aName  A given Name object whose last name is to be changed. */
public void giveLastNameTo(NameInterface aName);

/** Gets the full name.
    @return  A string containing the first and last names. */
public String toString();
```

2. Consider the class `Circle` and the interface `Circular`, as given in Segments P.16 and P17.
   a. Is the client or the method `setRadius` responsible for ensuring that the circle's radius is positive?
   b. Write a precondition and a postcondition for the method `setRadius`.
   c. Write comments for the method `setRadius` in a style suitable for `javadoc`.
   d. Revise the method `setRadius` and its precondition and postcondition to change the responsibility mentioned in your answer to Part *a*.

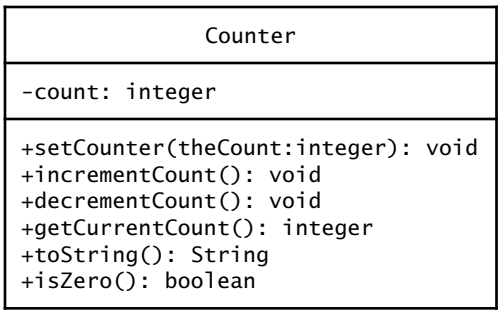
   a. The client is responsible for guaranteeing that the argument to the setRadius method is positive.
   b. Precondition: `newRadius` >= 0. Postcondition: The radius has been set to `newRadius`.
   c.
```
/** Sets the radius.
    @param newRadius  A non-negative real number. */
```

   d. Precondition: `newRadius` is the radius. Postcondition: The radius has been set to `newRadius` if `newRadius` >= 0.

```
/** Sets the radius.
    @param newRadius  A real number.
    @throws  ArithmeticException if newRadius < 0. */
public void setRadius(double newRadius) throws ArithmeticException
{
   if (newRadius < 0)
      throw new ArithmeticException("Radius was negative");
   else
      radius = newRadius;
  } // end setRadius
```

3. Write a CRC card and a class diagram for a proposed class called `Counter`. An object of this class will be used to count things, so it will record a count that is a nonnegative whole number. Include methods to set the counter to a given integer, to increase the count by 1, and to decrease the count by 1. Also include a method that returns the current count as an integer, a method `toString` that returns the current count as a string suitable for display on the screen, and a method that tests whether the current count is zero.

Counter
Responsibilities Set the counter to a value Add 1 to the counter Subtract 1 from the counter Get the value of the counter as an integer Get the value of the counter as a string Test whether the counter is zero Collaborations



4. Suppose you want to design software for a restaurant. Give use cases for placing an order and settling the bill. Identify a list of possible classes. Pick two of these classes, and write CRC cards for them.

System: Orders
Use case: Place an Order
Actor: Waitress
Steps:
1. Waitress starts a new order.
2. The waitress enters a table number.
3. Waitress chooses a menu item and adds it to the order.
   a. If there are more items, return to step 3.
4. The order is forwarded to the kitchen.

System: Orders
Use case: Settle Bill
Actor: Cashier
Steps:
1. The cashier enters the order id.
2. The system displays the total.
3. The customer makes a payment to the cashier.
4. The system computes any change due.
5. The cashier gives the customer a receipt.

Possible classes for this system are: Restaurant, Waitress, Cashier, Menu, MenuItem, Order, OrderItem, and Payment.

Chapter 1: Bags

1. Specify each method of the class `PiggyBank`, as given in Listing 1-3, by stating the method's purpose; by describing its parameters; and by writing preconditions, postconditions, and a pseudocode version of its header. Then write a Java interface for these methods that includes `javadoc`-style comments.

```
Purpose: Adds a given coin to this piggy bank.
Parameter: aCoin - a given coin
Precondition: None.
Postcondition: Either the coin has been added to the bank and the method returns true,
               or the method returns false because the coin could not be added to the bank.
public boolean add(aCoin)

Purpose: Removes a coin from this piggy bank.
Precondition: None.
Postcondition: The method returns either the removed coin or null in case the bank
               was empty before the method began execution.
public Coin remove()

Purpose: Detects whether this piggy bank is empty.
Precondition: None.
Postcondition: The method returns either true if the bank is empty or
               false if it is not empty.
public boolean isEmpty()
```

```
/**
    An interface that describes the operations of a piggy bank.
    @author Frank M. Carrano
    @version 4.0
*/
public interface PiggyBankInterface
{
   /** Adds a given coin to this piggy bank.
       @param aCoin  A given coin.
       @return  Either true if the coin has been added to the bank,
                or false if it has not been added. */
   public boolean add(Coin aCoin);

   /** Removes a coin from this piggy bank.
       @return  Either true if a coin has been removed from the bank,
                or false if it has not been removed. */
   public Coin remove();

   /** Detects whether this piggy bank is empty.
       @return  Either true if the bank is empty, or false if it not empty. */
   public boolean isEmpty();
} // end PiggyBankInterface
```

2. Suppose that `groceryBag` is a bag filled to its capacity with 10 strings that name various groceries. Write Java statements that remove and count all occurrences of `"soup"` in `groceryBag`. Do not remove any other strings from the bag. Report the number of times that `"soup"` occurred in the bag. Accommodate the possibility that `groceryBag` does not contain any occurrence of `"soup"`.

```
int soupCount = 0;
while (bag.remove("soup"))
   soupCount++;
System.out.println("Removed " + soupCount + " cans of soup.");
```

3. Given `groceryBag`, as described in Exercise 2, what effect does the operation `groceryBag.toArray()` have on `groceryBag`?

No effect; `groceryBag` is unchanged by the operation.

4. Given `groceryBag`, as described in Exercise 2, write some Java statements that create an array of the distinct strings that are in this bag. That is, if `"soup"` occurs three times in `groceryBag`, it should only appear once in your array. After you have finished creating this array, the contents of `groceryBag` should be unchanged.

```
Object[] items = groceryBag.toArray();
BagInterface<String> tempBag = new Bag<>(items.length);
for (Object anItem: items)
{
   String aString = anItem.toString();
   if (!tempBag.contains(aString))
      tempBag.add(aString);
} // end for
items = tempBag.toArray();
```

5. The *union* of two collections consists of their contents combined into a new collection. Add a method `union` to the interface `BagInterface` for the ADT bag that returns as a new bag the union of the bag receiving the call to the method and the bag that is the method's one argument. Include sufficient comments to fully specify the method.

   Note that the union of two bags might contain duplicate items. For example, if object *x* occurs five times in one bag and twice in another, the union of these bags contains *x* seven times. Specifically, suppose that `bag1` and `bag2` are `Bag` objects, where `Bag` implements `BagInterface`; `bag1` contains the `String` objects `a`, `b`, and `c`; and `bag2` contains the `String` objects `b`, `b`, `d`, and `e`. After the statement

```
BagInterface<String> everything = bag1.union(bag2);
```

   executes, the bag `everything` contains the strings a, b, b, b, c, d, and e. Note that union does not affect the contents of `bag1` and `bag2`.

