

## Problem Set 4

Due: 6pm Friday, October 4

### Exam 1 Notice:

In class on Friday, October 4, the first take-home exam will be handed out. It will be due at 6pm on Friday, October 11. **This is a hard deadline. No extensions will be given after this time.** The exam will cover the material in lecture through Lecture 9 (Tue. Oct. 1) and the material in problem sets through PS4: object diagrams, iteration, recursion, arrays, vectors, lists, enumerations, text processing, using and implementing abstract data types.

Because you should focus on the exam, it is strongly recommended that you submit PS4 on time (6pm Friday October 4).

**Overview:** In this problem set, you will get experience with implementing a vector data structure and recursive list programs.

**Download:** To begin this assignment, you should download a copy of the directory `~cs230/download/ps4`.

In your local copy of this directory, you will:

- Modify `IntVector.java` in Problem 1.
- Modify `BlackjackPoints.java` in Problem 2.
- Modify `ChangeMaker.java` in Problem 3.

### Submission:

- For Problem 1, your hardcopy submission should be your final version of `IntVector.java`.
- For Problem 2, your hardcopy submission should be your final version of `BlackjackPoints.java`.
- For Problem 3, your hardcopy submission should be your final version of `ChangeMaker.java`.

Your softcopy submission for this problem should be the entire `ps4` directory.

Remember to include a signed cover sheet (found at the end of this problem set description) at the beginning of your hardcopy submission.

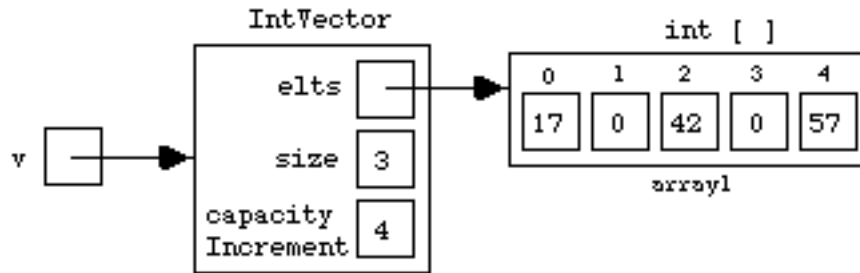
### Problem 1 [45]: An Array Implementation of Integer Vectors

#### The `IntVector` class

We have seen that the Java `Vector` class is a versatile data structure handy for a wide range of uses. The fact that the elements of a vector have type `Object` means that vectors can store any kind of object. However, as we have seen, this feature is a source of inconvenience and inefficiency when the elements of a vector all have the same type. In this case, every extraction of a vector element requires a type downcast, which not only makes the code uglier, but implies a run-time check on the type of the object. The situation is even worse in the case of storing primitive datatypes in a vector. In this case it is additionally necessary to package each primitive datum in an instance of a “wrapper class” (e.g. `Integer`) before storing it in the vector, and unpackage it upon extraction.

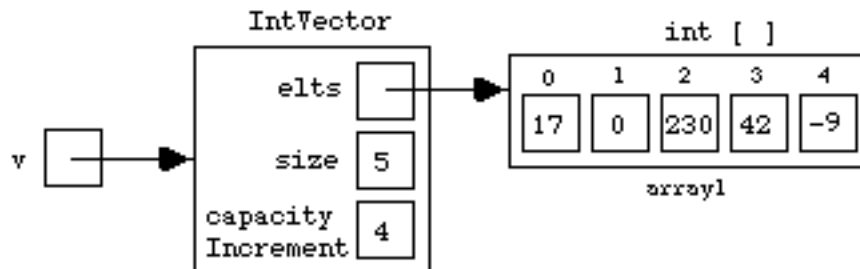
In this problem, we will explore the implementation of an `IntVector` class that is a template for a specialized kind of vector that can store only integers. The `IntVector` class has the same kinds of methods as the `Vector` class, except that wherever a `Vector` method returns an `Object`, an `IntVector` returns an `int`. When using the `IntVector` class to model a collection of integers, it is not necessary to package the integers into instances of the `Integer` class, nor is it necessary to use type downcasts when extracting elements from the vector.

