Problem Set 1

Due before lecture on Wednesday, September 22

Getting Started

Before starting this assignment, make sure that you have completed Problem Set 0, which can be found on the assignments page of the course website. In particular, make sure that you have entered the command

```
~cscie119/setmeup
```

from the command line on `nice.harvard.edu`.

Then, to get the files that you will need for this problem set, log into `nice.harvard.edu` and enter the following command:

```
gethw 1
```

This will create a directory named `ps1` in your Harvard home directory containing the files that you need for this assignment. To change into this directory, the command is:

```
cd ~/ps1
```

There are two approaches that you can take when working on the assignment from your own computer; follow the link entitled *Instructions for working on your own machine* on the assignments page of the course website to obtain a detailed description of each approach. Regardless of the approach that you choose, the final versions of your files for Problem Set 1 should ultimately be put in your `ps1` directory on `nice.harvard.edu` and submitted from there, following the directions below.

Submitting Your Work

Once you are ready to submit, make sure that all of your files are in your `ps1` directory on `nice.harvard.edu`.

To check that your code compiles, enter the following command from within the `ps1` directory: `make`

To submit your assignment, enter the following command: `make submit`

To check which files were submitted, enter the following command: `make check`

Important note: the submit commands only work if you are logged onto *`nice`*`.harvard.edu`.

I. Written Problems (35 points total)

If you are able to attend lecture, you are welcome to submit your solutions to the written problems in written form, handing them in prior to lecture. If you are unable to attend lecture, or if you prefer, you may submit your solutions electronically as either a text file or a PDF file. To do so, name the file written.txt *or* written.pdf*, and copy it into the same directory as your program files before you submit them electronically using the instructions at the end of this document.*

1. **Memory management and arrays** (5 points)
   Consider the following lines of Java code:

```
int[] a = {2, 4, 6, 8, 10, 12};
int[] b = new int[6];
int[] c = new int[6];
b = a;
for (int i = 0; i < a.length; i++)
    c[i] = a[i];
a[2] = c[5];
c[2]++;
System.out.println(a[2] + " " + b[2] + " " + c[2]);
```

   a) (3 pts.) Draw a single memory diagram that shows the final result of these lines. Include both the stack and the heap in your diagram. You may assume that these lines are part of the `main` method.

   b) (2 pts.) Indicate what will be printed by the final line of code shown above.

2. **Array practice** (10 points; 5 points each part)
   a) Write a method with the header

```
public static boolean isSorted(int[] arr)
```

   that takes a reference to an array of integers and returns `true` if the array is sorted (i.e., if the elements are in increasing order), and `false` otherwise. If the parameter is `null`, the method should throw an `IllegalArgumentException`. If passed an empty array (one with length 0) or an array with one element, the method should return `true` – although you shouldn't need to include special code for these cases.

   b) Write a method with the header

```
public static int mostFrequentValue(int[] arr)
```

   that takes a reference to a sorted array of integers and returns the value that occurs most frequently in the array. For example, consider this array:

```
int[] arr = {1, 2, 3, 3, 8, 8, 8, 8, 11, 11, 11, 14, 19, 19};
```

The method call `mostFrequentValue(arr)` should return 8, because 8 is the most frequently occurring value in the array, appearing four times.

If two or more values tie for the most occurrences, return the one that comes first. For example, if we added an extra 11 to the array shown above – giving that value four occurrences as well – the method should still return 8, because 8 comes before 11 in the array. If all of the values occur exactly once, the first element should be returned.

The array is guaranteed to be in sorted order, and you may assume that it has at least one element. If there is only one element, it is the value that should be returned. ***For full credit, you should <u>not</u> use another array as part of your solution, and you should <u>not</u> perform more than one scan through the array from left to right.***

These methods do *not* need to use recursion, and they do not need to be implemented as part of a class. Simply include the methods with your answers to the other written problems.

3. **Recursion and the runtime stack** (10 points)

Consider the following recursive method:

```
public static int mystery(int a, int b) {
    if (a < 0)
        return 1;
    else
        return 2 + mystery(a - b, b);
}
```

a) (3 pts.) Trace the execution of `mystery(20, 6)`. Use indentation to indicate which statements are performed by a given invocation of the method, following the approach used in the lecture notes to trace the `sum` method.

b) (2 pts.) What is the value returned by `mystery(20, 6)`?

c) (2 pts.) During the execution of `mystery(20, 6)`, method frames are added and then removed from the stack. How many method frames are on the stack when the base case is reached? You should assume that the initial call to `mystery(20, 6)` is made from within the `main` method, and you should include the stack frame for `main` in your count.

d) (3 pts.) Give an example of values of `a` and `b` that would produce infinite recursion, and explain why it would occur.