```
/** Creates a new bag that combines the contents of this bag and a
    second given bag without affecting the original two bags.
    @param anotherBag  The given bag.
    @return  A bag that is the union of the two bags. */
public BagInterface<T> union(BagInterface<T> anotherBag);
```

6. The *intersection* of two collections is a new collection of the entries that occur in both collections. That is, it contains the overlapping entries. Add a method `intersection` to the interface `BagInterface` for the ADT bag that returns as a new bag the intersection of the bag receiving the call to the method and the bag that is the method's one argument. Include sufficient comments to fully specify the method.

   Note that the intersection of two bags might contain duplicate items. For example, if object *x* occurs five times in one bag and twice in another, the intersection of these bags contains *x* twice. Specifically, suppose that `bag1` and `bag2` are `Bag` objects, where `Bag` implements `BagInterface`; `bag1` contains the `String` objects `a`, `b`, and `c`; and `bag2` contains the `String` objects `b`, `b`, `d`, and `e`. After the statement

```
BagInterface<String> commonItems = bag1.intersection(bag2);
```

   executes, the bag `commonItems` contains only the string `b`. If `b` had occurred in `bag1` twice, `commonItems` would have contained two occurrences of `b`, since `bag2` also contains two occurrences of `b`. Note that `intersection` does not affect the contents of `bag1` and `bag2`.

```
/** Creates a new bag that contains those objects that occur in both this
    bag and a second given bag without affecting the original two bags.
    @param anotherBag  The given bag.
    @return  A bag that is the intersection of the two bags. */
public BagInterface<T> intersection(BagInterface<T> anotherBag);
```

7. The *difference* of two collections is a new collection of the entries that would be left in one collection after removing those that also occur in the second. Add a method `difference` to the interface `BagInterface` for the ADT bag that returns as a new bag the difference of the bag receiving the call to the method and the bag that is the method's one argument. Include sufficient comments to fully specify the method.

   Note that the difference of two bags might contain duplicate items. For example, if object *x* occurs five times in one bag and twice in another, the difference of these bags contains *x* three times. Specifically, suppose that `bag1` and `bag2` are `Bag` objects, where `Bag` implements `BagInterface`; `bag1` contains the `String` objects `a`, `b`, and `c`; and `bag2` contains the `String` objects `b`, `b`, `d`, and `e`. After the statement

```
BagInterface leftOver1 = bag1.difference(bag2);
```

   executes, the bag `leftOver1` contains the strings `a` and `c`. After the statement

```
BagInterface leftOver2 = bag2.difference(bag1);
```

   executes, the bag `leftOver2` contains the strings `b`, `d`, and `e`. Note that `difference` does not affect the contents of `bag1` and `bag2`.

```
/** Creates a new bag of objects that would be left in this bag
    after removing those that also occur in a second given bag
    without affecting the original two bags.
    @param anotherBag  The given bag.
    @return  A bag that is the difference of the two bags. */
public BagInterface<T> difference(BagInterface<T> anotherBag);
```

8. Write code that accomplishes the following tasks: Consider two bags that can hold strings. One bag is named letters and contains several one-letter strings. The other bag is empty and is named vowels. One at a time, remove a string from letters. If the string contains a vowel, place it into the bag vowels; otherwise, discard the string. After you have checked all of the strings in letters, report the number of vowels in the bag vowels and the number of times each vowel appears in the bag.

```
BagInterface<String> allVowels = new Bag<>();
allVowels.add("a");
allVowels.add("e");
allVowels.add("i");
allVowels.add("o");
allVowels.add("u");
BagInterface<String> vowels = new Bag<>();

while (!letters.isEmpty())
{
   String aLetter = letters.remove();
   if (allVowels.contains(aLetter))
      vowels.add(aLetter);
} // end while

System.out.println("There are " + vowels.getCurrentSize() + " vowels in the bag.");
String[] vowelsArray = {"a", "e", "i", "o", "u"};
for (int index = 0; index < vowelsArray.length; index++)
{
   int count = vowels.getFrequencyOf(vowelsArray[index]);
   System.out.println(vowelsArray[index] + " occurs " + count + " times.");
} // end for
```

9. Write code that accomplishes the following tasks: Consider three bags that can hold strings. One bag is named `letters` and contains several one-letter strings. Another bag is named `vowels` and contains five strings, one for each vowel. The third bag is empty and is named `consonants`. One at a time, remove a string from `letters`. Check whether the string is in the bag `vowels`. If it is, discard the string. Otherwise, place it into the bag `consonants`. After you have checked all of the strings in `letters`, report the number of consonants in the bag `consonants` and the number of times each consonant appears in the bag.

```
while (!letters.isEmpty())
{
   String aLetter = letters.remove();
   if (!vowels.contains(aLetter))
      consonants.add(aLetter);
} // end while

System.out.println("There are " + consonants.getCurrentSize() +
                   " consonants in the bag.");
final String[] CONSONANTS = {"a", "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
                             "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z"};
for (int index = 0; index < CONSONANTS.length; index++)
{
   int count = consonants.getFrequencyOf(CONSONANTS[index]);
   System.out.println(CONSONANTS[index] + " occurs " + count + " times.");
} // end for
```

Chapter 2: Bag Implementations That Use Arrays

1. Why are the methods `getIndexOf` and `removeEntry` in the class `ArrayBag` private instead of public?

   The methods are implementation details that should be hidden from the client. They are not ADT bag operations and are not declared in `BagInterface`. Thus, they should not be public methods.

2. Implement a method `replace` for the ADT bag that replaces and returns a specified object currently in a bag with a given object.

```
/** Replaces an unspecified entry in this bag with a given object.
    @param replacement  The given object.
    @return  The original entry in the bag that was replaced. */
public T replace(T replacement)
{
   T replacedEntry = bag[numberOfEntries - 1];
   bag[numberOfEntries - 1] = replacement;
   return replacedEntry;
} // end replace
```

3. Revise the definition of the method `clear`, as given in Segment 2.23, so that it is more efficient and calls only the method `checkIntegrity`.