We will implement an instance of `IntVector` using an array of integers. Here is a diagram of one instance of `IntVector` that contains, in order, the three integers 17, 0, and 42:



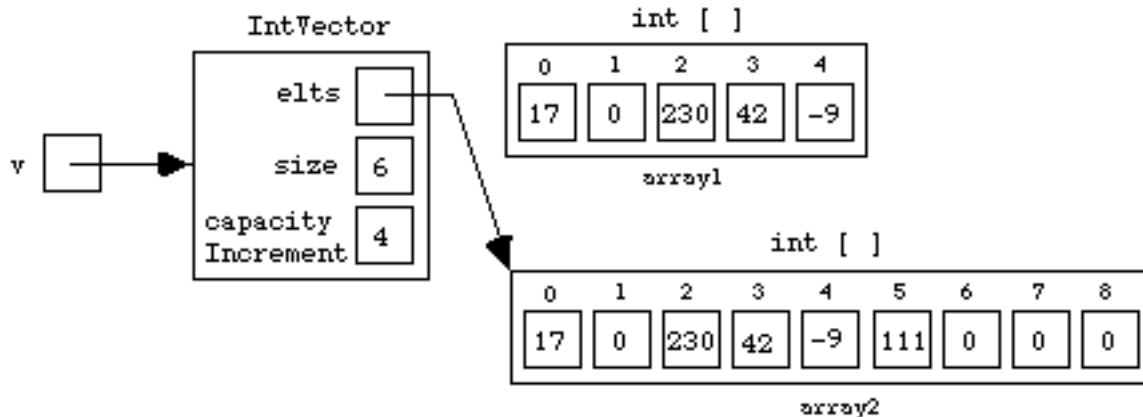
The `elts` instance variable points to an integer array that stores the elements of the vector. The number of slots in the array (known as the *capacity* of the vector) can be any number greater than or equal to the number of elements in the vector. The `size` instance variable indicates the number of slots (from left to right, starting at slot 0) that contain the elements of the vector. The remaining slots are extra space for the vector to “grow into”. These extra slots may contain any integers whatsoever; their values do not matter because they are ignored by the `IntVector` methods. We shall refer to the values of these extra slots as “garbage”. In the above example, the size of 3 indicates that only slots 0 through 2 hold vector elements and slots 3 and 4 hold garbage.

We will assume that as long as the number of elements in the vector remains less than or equal to the capacity, the array held by the `elts` variable does not change. In this case, vector operations that change, add, or remove elements must rearrange the elements within the slots of the existing array. For instance, performing `v.add(-9)` and then `v.add(2,230)` on the above vector representation should change the state of the objects as follows, where the fact that the array label has not changed indicates that it is the same array as in the previous diagram.



When an element is added or inserted into a vector whose size is equal to its capacity, it is necessary to increase its capacity. This is accomplished by creating a larger integer array, copying the vector elements from the old array to the new array, and making `elts` point to the new array. The number of slots by which the capacity should be increased is indicated by the `capacityIncrement` instance variable. For example, since the `capacityIncrement` is 4 in the example, performing

`v.add(111)` installs a new integer array of size  $5 + 4 = 9$  in `elts` before it performs the insertion. The old array will be reclaimed by the Java garbage collector. By default, the garbage slots in the new array will initially contain 0, although after other vector operations the garbage slots might later contain any integer.



We will assume the convention used in Java’s `Vector` class that if the `capacityIncrement` is 0, then the capacity of the vector should be doubled when the array needs to grow. For instance, if the `capacityIncrement` had been 0 above, then the new array would have had  $2*5 = 10$  slots.