4. **Recursive methods and inductive proofs** (10 points total)
Consider the following recursive method that takes an even number n and computes the sum of all even numbers from 0 to n:

```
public static int sumEvens(int n) {
    if (n < 0 || n % 2 != 0)
        throw new IllegalArgumentException("n must be >= 0 and even");
    else if (n == 0)
        return 0;
    else
        return n + sumEvens(n - 2);
}
```

We want to use proof by induction to show that it takes **n/2 + 1** calls of this method to compute the sum of the even numbers from 0 to n for any even number n >= 0.

a) (3 pts.) *Basis of the proof:* Show that the formula n/2 + 1 works when n = 0.

b) (2 pts.) *Inductive hypothesis:* State the assumption made in this part of the proof.

b) (5 pts.) *Inductive step:* Carry out this final step of the proof. (Note: because n can only be even, your induction step will need to show that, given the assumption that the formula holds for n = k, it also holds for n = k + ***2***.)

II. Programming Problems (65-75 points total)

1. **Adding methods to the ArrayBag class** (25 points)
In the file `ArrayBag.java`, add the methods described below to the `ArrayBag` class, and then add code to the `main()` method to test these methods. **In addition, you should update the `Bag` interface that we have given you in `Bag.java` to include these new methods.** These methods should be publicly accessible. **You should *not* add any new instance variables to the class.**

`int roomLeft()` – returns the number of additional items that the `ArrayBag` has room to store. For example, if the maximum size of the bag is 10 and there are currently 7 items in the bag, this method should return 3, since the bag has room for 3 more items. *Hint:* This method should only need one or two lines of code.

`boolean isEmpty()` – returns `true` if the `ArrayBag` is empty, and `false` otherwise.

`void increaseCapacity(int increment)` – increases the maximum capacity of the bag by the specified amount. For example, if `b` has a maximum capacity of 10, then `b.increaseCapacity(5)` should give `b` a maximum capacity of 15. As part of your implementation, you will need to create a new array with room to

support the new maximum capacity, copy any existing items into that array, and replace the original array with the new one by storing its reference in the calling object. If `increment` is 0, the method should just return. If `increment` is negative, the method should throw an `IllegalArgumentException`. See our second `ArrayBag` constructor for an example of throwing an exception.

`boolean addItems(Bag other)` – attempts to add to the calling `ArrayBag` all of the items found in the parameter `other`. If there is currently room for all of the items to be added, the items are added and the method returns `true`. If there isn't enough room for all of the items to be added, *none* of them are added and the method returns `false`. Note that the parameter is of type `Bag`. As a result, your method should use method calls to access the internals of that bag. See our implementation of the `containsAll()` method for an example of this.

`Bag intersectionWith(Bag other)` – creates and returns an `ArrayBag` containing *one occurrence* of any item that is found in both the calling object and the parameter `other`. For full credit, the resulting bag should not include any duplicates. For example, if `b1` represents the bag {2, 2, 3, 5, 7, 7, 7, 8} and `b2` represents the bag {2, 3, 4, 5, 5, 6, 7}, then `b1.intersectionWith(b2)` should return an `ArrayBag` representing the bag {2, 3, 5, 7}. If there are no items that occur in both bags, the method should return an empty `ArrayBag`. Give the new `ArrayBag` a maximum size that is the greater of the two bag's maximum sizes. Note that the parameter is of type `Bag`. As a result, your method should use method calls to access the internals of that bag. See our implementation of the `containsAll()` method for an example of this. The return type is also `Bag`, but polymorphism allows you to just return the `ArrayBag` that you create, because `ArrayBag` implements `Bag`.

2. **Recursion and strings** (10-15 points total; 5 points each part)
In a file named `StringRecursion.java`, implement the methods described below, and then create a `main()` method to test these methods. Your methods *must* be recursive; ***no credit will be given for methods that employ iteration***. In addition, ***global variables (variables declared outside of the method) are not allowed.*** You may find it helpful to employ the `substring`, `charAt`, and `length` methods of the `String` class as part of your solutions.

   a) `printReverse()` should take a `String` as a parameter and use recursion to print it in reverse order. The method should have the following header:

```
public static void printReverse(String str)
```

   For example, `printReverse("Harvard")` should print

```
dravraH
```

If the parameter is `null` or the empty string (`""`), the method should not print anything. This method should *not* call the `reverse` method that you write for part b.

b) `reverse()` should take a `String` as a parameter and use recursion to create and return a `String` that is the reverse of the original one. The method should have the following header:

```
public static String reverse(String str)
```

This problem is similar to part a, but the method should *return* the reversed string rather than printing it. If the parameter is `null`, the method should return `null`. If the parameter is the empty string, the method should return the empty string.

c) *(required for grad-credit students; "partial" extra credit for others)* `trim()` should take a `String` as a parameter and use recursion to create and return a `String` in which any leading and/or trailing spaces in the original string are removed. The method should have the following header:

```
public static String trim(String str)
```

For example, `trim("  hello world ")` should return `"hello world"` and `trim("recursion      ")` should return `"recursion"`. If the parameter is `null`, the method should return `null`. If the parameter is the empty string, the method should return the empty string. The `String` class comes with a built-in `trim()` method that does the same thing as the method that we're asking you to write; you may *not* use that method in your solution!