```
public void clear()
{
   checkIntegrity();
   for (int index = 0; index < numberOfEntries; index++)
     bag[index] = null;
   numberOfEntries = 0;
} // end clear
```

4. Revise the definition of the method `remove`, as given in Segment 2.24, so that it removes a random entry from a bag. Would this change affect any other method within the class `ArrayBag`?

   Begin the file containing `ArrayBag` with the following statement:

```
import java.util.Random;
```

   Add the following data field to `ArrayBag`:

```
private Random generator;
```

   Add the following statement to the initializing constructor of `ArrayBag`:

```
generator = new Random();
```

   The definition of the method `remove` follows:

```
public T remove()
{
   T result = removeEntry(generator.nextInt(numberOfEntries));
   return result;
} // end remove
```

5. Define a method `removeEvery` for the class `ArrayBag` that removes all occurrences of a given entry from a bag.

The following method is easy to write, but it is inefficient since it repeatedly begins the search from the beginning of the array `bag`:

```
/** Removes every occurrence of a given entry from this bag.
    @param anEntry  The entry to be removed. */
public void removeEvery(T anEntry)
{
   int index = getIndexOf(anEntry);
   while (index > -1)
   {
      T result = removeEntry(index); // removeEntry is a private method in ArrayBag
      index = getIndexOf(anEntry);
   } // end while
} // end removeEvery
```

The following method continues the search from the last found entry, so it is more efficient. But it is easy to make a mistake while coding:

```
public void removeEvery2(T anEntry)
{
   for (int index = 0; index < numberOfEntries; index++)
   {
      if (anEntry.equals(bag[index]))
      {
         removeEntry(index);
         // Since entries in array bag are shifted, index can remain the same;
         // but the for statement will increment index, so need to decrement it here:
         index--;
      } // end if
   } // end for
} // end removeEvery2
```

6. An instance of the class `ArrayBag` has a fixed size, whereas an instance of `ResizableArrayBag` does not. Give some examples of situations where a bag would be appropriate if its size is: a. Fixed; b. Resizable.

a. Simulating any application involving an actual bag, such a grocery bag.
b. Maintaining any collection that can grow in size or whose eventual size is unknown.

7. Suppose that you wanted to define a class `PileOfBooks` that implements the interface described in Project 2 of the previous chapter. Would a bag be a reasonable collection to represent the pile of books? Explain.

No. The books in a pile have an order. A bag does not order its entries.

8. Consider an instance `myBag` of the class `ResizableArrayBag`, as discussed in Segments 2.36 to 2.40. Suppose that the initial capacity of `myBag` is 10. What is the length of the array `bag` after
a. Adding 145 entries to `myBag`?
b. Adding an additional 20 entries to `myBag`?

a. 160. During the 11th addition, the bag doubles in size to 20. At the 21st addition, the bag's size increases to 40. At the 41st addition, it doubles in size again to 80. At the 81st addition, the size becomes 160 and stays that size during the addition of the 145th entry.
b. 320. The array can accommodate 160 entries. Since it contains 145 entries, it can accommodate 15 more before having to double in size again.

9. Consider a method that accepts as its argument an instance of the class `ArrayBag` and returns an instance of the class `ResizableArrayBag` that contains the same entries as the argument bag. Define this method

   a. Within `ArrayBag`.

   b. Within `ResizableArrayBag`.

   c. Within a client of `ArrayBag` and `ResizableArrayBag`.

```
public static ResizableArrayBag<String> convertToResizable(ArrayBag<String> aBag)
{
   ResizableArrayBag<String> newBag = new ResizableArrayBag<>();

   Object[] bagArray = aBag.toArray();
   for (int index = 0; index < bagArray.length; index++)
      newBag.add((String)bagArray[index]);

   return newBag;
} // end convertToResizable
```

10. Suppose that a bag contains `Comparable` objects such as strings. A `Comparable` object belongs to a class that implements the standard interface `Comparable<T>`, and so has the method `compareTo`. Implement the following methods for the class `ArrayBag`:

- The method `getMin` that returns the smallest object in a bag
- The method `getMax` that returns the largest object in a bag
- The method `removeMin` that removes and returns the smallest object in a bag
- The method `removeMax` that removes and returns the largest object in a bag

Students might have trouble with this exercise, depending on their knowledge of Java. The necessary details aren't covered until Java Interlude 3. You might want to ask for a pseudocode solution instead of a Java method.

Change the header of `BagInterface` to

```
public interface BagInterface<T extends Comparable<? super T>>
```

Change the header of `ArrayBag` to

```
public class ArrayBag<T extends Comparable<? super T>> implements BagInterface<T>
```

Allocate the array `tempBag` in the constructor of `ArrayBag` as follows:

```
T[] tempBag = (T[])new Comparable<?>[desiredCapacity];
```

Allocate the array `result` in the method `toArray` as follows:

```
T[] result = (T[])new Comparable<?>[numberOfEntries];
```

The required methods follow:

```
/** Gets the smallest value in this bag.
    @returns  A reference to the smallest object, or null if the bag is empty. */
public T getMin()
{
   if (isEmpty())
      return null;
   else
      return bag[getIndexOfMin()];
} // end getMin

// Returns the index of the smallest in this bag.
// Precondition: The bag is not empty.
private int getIndexOfMin()
{
   int indexOfSmallest = 0;
   for (int index = 1; index < numberOfEntries; index++)
   {
```

```
      if (bag[index].compareTo(bag[indexOfSmallest]) < 0)
         indexOfSmallest = index;
   } // end for

   return indexOfSmallest;
} // end getIndexOfMin

/** Gets the largest value in this bag.
    @returns  A reference to the largest object, or null if the bag is empty */
public T getMax()
{
   if (isEmpty())
      return null;
   else
      return bag[getIndexOfMax()];
} // end getMax

// Returns the index of the largest value in this bag.
// Precondition: The bag is not empty.
private int getIndexOfMax()
{
   int indexOfLargest = 0;
   for (int index = 1; index < numberOfEntries; index++)
   {
      if (bag[index].compareTo(bag[indexOfLargest]) > 0)
         indexOfLargest = index;
   } // end for
   return indexOfLargest;
} // end getIndexOfMax

/** Removes the smallest value in this bag.
    @returns  A reference to the removed (smallest) object,
              or null if the bag is empty. */
public T removeMin()
{
   if (isEmpty())
      return null;
   else
   {
      int indexOfMin = getIndexOfMin();
      T smallest = bag[indexOfMin];
      removeEntry(indexOfMin);
      return smallest;
   } // end if
} // end removeMin

/** Removes the largest value in this bag.
    @returns  A reference to the removed (largest) object,
              or null if the bag is empty. */
public T removeMax()
{
   if (isEmpty())
      return null;
   else
   {
      int indexOfMax = getIndexOfMax();
      T largest = bag[indexOfMax];
      removeEntry(indexOfMax);
      return largest;
   } // end if
} // end removeMax
```

11. Suppose that a bag contains `Comparable` objects, as described in the previous exercise. Define a method for the class `ArrayBag` that returns a new bag of items that are less than some given item. The header of the method could be as follows:

```
public BagInterface<T> getAllLessThan(Comparable<T> anObject)
```

Make sure that your method does not affect the state of the original bag.

See the note in the solution to Exercise 10 about student background.

