Project 3: QuadTrees for Image Processing

DUE: Sunday, March 21, 11:59pm – Extra Credit for Early Submission!

Introduction

The quadtree data structure is a special type of tree that can recursively divide a 2D space into four quadrants. Each node in a quadtree has either zero or four children. Quadtrees have various applications in domains like image processing, spatial indexing, collision detection, storing of sparse data, and others. This project is inspired by the use of quadtrees in image processing but you don't need to have a background on this topic. All you need to know is described in this document.

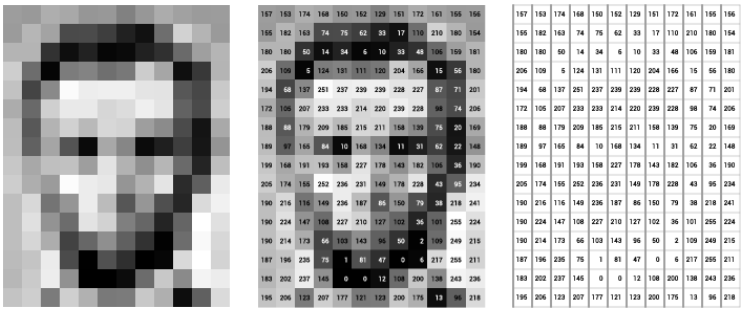
Where should I start?

Its recommended to follow these steps in this specific order:

1. Read the Background section to fully understand what kind of data stuctures you're called to implement. Do **not** start coding before you do that!
2. Check to see if you have a tool to view PGM files. Open the provided **`image1.pgm`** with your favorite image viewing application. Most applications support the PGM format. If not, download the popular GIMP which is free and opensource. Although this is not a required step to complete the project, it will help you a lot to be able to visualize the images you're working on as well as the ones you produce.
3. Read the Rules and Assumptions section. It's very important to follow the rules of this assignment otherwise you might get zero points on certain tasks.
4. Start working on your code by implementing the following methods in this specific order:
    1. `Utilities.loadFile` because you can't do anything unless you load an image file first
    2. `QuadTreeImag` constructor because it creates the quadtree which is the basis for everything else
    3. `QuadTreeImage.sizeRatio` because it's a low-hanging fruit and will quickly get you into the tree-traversal mood
    4. Decide if you want to add more methods to class `ImageBlob`. Reminder: you may not add any fields to this class
    5. `QuadTreeImage.compareTo` because it's not too hard and will get you one step closer to fully grasping the concept of tree traversal
    6. The rest is up to you to decide. Just make sure you leave the `setColor` and `intersectionMask` for the end!

Background

A digital image with dimensions *H*x*W* is represented with a **matrix of pixels** that has *H* rows and *W* columns. In grayscale images, each pixel is an integer that takes values from 0-255 (i.e. 1 byte). A value of zero corresponds to the black color and a value of 255 corresponds to the white color. Everything in between is a tone of gray; the darker the gray the closer to 0, the lighter the gray the closer to 255. Figure 1a depicts a zoom-in on a 16x12 grayscale image. In Figure 1b you can see the values of each pixel superimposed on the image. And in Figure 1c you can see the matrix representation of this grayscale image. In this project we will not work with color images but everything we discuss here could equally well apply to color images too.



a *b* *c*

Figure 1: Grayscale image

The reason we want to use quadtrees to store images is that certain image processing operations (e.g. intersection, union) can be done very efficiently. If you consider the fact that a typical image nowadays has about 10-20 millions pixels, it is easy to infer why efficiency is very important for image processing applications.