The initial capacity and capacity increment of an `IntVector` instance are specified by arguments to the `IntVector` constructors:

```
// Constructor Methods
public IntVector(int initialCapacity, int capacityIncrement);
public IntVector(int capacityIncrement);
public IntVector();
```

An invocation of the first constructor creates an integer vector with no elements (i.e., `size` is 0) whose initial capacity is `initialCapacity` and whose capacity increment is `capacityIncrement`. In the other constructor methods; the parameters not supplied are assumed to be 0.

The `IntVector` class supports the following subset of the “new style” instance methods supported by Java’s `Vector` class. If you have questions about the behavior of a particular method, consult the on-line Java 2 SDK 1.4 API, accessible from the CS230 Documentation page. The `Vector` class is in the `java.util` package. (Of course, the following specifications differ from those in the `Vector` class by replacing every element of `Object` with `int`.)

```
// Instance Methods
public boolean add (int elt); // Note: always returns true.
public void add (int index, int elt);
public int capacity();
public void ensureCapacity (int minCapacity);
public int get (int index);
public int indexOf (int elt);
public int remove (int index);
public void removeAllElements();
public boolean removeElement (int elt);
public int set (int index, int elt);
public int size ();
```

### *Implementing the IntVector Class*

Your task in this problem is to implement the `IntVector` class using the “extensible array” representation discussed above. You should flesh out the skeletons for the above constructor methods and instance methods in the `IntVector` class in the file `IntVector.java`. Invoking `java IntVector` tests your implementation on a suite of test cases. Each test case will be run on vectors whose capacity increments are 0, 1, and 8.

### *Notes*

- The case where both the capacity and `capacityIncrement` are 0 is problematic, since doubling 0 yields 0 rather than a larger capacity! Your code should handle this as a special case by setting the new capacity to be 1.
- Your code will need to indicate an error when an index is not within the valid indices of the vector in the `get`, `set`, (two-argument) `add`, and `remove` methods. To do this you should use the following statement:

```
throw new ArrayIndexOutOfBoundsException(message);
```

where `message` is a string indicating the specific error message.

## **Problem 2 [30]: Blackjack Points**

### *Counting points in Blackjack*

In the game of Blackjack, the goal is to acquire a hand of cards worth as close to 21 points as possible (inclusive) without going over 21 points. The points for each card are determined as follows:

- A card with value two through ten is worth its value.
- Face cards (jacks, queens, kings) are worth 10 points.
- An ace is worth either 1 point or 11 points – the player decides which. Of course, the player will attempt to assign aces values that may the hand as close as possible to 21

If an ace were to have only one value (say 1), then the point value of a hand would be easy to calculate, as show in the following `handPoints` method:

```

public static int handPoints (Hand h) {
// Assume that every ace counts as a 1;
    int points = 0;
    for (int i = 0; i < h.size(); i++) {
        points = points + cardPoints(h.get(i), false);
    }
    return points;
}

private static int cardPoints (Card c, boolean acesHigh) {
// Return the number of points for the card c.
// If acesHigh is true, counts aces as 11; otherwise counts them as 1.
    if (c.value().equals(Value.ACE)) {
        if (acesHigh) {
            return 11;
        } else {
            return 1;
        }
    } else if (c.value().toInt() >= 10) {
        return 10;
    } else {
        return c.value().toInt();
    }
}
}

```

The fact that aces can have two possible values makes the calculation trickier. There are many ways to address the ace-has-two-possible-values problem. Here we shall explore one approach that is particularly elegant and flexible (in the sense that it's easy to extend to games where many cards can each have many possible values).

In this approach, we use an auxiliary method `pointList` that calculates a *list* of the possible points for a given hand. The list is such that the elements are sorted from low to high and there are no duplicates. Given such a method, the `handPoints` method can be rewritten to handle two-valued aces as follows:

```

public static int handPoints (Hand h) {
    return closestTo21(pointList(h));
}

```

Here, `closestTo21` is a method that finds the value in a non-empty sorted integer list that is closest to 21, choosing any value less than or equal to 21 in preference to one greater than 21.