```
/** Creates a new bag of objects that are in this bag and are less than a given object.
    @param  anObject  A given object.
    @return  A new bag of objects that are in this bag and are less than anObject. */
public BagInterface<T> getAllLessThan(Comparable<T> anObject)
{
   BagInterface<T> result = new ArrayBag<>();

   for (int index = 0; index < numberOfEntries; index++)
   {
      if (anObject.compareTo(bag[index]) > 0)
         result.add(bag[index]);
   } // end for

   return result;
} // end getAllLessThan
```

12. Define an `equals` method for the class `ArrayBag` that returns true when the contents of two bags are the same. Note that two equal bags contain the same number of entries, and each entry occurs in each bag the same number of times. The order of the entries in each array is irrelevant.

```
public boolean equals(Object other)
{
   boolean result = false;
   if (other instanceof ArrayBag)
   {
      // The cast is safe here
      @SuppressWarnings("unchecked")
      ArrayBag<T> otherBag = (ArrayBag<T>)other;
      int otherBagLength = otherBag.getCurrentSize();
      if (numberOfEntries == otherBagLength) // Bags must contain the same number of objects
      {
         result = true;                      // Assume equal
         for (int index = 0; (index < numberOfEntries) && result; index++)
         {
            T thisBagEntry = bag[index];
            T otherBagEntry = otherBag.bag[index];
            if (!thisBagEntry.equals(otherBagEntry))
               result = false;               // Bags have unequal entries
         } // end for
      } // end if
   // Else bags have unequal number of entries
   } // end if

   return result;
} // end equals
```

13. The class `ResizableArrayBag` has an array that can grow in size as objects are added to the bag. Revise the class so that its array also can shrink in size as objects are removed from the bag. Accomplishing this task will require two new private methods, as follows:

- The first new method checks whether we should reduce the size of the array:

  ```
  private boolean isTooBig()
  ```

  This method returns true if the number of entries in the bag is less than half the size of the array and the size of the array is greater than 20.

- The second new method creates a new array that is three quarters the size of the current array and then copies the objects in the bag to the new array:

  ```
  private void reduceArray()
  ```

Implement each of these two methods, and then use them in the definitions of the two `remove` methods.

```
private boolean isTooBig()
{
   return (numberOfEntries < bag.length / 2) && (bag.length > 20);
} // end isTooBig

private void reduceArray()
{
   T[] oldBag = bag;                                  // Save reference to array
   int oldSize = oldBag.length;                       // Save old max size of array

   @SuppressWarnings("unchecked")
   T[] tempBag = (T[])new Object[3 * oldSize / 4]; // Reduce size of array; unchecked cast
   bag = tempBag;

   // Copy entries from old array to new, smaller array
   for (int index = 0; index < numberOfEntries; index++)
      bag[index] = oldBag[index];
} // end reduceArray

public T remove()
{
   T result = removeEntry(numberOfEntries - 1);
   if (isTooBig())
      reduceArray();

   return result;
} // end remove

public boolean remove(T anEntry)
{
   int index = getIndexOf(anEntry);
   T result = removeEntry(index);

   if (isTooBig())
      reduceArray();

   return anEntry.equals(result);
} // end remove
```

14. Consider the two private methods described in the previous exercise.
    a. The method `isTooBig` requires the size of the array to be greater than 20. What problem could occur if this requirement is dropped?
    b. The method `reduceArray` is not analogous to the method `doubleCapacity` in that it does not reduce the size of the array by one half. What problem could occur if the size of the array is reduced by one half instead of three quarters?

    a. If the size of the array is less than 20, it will need to be resized after very few additions or removals. Since 20 is not very large, the amount of wasted space will be negligible.
    b. If the size of the array is reduced by half, a sequence of alternating removes and adds can cause a resize with each operation.

15. Define the method `union`, as described in Exercise 5 of Chapter 1, for the class `ResizableArrayBag`.

```
public BagInterface<T> union(BagInterface<T> anotherBag)
{
   BagInterface<T> unionBag = new ResizableArrayBag<>();
   ResizableArrayBag<T> otherBag = (ResizableArrayBag<T>)anotherBag;

   int index;

   // Add entries from this bag to the new bag
   for (index = 0; index < numberOfEntries; index++)
      unionBag.add(bag[index]);

   // Add entries from the second bag to the new bag
   for (index = 0; index < otherBag.getCurrentSize(); index++)
      unionBag.add(otherBag.bag[index]);

   return unionBag;
} // end union
```

16. Define the method `intersection`, as described in Exercise 6 of the previous chapter, for the class `ResizableArrayBag`.

```
public BagInterface<T> intersection(BagInterface<T> anotherBag)
{
   // The count of an item in the intersection is the smaller of the count in each bag.
   BagInterface<T> intersectionBag = new ResizableArrayBag<>();
   ResizableArrayBag<T> otherBag = (ResizableArrayBag<T>)anotherBag;
   BagInterface<T> copyOfAnotherBag = new ResizableArrayBag<>()

   int index;

   // Copy the second bag
   for (index = 0; index < otherBag.numberOfEntries; index++)
      copyOfAnotherBag.add(otherBag.bag[index]);

   // Add to intersectionBag each item in this bag that matches an item in anotherBag;
   // once matched, remove it from the second bag

   for (index = 0; index < getCurrentSize(); index++)
   {
      if (copyOfAnotherBag.contains(bag[index]))
      {
         intersectionBag.add(bag[index]);
         copyOfAnotherBag.remove(bag[index]);
      } // end if
   } // end for

   return intersectionBag;
} // end intersection
```

17. Define the method difference, as described in Exercise 7 of the previous chapter, for the class ResizableArrayBag.