Let's now see how an image can be stored efficiently with a quadtree. For illustration purposes only, we will demonstrate the process for a special type of grayscale image, called bitmap. Pixels in a bitmap can only take two of the 256 possible values, either black or white. In Figure 2 we have an 8x8 bitmap on the left and the respective quadtree on the right. The leafs of the tree represent regions of pixels that have the same color. These nodes don't need any children because they can efficiently represent the whole region by using three value only: the two coordinates of the region's origin, and the color. Inner nodes, on the other hand, represent regions of pixels that do not have the same color. Even if it's a single pixel in the whole region that has a different color from the rest, the region needs to be further subdivided. Keep in mind that all inner nodes *must* have exactly 4 children representing the 4 sub-quadrants. Obviously, leafs can be as small as 1x1, while inner nodes can't be smaller than 2x2. In Figure 2, leafs are depicted with the color of their region (either black or white), while inner nodes with gray. The root of the tree represents the whole image and can be either a leaf or an inner node. The four children/subquadrants of an inner node must follow a predetermined spatial order; in this figure it's a clockwise order (see 1→2→3→4) but, in general, it can be any fixed order you want.

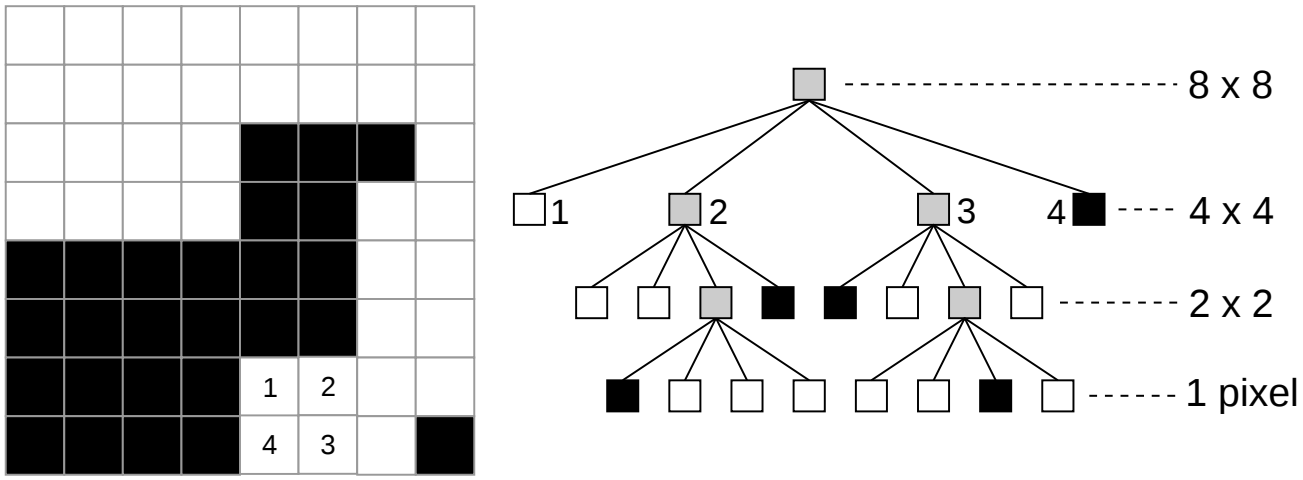


Figure 2: Quadtree for a bitmap (source: wikipedia)

This partitioning of the image works best for images where similar colors take up a large portion of space like in the top left or the bottom left quadrants of the bitmap in Figure 2. Take a look at an actual image in Figure 3 for an example of how efficient this data structure can become.

