# CS 2510 Exam 3 – Spring 2013

Name: _____

Student Id (last 4 digits): _____

• Write down the answers in the space provided.
• You may use all syntax that you know from *Java* we have covered so far. You do not need (nor should use) any loops statements. If you need a method and you don't know whether it is provided, define it.
• For tests you only need to provide the expression that computes the actual value, connecting it with an arrow to the expected value. For example `s.method(arg) -> true` is sufficient.
• Remember that the phrase "design a class" or "design a method" means more than just providing a definition. It means to design them according to the **design recipe**. You are *not* required to provide a method template unless the problem specifically asks for one. However, be prepared to struggle if you choose to skip the template step.
• Make sure you answer as completely as you can at least one of the parts *C*, *D*, or *E*. • We will not answer *any* questions during the exam.

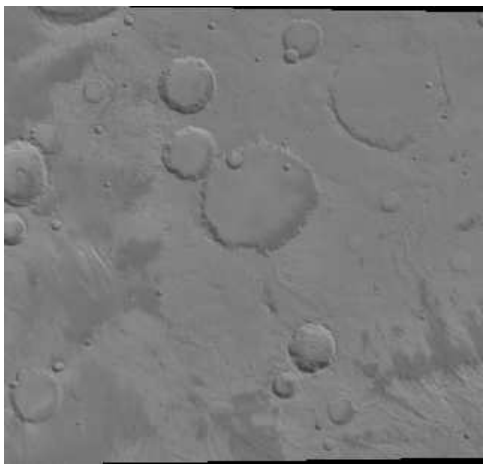
*Good luck.*

| **Problem** | Points | / |
|---|---|---|
| A | | / 5 |
| B | | / 6 |
| C | | / 6 |
| D | | / 4 |
| E | | / 8 |
| F | | / 6 |
| **Total** | | /35 |

34 POINTS

**Problem 1**

The methods you design here would help you in enhancing pale grayscale images, such as this image taken a long time ago by the Mars Viking Explorer:



By changing the shade of every pixel using the technique known as *linear scaling* the image features become more pronounced:

The image is saved as a long list of shade values for each pixel in the image. The pixel values range from 0 to 255, and the size of the whole image is about 300 by 300 pixels.

The shades in the pale gray image do not span the whole available range - they are all in the approximate range from 70 to 170. So, the enhancement process changes the shades close to 70 to much darker shades, closer to 0, while making the lighter shades close to 170 even lighter, close to 255. The scaling is based on computing the lowest and the highest shade in the image.

An even better method is called *histogram equalization*. To use this method one needs to first compute the histogram of all shades - the frequency of how often a particular shade appears in the image. We will not design all methods for this technique, but we will compute the histogram.

For all problems here, we assume that the entire image is represented as one `ArrayList<Integer>`.

The methods you design will be a part of some `Algorithms` class, or can be defined directly in the `Examples` class.

A. *(5 points)*

Design the method *buildHisto* that consumes an `ArrayList<Integer>` where we know that the values of the elements of the `ArrayList` will be in the range $[0, 256)$. For your examples, you can (and should) use a smaller range of values.

The method produces a `HashMap<Integer, Integer>` that represents the histogram (frequency count) of the values in the given `ArrayList`.

So, if the given `ArrayList` had five one's, three sixes, and two nines, the hash map would contain pairs$(1, 5)(6, 3)$, $(9, 2)$, i.e. the key is the value of an element in the given `ArrayList` and the corresponding value is the frequency with which this element appears in the given `ArrayList`.

Include the above example in your tests, but make at least one additional example.

This page is intentionally blank - for your work

B. *(6 points)*

Design the method `hash2list` that consumes a `HashMap<Integer, Integer>` (such as you have built in the previous part). It assumes that the keys in the given `HashMap` are in the range $[0, 255)$ and it produces an `ArrayList<Integer>` where the value of the item at index `i` is the frequency of that value as represented by the given `HashMap`. Of course, if the value does not appear in the `HashMap`, its frequency is zero.

This page is intentionally blank - for your work

C. *(6 points)* We assume that the values in the given `ArrayList<Integer>` are non-zero only for the elements in the middle of the index range. We know that the first few, and the last few elements are zeros.

   Design the methods `findFirst` and `findLast` that determines the first (and the last) index in the given `ArrayList<Integer>` whose value is non-zero.

D. *(4 points)* Design the method that finds the maximum value in the given `ArrayList<Integer>` whose values are known to be non-negative numbers.

E. *(8 points)* The interface `T2S` represents a method that consumes an object of the type `T` and produces an object of the type `S`:

```
// to represent a function from the type T to the type S
interface T2S<T, S>{
  // produce an integer value from the given one
  public S apply(T i);
}
```

We want to define a linear scaling function that distributes the values in the range `[low, high]` to values in the range `[0, size]`.

Design the class that implements the interface `T2S` with a method that consumes an `Integer` and produces an `Integer` as follows. First, it assumes that the values the method consumes are in the range `[0, size]`. It computed the resulting values using the following rules:

- If the given value is $\leq$ `low` it returns `0`.
- If the given value is $\geq$ `high` it returns `size`.
- Otherwise it computes the value for the given element $i$ by applying uses the formula:
  $result = size * (i - low) / (high - low)$

This page is intentionally blank - for your work

F. *(6 points)* Design the method `map` that consumes an `ArrayList<T>` and an instance of the class `T2S<T, S>` and produces an `ArrayList<S>` by invoking the `apply` method on every element of the given `ArrayList`.

Use the instance of the `LinearScaling` defined in the previos part in one of your examples.