```
public BagInterface<T> difference(BagInterface<T> anotherBag)
{
   // The count of an item in the difference is the difference of the counts in the two bags.
   BagInterface<T> differenceBag = new ResizableArrayBag<>();
   ResizableArrayBag<T> otherBag = (ResizableArrayBag<T>)anotherBag;

   int index;

   // copy this bag
   for (index = 0; index < numberOfEntries; index++)
   {
      differenceBag.add(bag[index]);
   } // end for

   // remove the ones that are in anotherBag
   for (index = 0; index < otherBag.getCurrentSize(); index++)
   {
      if (differenceBag.contains(otherBag.bag[index]))
      {
         differenceBag.remove(otherBag.bag[index]);
      } // end if
   } // end for

   return differenceBag;
} // end difference
```

18. Write a Java program to play the following game. Randomly select six of the cards Ace, Two, Three, . . . , Jack, Queen, and King of clubs. Place the six cards into a bag. One at a time, each player guesses which card is in the bag. If the guess is correct, you remove the card from the bag and give it to the player. When the bag is empty, the player with the most cards wins.

Chapter 3: A Bag Implementation That Links Data

1. Add a constructor to the class `LinkedBag` that creates a bag from a given array of objects.

```
public LinkedBag(T[] items, int numberOfitems)
{
   this();

   for (int index = 0; index < numberOfitems; index++)
      add(items[index]);
} // end constructor
```

2. Consider the definition of `LinkedBag`'s `add` method that appears in Segment 3.12. Interchange the second and third statements in the method's body, as follows:

```
firstNode = newNode;
newNode.next = firstNode;
```

   a. What is displayed by the following statements in a client of the modified `LinkedBag`?

```
BagInterface<String> myBag = new LinkedBag<>();
myBag.add("30");
myBag.add("40");
myBag.add("50");
myBag.add("10");
myBag.add("60");
myBag.add("20");
int numberOfEntries = myBag.getCurrentSize();
Object[] entries = myBag.toArray();
for (int index = 0; index < numberOfEntries; index++)
   System.out.print(entries[index] + " ");
```

   b. What methods, if any, in `LinkedBag` could be affected by this change to the method `add` when they execute? Why?

   a. 20 20 20 20 20 20
   b. The change to the `add` method causes `add` to create a one-node chain containing the last entry added to the bag. However, `numberOfEntries` count the numbers of additions, which is 6 in this case. Other methods execute using the incorrect contents of the bag.

3. Repeat Exercise 2 Chapter 2 for the class `LinkedBag`.
   Implement a method `replace` for the ADT bag that replaces and returns any object currently in a bag with a given object.

```
/** Replaces an unspecified entry in this bag with a given object.
    @param replacement  The given object.
    @return  The original entry in the bag that was replaced. */
public T replace(T replacement)
{
   T replacedEntry = firstNode.data;
   firstNode.data = replacement;

   return replacedEntry;
} // end replace
```

4. Revise the definition of the method `remove`, as given in Segment 3.21, so that it removes a random entry from a bag. Would this change affect any other method within the class `LinkedBag`?

Begin the file containing `LinkedBag` with the following statement:

```
import java.util.Random;
```

Add the following data field to `LinkedBag`:

```
private Random generator;
```

Add the following statement to the initializing constructor of `ArrayBag`:

```
generator = new Random();
```

Besides this change to the constructor, no other method must change. Although the method `clear` calls this `remove` method, `clear` will still work correctly.

One definition of the method `remove` follows:

```
/** Removes one random entry from this bag, if possible.
    @return  Either the removed entry, if the removal was successful, or null. */
public T remove()
{
   int randomEntryNumber = generator.nextInt(numberOfEntries); // 0 to numberOfEntries - 1

   T entryToRemove = null;
   if (!isEmpty())
   {
      Node searcher = firstNode;
      for (int counter = 0; counter < randomEntryNumber; counter++)
         searcher = searcher.next;
      entryToRemove = searcher.data;
      remove(entryToRemove);
   } // end if
   return entryToRemove;
} // end remove
```

This method locates the n^{th} node in the chain, where n is a random integer. After getting the entry in this node, the method calls the second `remove` method to remove it from the bag. Unfortunately, that `remove` method must locate the node that contains the entry. To avoid this repeated search, we replace the call to `remove` with the statements in that method that effect the deletion of the desired entry, as follows:

```
public T remove()
   {
      int randomEntryNumber = generator.nextInt(numberOfEntries); // 0 to numberOfEntries - 1

      T entryToRemove = null;
      if (!isEmpty())
      {
         Node searcher = firstNode;
         for (int counter = 0; counter < randomEntryNumber; counter++)
            searcher = searcher.next;
         entryToRemove = searcher.data;

         searcher.data = firstNode.data; // Replace located entry with entry in first node
         firstNode = firstNode.next;     // Remove first node
         numberOfEntries--;
      } // end if
      return entryToRemove;
   } // end remove
```

5. Define a method `removeEvery` for the class `LinkedBag` that removes all occurrences of a given entry from a bag.

```
/** Removes every occurrence of a given entry from this bag.
    @param anEntry  The entry to be removed. */
public void removeEvery(T anEntry)
{
   if (!isEmpty())
   {
      Node searcher = firstNode;
      while (searcher != null)
      {
         T nextEntry = searcher.data;
         if (nextEntry.equals(anEntry))
         {
            searcher.data = firstNode.data; // Replace located entry with entry in first node
            firstNode = firstNode.next;     // Remove first node
            numberOfEntries--;
         } // end if
         searcher = searcher.next;          // Continue searching
      } // end while
   } // end if
} // end removeEvery
```

6. Repeat Exercise 10 in Chapter 2 for the class `LinkedBag`.
Suppose that a bag contains `Comparable` objects such as strings. A `Comparable` object belongs to a class that implements the standard interface `Comparable<T>`, and so has the method `compareTo`. Implement the following methods for the class `LinkedBag`:

- The method `getMin` that returns the smallest object in a bag
- The method `getMax` that returns the largest object in a bag
- The method `removeMin` that removes and returns the smallest object in a bag
- The method `removeMax` that removes and returns the largest object in a bag