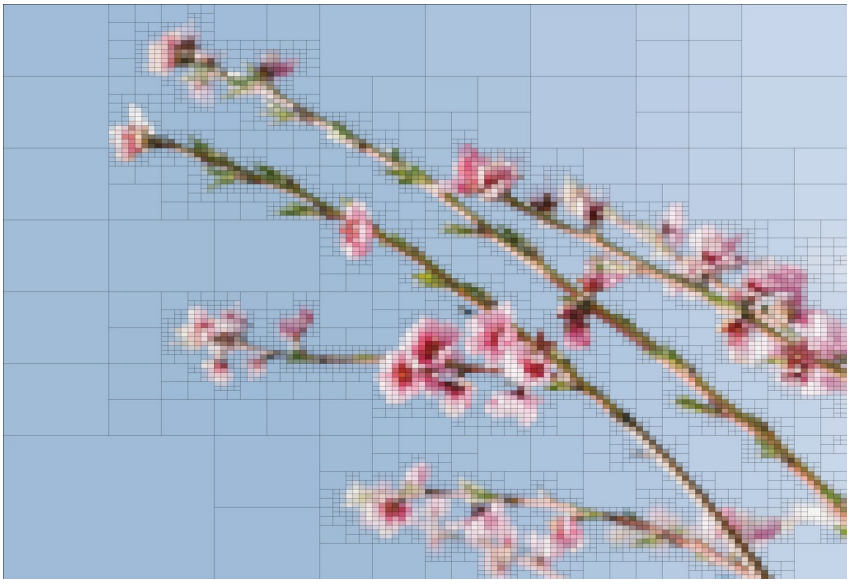


Figure 3: Example of quadtree partition (source: medium.com/@tannerwyork)

Images come in many different formats, but to keep things simple for this project, we will use the PGM format (portable graymap format). PGM images can be stored in either a binary or a text format. We will use the text format to make the reading/editing of the images and the debugging of your code simpler. All major photo viewing/editing applications should support this format. So, it will be easy to visually inspect the input you use and the output you generate. In the snippet below, you can see an example of an actual image stored in a PGM text format. The first row is the code **`P2`** that indicates the format of the file. The second row contains the width and the height of the image (**`6`** and **`4`** respectively). The third row defines the maximum grayscale value which is **`255`** in this case (minimum is **`0`** by default). And what follows is simply a series of *`width*height`* pixel values. Open your favorite

plain text editor, copy+paste the below snippet, and save the file as **`image.pgm`** (still in plain text). Then, open it with your favorite photo viewing/editing application to verify that it looks like Figure 4. You will have to zoom-in a lot because this image is just `6x4` pixels. You can find more information about the format at https://en.wikipedia.org/wiki/Netpbm (but really, you don't need to know more).

```
P2

6 4

255

10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240
```

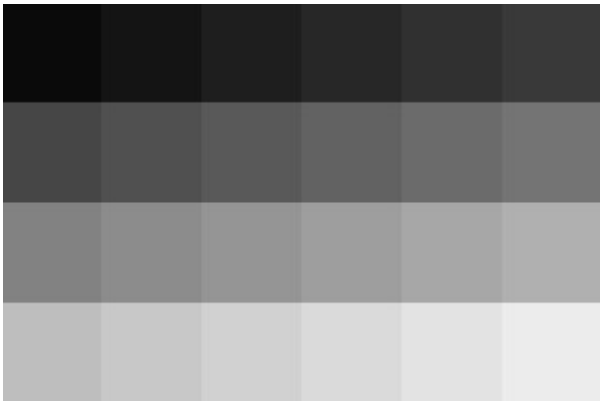


Figure 4: Image rendering of the above PGM snippet

Rules and Assumptions

You must:

- Have code that compiles with the command: **`javac *.java`** in your user directory **without errors or warnings**.

You may:

- Add additional methods and variables, however these **must be private**.
- Add additional nested, local, or anonymous classes, but any nested classes must be private.
- Use the provided **`Project3.java`** to get started and test your code but you should **not** include it in your submitted files. Keep in mind that **this is not a tester**, it is only provided to give you an idea of how your code should work.

You may **NOT**:

- Add additional *public* methods, variables, or classes. You may have public methods in private nested classes.
- Use any built in Java Collections Framework classes in your program (e.g. no **`LinkedList`**, **`ArrayList`**, etc.).
- Add any additional import statements (or use the *fully qualified name* to get around adding import statements).
- Store anywhere in your program the **`Pixel[][] array`** that was passed to the **`QuadTreeImage`** constructor. This array is used only for constructing the quadtree. Violation of this rule will result in a 0 on the **entire** assignment.
- Make your program part of a package.
- Change any code in any “DO NOT EDIT” section.
- Add `@SuppressWarnings` to avoid fixing warnings. If there is any method that is exempted from this rule, it will be clearly stated in the description.
- Add any additional libraries/packages which require downloading from the internet.