#### *Your Task*

In this problem, your task is to flesh out several list methods in the `BlackjackPoints` class in `ps4/BlackjackPoints.java` that implement the above strategy:

a. [5]

```

public static int closestTo21 (IntList L);

```

Assume that `L` is a non-empty list of integers sorted from low to high. If `L` contains elements less than or equal to 21, returns the largest integer that is less than or equal to 21. If `L` contains only elements greater than 21, returns the smallest such element.

b. [5]

```
public static IntList mapAdd (int n, IntList L);
```

Returns a new integer list that has the same length as L and in which each element is n greater than the corresponding element of L.

c. [5]

```
public static IntList merge (IntList L1, IntList L2);
```

Assume that L1 and L2 are integer lists without duplicates sorted from low to high. Returns a new integer list that has all the elements of both L1 and L2, without any duplicates, sorted from low to high.

d. [15]

```
public static IntList pointList (Hand h);
```

Returns a list of all possible Blackjack point values for hand h, sorted from low to high, without duplicates. Calling this method should have no observable effect on h. That is, h immediately after a call to pointList should seem to be exactly the same hand as h immediately before the call to pointList.

*Notes:*

- No testing methods have been provided for you in this problem. Part of the problem is for you to develop appropriate tests and use them to show that your methods work as expected. Your `BlackjackPoints.main()` method should take string arguments that allow you to test all of the above methods (including `handPoints`) individually. You should include transcripts of your testing output.
- Use `IL.fromString(intListString)` to convert a string representation of an integer list (*intListString*) to a bona fide `IntList`.
- Use `Hand.fromString(handString)` to convert a string representation of the hand (*handString*) to a bona fide `Hand`. `Hand` representations are comma-delimited sequences of card representations wrapped in `Hand[` and `]`. For example: `Hand[7S,4C,KH,AD,TS]`
- Your `pointList` method (or one of the auxiliary methods, if you employ them) should calculate the point list recursively. That is, the point list for a hand should be determined by combining the possible points for the first (alternatively, last) card and the point list for the hand excluding the first (respectively, last) card. As part of combining, you should find that `mapAdd` and `merge` are *very* handy.
- You may introduce any auxiliary methods that you find helpful for implementing the above methods.
- Since it is tedious to use fully qualified list method names like `IntList.head`, `IntList.isEmpty`, etc., `BlackjackPoints` has been configured so that each `IntList` method can be invoked with the prefix `IL`, as in `IL.head`, `IL.isEmpty`, etc.



Amount of money	Denominations	Ways to Make Change
5	[5,1]	[[5], [1,1,1,1,1]] 1 nickel or 5 pennies
16	[10,5,1]	[[10, 5, 1], [10, 1, 1, 1, 1, 1, 1], [5, 5, 5], [5, 5, 1, 1, 1, 1, 1, 1], [5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
16	[10,5]	[]
21	[10,5,1]	[[10, 10, 1], [10, 5, 5, 1], [10, 5, 1, 1, 1, 1, 1, 1], [10, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [5, 5, 5, 5, 1], [5, 5, 5, 1, 1, 1, 1, 1, 1], [5, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1]] ]

Figure 1: Some examples of change making.

- Since `makeChange` involves working with `IntList` and `IntListList` lists, we can no longer just use list methods and operators without the class name because Java won't know which method we want (i.e. do we want the one that applies to the `IntList` or the `IntListList` class?). Instead, we must specify the name of the class with the method (e.g. `IntList.prepend` or `IntListList.prepend`). However, specifying the full name of the class makes the code harder to read so we have provided you with the following “shortcut” way to refer to `IntList` and `IntListList` methods and operators for doing this problem.