3. **Lagrange's theorem** (30 points)

A perfect square is an integer than can be written as the square of another integer. For example, here are the first 10 positive perfect squares:

$1 = 1^2$	$36 = 6^2$
$4 = 2^2$	$49 = 7^2$
$9 = 3^2$	$64 = 8^2$
$16 = 4^2$	$81 = 9^2$
$25 = 5^2$	$100 = 10^2$

Lagrange proved that any positive integer can be expressed as the sum of at most 4 positive perfect squares. For example:

$1 = 1$
$2 = 1 + 1$
$3 = 1 + 1 + 1$
$4 = 4$

5 = 4 + 1
6 = 4 + 1 + 1
7 = 4 + 1 + 1 + 1
8 = 4 + 4
9 = 9
10 = 9 + 1
11 = 9 + 1 + 1
12 = 9 + 1 + 1 + 1
13 = 9 + 4
14 = 9 + 4 + 1
15 = 9 + 4 + 1 + 1

In the file `Lagrange.java`, write a Java program that takes a positive integer and breaks it into a sum of at most 4 perfect squares. At the heart of your program should be a method called `findSum()` that uses recursive backtracking to search for an appropriate sum. You will need to determine this method's return type and the number and meaning of its parameters. You may also include other helper methods as you see fit.

To see why this problem lends itself to recursive backtracking, consider the following illustration of one possible procedure for breaking up the number 140:

We're trying to break 140 into the sum of at most 4 perfect squares.

The largest perfect square less than or equal to 140 is 121 = 11^2, so we try using it as the first term: terms[0] = 121.

*140 – 121 = 19, and we've already used up one of our four terms, so we need to break 19 into the sum of **at most 3** perfect squares. We apply our procedure recursively to solve this smaller problem.*

The largest perfect square less than or equal to 19 is 16, so we try using it as the second term: terms[1] = 16.

*19 – 16 = 3, and we've already used up two of our four terms, so we need to break 3 into the sum of **at most 2** perfect squares. We apply our procedure recursively to solve this smaller problem.*

The largest perfect square less than or equal to 3 is 1, so we try using it as the third term: terms[2] = 1.

*3 – 1 = 2, and we've already used up three of our four terms, so we need to break 2 into the sum of **at most 1** perfect square. We apply our procedure recursively to solve this smaller problem.*

Since 2 isn't a perfect square itself, this can't work, so we backtrack.

We're back to trying to break 3 into at most 2 perfect squares. We already tried including 1 in the sum. There are no positive perfect squares less than 1, so we backtrack again.

We're back to the trying to break 19 into at most 3 perfect squares. We already tried including 16 in the sum. The next smallest perfect square is 9, so we try using it as the second term: terms[1] = 9.

19 – 9 = 10, so we need to break 10 into the sum of at most 2 perfect squares.

The largest perfect square less than or equal to 10 is 9, so we try using it as the third term: terms[2] = 9.

10 – 9 = 1, so we need to break 1 into the sum of at most 1 perfect square.

1 is a perfect square itself, so we use it as the fourth term: terms[3] = 1.

Success! 140 = 121 + 9 + 9 + 1

Your program should repeatedly ask the user for a positive integer, read it from the keyboard, and display the result of breaking it into at most 4 perfect squares. If the user enters -1, the program should quit. For example:

```
Enter a positive integer (-1 to quit):  13
13 = 9 + 4

Enter a positive integer (-1 to quit):  140
140 = 121 + 9 + 9 + 1

Enter a positive integer (-1 to quit):  -1
Goodbye!
```

Your program should print only one solution for each integer, even though there may be more than one solution in some cases.

Try to make your program as efficient as possible. In particular, it is much more efficient to start with the largest possible perfect square and work down (as illustrated in the example above) than it is to start with the smallest possible perfect square and work up. To make your job easier, we have given you a helper method named `largestSquare()` that takes a positive integer n and returns the largest perfect square less than or equal to n.

4. **Make it smarter** (5 pts; *required for grad-credit; partial extra credit for others*)
Make your `findSums()` method smarter about when to give up. For instance, in the process of breaking up 140, we could have backtracked as soon as we realized that we needed to break 3 into at most two perfect squares.