```
/** Gets the smallest value in this bag.
    @returns  A reference to the smallest object, or null if the bag is empty. */
public T getMin()
{
   T smallestValue = null;
   if (!isEmpty())
   {
      smallestValue = firstNode.data;
      Node nextNode = firstNode.next;
      while (nextNode != null)
      {
         T nextValue = nextNode.data;
         if (nextValue.compareTo(smallestValue) < 0)
            smallestValue = nextValue;
         nextNode = nextNode.next;
      } // end while
   } // end if

   return smallestValue;
} // end getMin
```

```
/** Gets the largest value in this bag.
    @returns  A reference to the largest object, or null if the bag is empty. */
public T getMax()
{
   T largestValue = null;
   if (!isEmpty())
   {
      largestValue = firstNode.data;
      Node nextNode = firstNode.next;
      while (nextNode != null)
      {
         T nextValue = nextNode.data;
         if (nextValue.compareTo(largestValue) > 0)
            largestValue = nextValue;
         nextNode = nextNode.next;
      } // end while
   } // end if

   return largestValue;
} // end getMax

/** Removes the smallest value in this bag.
    @returns  A reference to the removed object, or null if the bag was empty
              prior to this operation. */
public T removeMin()
{
   if (!isEmpty())
   {
      T smallestValue = getMin();
      remove(smallestValue);
      return smallestValue;
   }
   else
      return null;
} // end removeMin

/** Removes the largest value in this bag.
    @returns  A reference to the removed object, or null if the bag was empty
              prior to this operation. */
public T removeMax()
{
   if (!isEmpty())
   {
      T largestValue = getMax();
      remove(largestValue);
      return largestValue;
   }
   else
      return null;
} // end removeMax
```

7. Repeat Exercise 11 in Chapter 2 for the class `LinkedBag`.
Suppose that a bag contains `Comparable` objects, as described in the previous exercise. Define a method for the class `ArrayBag` that returns a new bag of items that are less than some given item. The header of the method could be as follows:

```
public BagInterface<T> getAllLessThan(Comparable<T> anObject)
```

Make sure that your method does not affect the state of the original bag.

See the note in the solution to Exercise 10 of Chapter 2 about student background.

```
/** Creates a new bag of objects that are in this bag and are  less than a given object.
    @param  anObject  A given object.
    @return  A new bag of objects that are in this bag and are less than anObject. */
public BagInterface<T> getAllLessThan(Comparable<T> anObject)
{
   BagInterface<T> result = new LinkedBag<>();
   Node nextNode = firstNode;
   while (nextNode != null)
   {
      if (anObject.compareTo(nextNode.data) > 0)
         result.add(nextNode.data);
      nextNode = nextNode.next;
   } // end while

   return result;
} // end getAllLessThan
```

8. Define an `equals` method for the class `LinkedBag`. Consult Exercise 11 in the previous chapter for details about this method.

```
public boolean equals(Object other)
{
   boolean result = false;

   if (other instanceof LinkedBag)
   {
      // The cast is safe here
      @SuppressWarnings("unchecked")
      LinkedBag<T> otherBag = (LinkedBag<T>)other;
      int otherBagLength = otherBag.getCurrentSize();
      if (numberOfEntries == otherBagLength) // Bags must contain the same number of objects
      {
         result = true;                      // Assume equal
         Node thisNextNode = this.firstNode;
         Node otherNextNode = otherBag.firstNode;
         int nodeCounter = 0;

         while ((nodeCounter < numberOfEntries) && result)
         {
            T thisBagEntry = thisNextNode.data;
            T otherBagEntry = otherNextNode.data;
            if (thisBagEntry.equals(otherBagEntry))
            {
               thisNextNode = thisNextNode.next;
               otherNextNode = otherNextNode.next;
               nodeCounter++;
            }
            else
               result = false;               // Bags have unequal entries
         } // end while
      } // end if
      // Else bags have unequal number of entries
   } // end if

   return result;
} // end equals
```

9. Define the method `union`, as described in Exercise 5 of Chapter 1, for the class `LinkedBag`.

```
public BagInterface<T> union(BagInterface<T> anotherBag)
{
   BagInterface<T> unionBag = new LinkedBag<>();
   LinkedBag<T> otherBag = (LinkedBag<T>)anotherBag;

   int index;

   // Add entries from this bag to the new bag
   Node nextNode = firstNode;
   for (index = 0; index < numberOfEntries; index++)
   {
      unionBag(nextNode.data);
      nextNode = nextNode.next;
   } // end for

   // Add entries from the second bag to the new bag
   nextNode = otherBag.firstNode;
   for (index = 0; index < otherBag.numberOfEntries; index++)
   {
      unionBag.add(nextNode.data);
      nextNode = nextNode.next;
   } // end for

   return unionBag;
} // end union
```

10. Define the method `intersection`, as described in Exercise 6 of Chapter 1, for the class `LinkedBag`.

```
public BagInterface<T> intersection(BagInterface<T> anotherBag)
{
   // The count of an item in the intersection is
   // the smaller of the count in each bag

   BagInterface<T> intersectionBag = new LinkedBag<>();
   LinkedBag<T> otherBag = (LinkedBag<T>)anotherBag;
   BagInterface<T> copyOfAnotherBag = new LinkedBag<>()

  // Copy the second bag
   Node nextNode = otherBag.firstNode;
   for (int index = 0; index < otherBag.numberOfEntries; index++)
   {
      copyOfAnotherBag.add(nextNode.data);
      nextNode = nextNode.next;
   } // end for

   // Add to the new bag each item in this bag that matches an item in the second bag;
   // once matched, remove it from the second bag
   for (int index = 0; index < numberOfEntries; index++)
   {
      if (copyOfAnotherBag.contains(nextNode.data))
      {
         intersectionBag.add(nextNode.data);
         copyOfAnotherBag.remove(nextNode.data);
         nextNode = nextNode.next;
      } // end if
   } // end for

   return intersectionBag;
} // end intersection
```

11. Define the method `difference`, as described in Exercise 7 of Chapter 1, for the class `LinkedBag`.

```
public BagInterface<T> difference(BagInterface<T> anotherBag)
{
   // The count of an item in the difference is
   // the difference of the counts in the two bags.

   BagInterface<T> differenceBag = new LinkedBag<>();
   LinkedBag<T> otherBag = (LinkedBag<T>)anotherBag;

   // Copy this bag
   Node nextNode = firstNode;
   for (int index = 0; index < numberOfEntries; index++)
   {
      differenceBag.add(nextNode.data);
      nextNode = nextNode.next;
   } // end for

   // Remove the ones that are in anotherBag
   nextNode = otherBag.firstNode;
   for (int index = 0; index < otherBag.numberOfEntries; index++)
   {
      if (differenceBag.contains(nextNode.data)
      {
         differenceBag.remove(nextNode.data);
      } // end if
      nextNode = nextNode.next;
   } // end for

   return differenceBag;
} // end difference
```

12. In a **doubly linked chain**, each node can reference the previous node as well as the next node. Figure 3-11 shows a doubly linked chain and its head reference. Define a class to represent a node in a doubly linked chain. Write the class as an inner class of a class that implements the ADT bag. You can omit set and get methods.