Assumptions

- We will assume that images have a **square size** and the width of an image is a **power of 2** (i.e. 1, 2, 4, 8, 16, 32, etc). This will simplify the creation of the quadtree as you won’t have to deal with rectangular regions. It will also make memory usage more efficient since you don’t need to store the coordinates of the image regions into the tree nodes.

Tasks

You must implement the following classes and methods

`final class Utilities`

It contains two utility methods that we need in order to read/write images from/to files.

`static Short[][] loadFile(String pgmFile)`

It reads a PGB image from a file and stores the data in an array of type `Short`. Modifying the return type of this method to `Integer` or `Float` should not affect the rest of your program since everything else in this project is using generics.

`static void exportImage(QuadTreeImage image, String filename)`

It writes a QuadTreeImage into a PGM file. The signature is not complete, you need to add the generic type. There is no time-complexity restriction for this method. This means that you *are* allowed to call the `getColor` method if you want. This is the only place in your entire code that you can can use the `getColor`.

`class QuadTreeImage<Pixel>`

This class is the most important component of this project. It uses a quadtree structure to represent a grayscale image and provides various methods for processing the tree (a.k.a. image). The generic type `Pixel` can be any subtype of `java.lang.Number`. The class must implement the `Comparable` and the `Iterable` interfaces. The above signature is obviously **not** complete.

`QuadTreeImage(Pixel[][] array)`

The constructor builds the quadtree from an array that holds the original data read from the PGM file. You **may not store the array** anywhere in your program after the creation of the tree.

`Pixel getColor(int w, int h)`

Returns the value of the pixel at location *w*, *h* where *w* and *h* are the column and row indexes in the original image respectively. If the coordinates *w* and *h* are invalid, it throws an `IndexOutOfBoundsException`. You may **not** make any use of this method anywhere outside the method `exportImage`. It will result in 0 points for any method that violates this restriction. The same is true if you try to bypass the restriction by building your own helper method that has the same or a similar functionality.