<b>IntList Methods</b>	<b>IntListList Methods</b>
<code>IL.empty</code>	<code>ILL.empty</code>
<code>IL.prepend</code>	<code>ILL.prepend</code>
<code>IL.head</code>	<code>ILL.head</code>
<code>IL.tail</code>	<code>ILL.tail</code>
<code>IL.isEmpty</code>	<code>ILL.isEmpty</code>
<code>IL.length</code>	<code>ILL.length</code>
<code>IL.append</code>	<code>ILL.append</code>
<code>IL.postpend</code>	<code>ILL.postpend</code>

- You may find it useful to use the following `mapPrepend` method from lecture. It has been included in the `ChangeMaker` class for your convenience:

```
public static IntListList mapPrepend (int n, IntListList L) {
    if (ILL.isEmpty(L)) {
        return L;
    } else {
        return ILL.prepend (IL.prepend(n, ILL.head(L)),
                            mapPrepend(n, ILL.tail(L)));
    }
}
```

- Hint for the general case(s) of your recursion: when making change for a given amount, if the first coin in the denomination list isn't too big, you can either use it in the resulting change lists or not. Consider a particular example, such as making change for 21 cents using the denominations [10, 5, 1]. What change lists do you get if you use the 10? What change lists do you get if you don't use the 10. Do you see a pattern? Use wishful thinking!
- Think about the following carefully for the base case(s) of the recursion: What does it mean to make change from an empty list of coins? What does it mean to make change for zero money? (Depending on how you approach the problem, you might also need to consider making change for a negative amount of money.)
- It is possible to write a definition for `makeChange` in very few lines. If your solution is longer than 15 lines or so, you are probably on the wrong track!

```

-----
There are 2 ways to make change for 15 cents using [10, 5]:
[[10, 5],
 [5, 5, 5]
]
-----
There are 0 ways to make change for 16 cents using [10, 5]:
[ ]
-----
There are 2 ways to make change for 5 cents using [10, 5, 1]:
[[5],
 [1, 1, 1, 1, 1]
]
-----
There are 6 ways to make change for 16 cents using [10, 5, 1]:
[[10, 5, 1],
 [10, 1, 1, 1, 1, 1, 1],
 [5, 5, 5, 1],
 [5, 5, 1, 1, 1, 1, 1, 1],
 [5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
]
-----
There are 9 ways to make change for 21 cents using [25, 10, 5, 1]:
[[10, 10, 1],
 [10, 5, 5, 1],
 [10, 5, 1, 1, 1, 1, 1, 1],
 [10, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [5, 5, 5, 5, 1],
 [5, 5, 5, 1, 1, 1, 1, 1, 1],
 [5, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
]
-----
There are 13 ways to make change for 26 cents using [25, 10, 5, 1]:
[[25, 1],
 [10, 10, 5, 1],
 [10, 10, 1, 1, 1, 1, 1, 1],
 [10, 5, 5, 5, 1],
 [10, 5, 5, 1, 1, 1, 1, 1, 1],
 [10, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [10, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [5, 5, 5, 5, 5, 1],
 [5, 5, 5, 5, 1, 1, 1, 1, 1, 1],
 [5, 5, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [5, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
]

```

Figure 2: The output of java ChangeMaker test (Part 1).



*Problem Set Header Page  
Please make this the first page of your hardcopy submission.*

## **CS230 Problem Set 4**

### **Due 6pm Friday, October 4**

Name:

Date & Time Submitted:

Collaborators (*anyone you worked with on the problem set*):

*By signing below, I attest that I have followed the collaboration policy as specified in the Course Information handout.*

Signature:

*In the **Time** column, please estimate the time you spend on the parts of this problem set. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the **Score** column when grading your problem set.*

<b>Part</b>	<b>Time</b>	<b>Score</b>
General Reading		
Problem 1 [45]		
Problem 2 [30]		
Problem 3 [25]		
<b>Total</b>		