```
private class DoublyLinkedNode
{
   private T    data;                 // Entry in bag
   private DoublyLinkedNode next;     // Link to next node
   private DoublyLinkedNode previous; // Link to previous node

   private DoublyLinkedNode(T dataPortion)
   {
      this(dataPortion, null, null);
   } // end constructor

   private DoublyLinkedNode(T dataPortion, DoublyLinkedNode nextNode,
                                           DoublyLinkedNode previousNode)
   {
      data = dataPortion;
      next = nextNode;
      previous = previousNode;
   } // end constructor
} // end DoublyLinkedNode
```

13. Repeat Exercise 12, but instead write the class within a package that contains an implementation of the ADT bag. Set and get methods will be necessary.

```
package BagPackage;
class DoublyLinkedNode<T>
{
   private T                    data;     // Entry in bag
   private DoublyLinkedNode<T> next;     // Link to next node
   private DoublyLinkedNode<T> previous; // Link to previous node

   DoublyLinkedNode(T dataPortion)
   {
      this(dataPortion, null, null);
   } // end constructor

   DoublyLinkedNode(T dataPortion, DoublyLinkedNode<T> nextNode,
                                   DoublyLinkedNode<T> previousNode)
   {
      data = dataPortion;
      next = nextNode;
      previous = previousNode;
   } // end constructor

   T getData()
   {
      return data;
   } // end getData

   void setData(T newData)
   {
      data = newData;
   } // end setData

   DoublyLinkedNode<T> getNextNode()
   {
      return next;
   } // end getNextNode

   void setNextNode(DoublyLinkedNode<T> nextNode)
   {
      next = nextNode;
   } // end setNextNode

   DoublyLinkedNode<T> getPreviousNode()
   {
      return previous;
   } // end getPreviousNode

   void setPreviousNode(DoublyLinkedNode<T> previousNode)
   {
      previous = previousNode;
   } // end setPreviousNode
} // end DoublyLinkedNode
```

14. List the steps necessary to add a node to the beginning of the doubly linked chain shown in Figure 3-11.

```
newNode = a new DoublyLinkedNode containing the new entry and two null references
firstNode = newNode
if (firstNode != null)
{
   newNode.setNextNode(firstNode)
   firstNode.setPreviousNode(newNode)
}
firstNode = newNode
```

15. List the steps necessary to remove the first node from the beginning of the doubly linked chain shown in Figure 3-11.

```
firstNode = firstNode.getNextNode()
firstNode.setPreviousNode(null)
```

Chapter 4: The Efficiency of Algorithms

1. Using Big Oh notation, indicate the time requirement of each of the following tasks in the worst case. Describe any assumptions that you make.
   a. After arriving at a party, you shake hands with each person there.
   b. Each person in a room shakes hands with everyone else in the room.
   c. You climb a flight of stairs.
   d. You slide down the banister.
   e. After entering an elevator, you press a button to choose a floor.
   f. You ride the elevator from the ground floor up to the *n*th floor.
   g. You read a book twice.

   a. O(*n*), where *n* is the number of people at the party under the assumptions that there is a fixed maximum time between hand shakes, and there is a fixed maximum time for a handshake.
   b. O(n^2), where *n* is the number of people at the party under the assumptions that there is a fixed maximum time between hand shakes, and there is a fixed maximum time for a handshake.
   c. O(n),where n is the number of steps under the assumptions that you never take a backward step, and there is a fixed maximum time between steps.
   d. O(*h*), where *h* is the height of the banister under the assumptions that you never slow down and you reach a limiting speed.
   e. O(1), under the assumption that you reach a decision and press the button within a fixed amount of time.
   f. O(*n*), where *n* is the number of the floor under the assumptions that the elevator only stops at floors, there is a fixed maximum time for each stop, and there is a fixed maximum time that an elevator requires to travel between two adjacent floors.
   g. O(*n*), where *n* is the number of pages in the book and under the assumptions that you never read a word more than twice, you read at least one word in a session, there is a fixed maximum time between sessions, and there is a fixed maximum number of words on a page.

2. Describe a way to climb from the bottom of a flight of stairs to the top in time that is no better than O(n^2).

   Repeat until you reach the top:
   Go up n / 2 steps, then down n / 2 - 1 steps

3. Using Big Oh notation, indicate the time requirement of each of the following tasks in the worst case.
   a. Display all the integers in an array of integers.
   b. Display all the integers in a chain of linked nodes.
   c. Display the n^{th} integer in an array of integers.
   d. Compute the sum of the first *n* even integers in an array of integers.

   a. O(*n*), where *n* is the number of integers in the array.
   b. O(*n*), where *n* is the number of integers in the chain.
   c. O(1). You can access the n^{th} integer directly without searching the array.
   d. O(*n*).

4. By using the definition of Big Oh, show that
   a. $6n^2 + 3$ is $O(n^2)$
   b. $n^2 + 17n + 1$ is $O(n^2)$
   c. $5n^3 + 100\ n^2 - n - 10$ is $O(n^3)$
   d. $3n^2 + 2n$ is $O(2^n)$

   a. We have to find a positive real number c and a positive integer N such that $f(n) \leq c\ g(n)$ for all $n \geq N$, where $f(n)$ is $6n^2 + 3$ and $g(n)$ is n^2. Choose $c = 9$ and $N = 1$. We must show that $6n^2 + 3 \leq 9n^2$ if $n \geq 1$ or equivalently show that $-3n^2 + 3 \leq 0$ if $n \geq 1$. But since $n \geq 1$,
   $n^2 \geq 1$
   $3n^2 \geq 3$
   $3n^2 - 3 \geq 0$
   $-3n^2 + 3 \leq 0$

5. Algorithm X requires $n^2 + 9n + 5$ operations, and Algorithm Y requires $5n^2$ operations. What can you conclude about the time requirements for these algorithms when n is small and when n is large? Which is the faster algorithm in these two cases?

n	$n^2 + 9n + 5$	$5n^2$
1	15	5
2	27	20
3	41	45
100,000	10,000,900,005	50,000,000,000
1,000,000	1,000,009,000,005	5,000,000,000,000
100,000,000	10,000,000,900,000,000	50,000,000,000,000,000
1,000,000,000	1,000,000,009,000,000,000	5,000,000,000,000,000,000

   From the sample data, you can see that Algorithm Y is faster than Algorithm X when n is 1 or 2. After that, X is faster. As n becomes very large, Y requires about five times the number of operations as does X. However, the order of these numbers for a given n are the same. Thus, we say that the algorithms are each $O(n)$.

6. Show that $O(\log_a n) = O(\log_b n)$ for $a, b > 1$. Hint: $\log_a n = \log_b n / \log_b a$.

   $O(\log_a n) = O(\log_b n / \log_b a)$
   Since $1 / \log_b a$ is a constant—call it k—we have
   $O(\log_a n) = O(k \log_b n) = O(\log_b n)$ by the first identity in Segment 4.16.

8. Segment 4.9 and the chapter summary showed the relationships among typical growth-rate functions. Indicate where the following growth-rate functions belong in this ordering:
   a. $n^2 \log n$
   b. $\sqrt{n}$
   c. $n^2 / \log n$
   d. $3n$

   a. $n^2 \log n = \Omega(n^2)$ (lower bound)
   $n^2 \log n = O(n^3)$ (upper bound)
   So $n^2 \log n$ lies between n^2 and n.

b. $\sqrt{n} = \Omega(\log^2 n)$ (lower bound)

$\sqrt{n} = O(n)$ (upper bound)

So $\sqrt{n}$ lies between $\log^2 n$ and n.

c. $n^2/\log n = \Omega(n \log n)$ (lower bound)

$n^2/\log n = O(n^2)$ (upper bound)

So $n^2/\log n$ lies between $n \log n$ and n.

d. $2^n < 3^n$ (lower bound)

$3^n < n!$ for $n > 6$ (upper bound)

So 3^n lies between 2^n and $n!$

10. What is the Big Oh of the following computation?

```
int sum = 0;
for (int counter = n; counter > 0; counter = counter - 2)
   sum = sum + counter;
```

$O(n)$.

11. What is the Big Oh of the following computation?

```
int sum = 0;
for (int counter = 1; counter < n; counter = 2 * counter)
   sum = sum + counter;
```

$O(n)$.

12. Suppose that your implementation of a particular algorithm appears in Java as follows:

```
for (int pass = 1; pass <= n; pass++)
{
   for (int index = 0; index < n; index++)
   {
      for (int count = 1; count < 10; count++)
      {
         . . .
      } // end for
   } // end for
} // end for
```

The algorithm involves an array of n items. The previous code shows the only repetition in the algorithm, but it does not show the computations that occur within the loops. These computations, however, are independent of n. What is the order of the algorithm?

$O(n^2)$.

13. Repeat the previous exercise, but replace 10 with n in the inner loop.

$O(n^3)$.

14. What is the Big Oh of `method1`? Is there a best case and a worst case?

```
public static void method1(int[] array, int n)
{
   for (int index = 0; index < n - 1; index++)
   {
      int mark = privateMethod1(array, index, n - 1);
      int temp = array[index];
      array[index] = array[mark];
      array[mark] = temp;
   } // end for
} // end method1

public static int privateMethod1(int[] array, int first, int last)
{
   int min = array[first];
   int indexOfMin = first;
   for (int index = first + 1; index <= last; index++)
   {
      if (array[index] < min)
      {
         min = array[index];
         indexOfMin = index;
      } // end if
   } // end for
   return indexOfMin;
} // end privateMethod1
```

$O(n^2)$ regardless of the order of the data within the array.

15. What is the Big Oh of `method2`? Is there a best case and a worst case?

```
public static void method2(int[] array, int n)
{
   for (int index = 1; index <= n - 1; index++)
      privateMethod2(array[index], array, 0, index - 1);
} // end method2

public static void privateMethod2(int entry, int[] array, int begin, int end)
{
   int index = end;
   While ((index >= begin) && (entry < array[index]))
   {
      array[index + 1] = array[index];
      index--;
   } // end while
   array[index + 1] = entry;
} // end privateMethod2
```

$O(n^2)$. $O(n^2)$ in the worst case. $O(n)$ in the best case.

16. Consider two programs, A and B. Program A requires 1000 x n^2 operations and Program B requires 2^n operations. For which values of n will Program A execute faster than Program B?

First, solve to determine when the two programs take the same amount of time:

$1000\ n^2 = 2^n$

$\log (1000\ n^2) = \log 2^n$

$\log 1000 + \log n = n$

or

$n = \log 1000 + \log n$

$\approx 10 + 2 \log n$

Using an iterative process, try $n = 10$ on the right side:

$n \approx 10 + 2 \log 10$
$\approx 10 + 6.6$
≈ 16.6
$n \approx 10 + 2 \log 16.6$
$\approx 10 + 8$
≈ 18
$n \approx 10 + 2 \log 18$
$\approx 10 + 8$
≈ 18

So now,

If $n = 18$, A requires 324,000 operations, and B requires 262,144 operations.

If $n = 19$, A requires 361,000 operations, and B requires 524,288.

Program A is faster if $n > 18$. B is faster if $n < 18$.

17. Consider four programs—A, B, C, and D—that have the following performances:

A $O(\log n)$

B $O(n)$

C $O(n^2)$

D $O(2^n)$

If each program requires 10 seconds to solve a problem of size 1000, estimate the time required by each program when the size of its problem increases to 2000.

Program A requires time $T(n) \approx k \times \log n$

Use the given values for n and $T(n)$ to solve for k:

$T(1000) \approx k \times \log 1000 \approx 10$

$10 \times k \approx 10 \times k \approx 1$

So $T(n) \approx \log n$

For $n = 2000$, $T(2000) \approx \log 2000 \approx 11$

Program B requires time $T(n) \approx k \times n$

Use the given values for n and $T(n)$ to solve for k:

$T(1000) \approx k \times 1000 \approx 10 \times k \approx 1/100$

So $T(n) \approx n / 100$

For $n = 2000$, $T(2000) \approx 2000 / 100 \approx 20$

Program C requires time $T(n) \approx k \times n^2$

Use the given values for n and $T(n)$ to solve for k:

$T(1000) \approx k \times 1000^2 \approx 10 \times k \approx 10^{-5}$

So $T(n) \approx 10^{-5} \times n^2$

For $n = 2000$, $T(2000) \approx 10^{-5} \times 2000^2 \approx 10^{-5} \times 4 \times 10^6 \approx 40$

Program D requires time $T(n) \approx k\, 2^n$

Use the given values for n and $T(n)$ to solve for k:

$T(1000) \approx k \times 2^{1000} \approx 10$

$k \approx 10 / 2^{1000}$

So $T(n) \approx (10 / 2^{1000})\, 2^n$

For $n = 2000$, $T(2000) \approx (10 / 2^{1000})\, 2^{2000} \approx 10 \times 2^{1000} \approx 10^{301}$