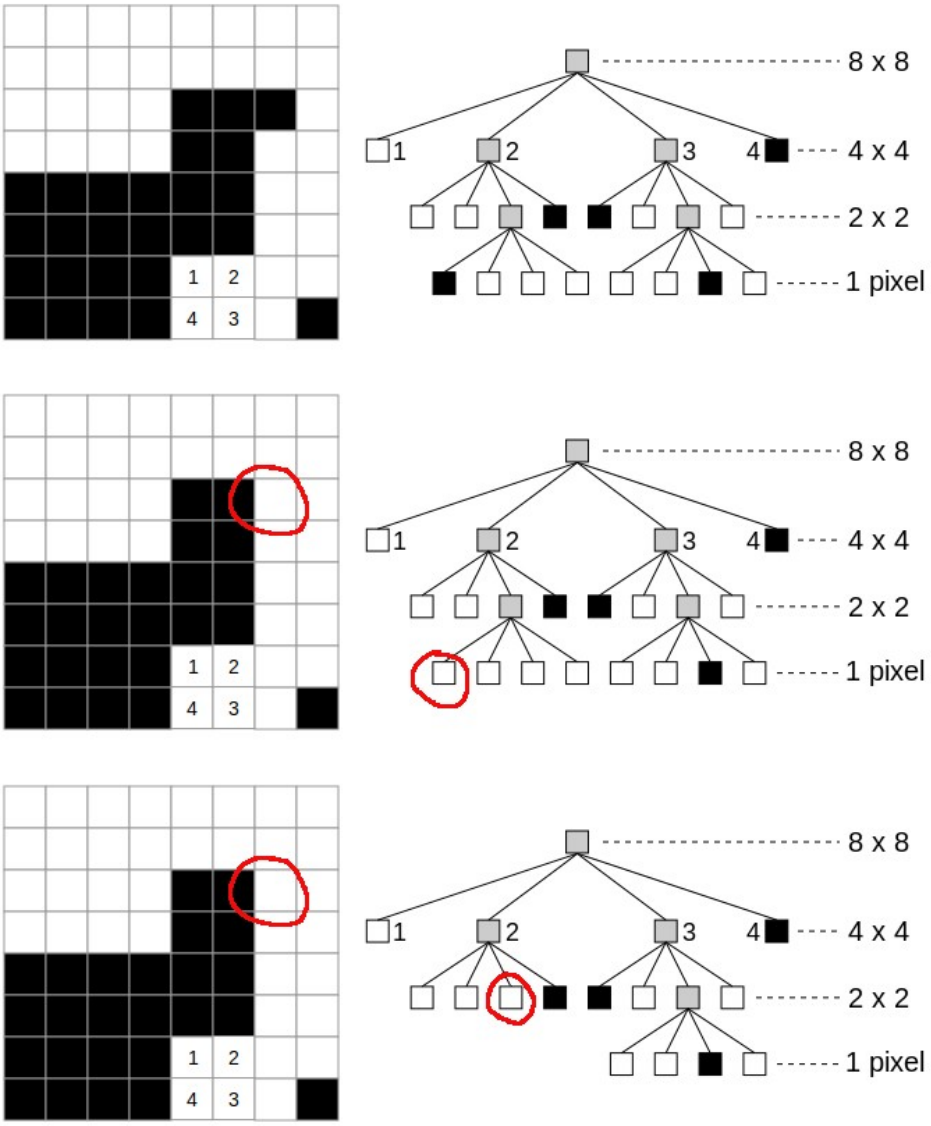


Figure 5: Effect of calling setColor(2,6,255)

`void setColor(int w, int h, Pixel value)`

Sets a new value for the pixel at location *w*, *h* where *w* and *h* are the column and row indexes in the original image. If the new value is different from the previous one, the tree will require a restructuring. See Figure 5 for an example. By setting the pixel 2,6 to value 255, all leafs become white and the subtree should now be removed. Keep in mind that such an action can trigger a restructuring that goes many levels up the tree. Think, for instance, what would happen if the 2x2 black node to the right of the one we've just changed was a white node originally.

A major restriction for this method is that you're not allowed to recreate the image array, you must work directly on the tree. It will result in 0 points if you violate this restriction. You may not use the `getColor` or any kind of helper method with similar functionality to bypass this restriction and recreate the image array.

Time-complexity restriction: it must run in O(*m*) where *m* is the number of nodes in the tree; not in O(*n*) where *n* is the number of pixels in the image.

`float sizeRatio(int bytesPerPixel, int bytesPerPointer)`

Returns the ratio of the size (in bytes) of the image stored as a quadtree to the size (in bytes) of the same image stored as an array. In Figure 2, for example, the original image requires 64 pixels and the quadtree requires 21 nodes. Assume that each node contains the minimum information which one pixel value plus the four pointers to its children.

`String toString()`

Overrides the default behavior. It returns a representation of the quadtree that lists the nodes of the quadtree in a **breadth-first-traversal** order (and **clockwise** within each level). **Only leafs** should be included in the string. For each leaf, you report: the top row coordinate, followed by the left column coordinate, followed by the size of the image region it represents, followed by the pixel value. The following is an example of the string returned for Figure 2 (assuming it's a proper PGB where black is represented with a 0 and white is represented with a 255):

```
{0 0 4 255},{4 0 4 0},{0 4 2 255},{0 6 2 255},{2 4 2 0},{4 4 2 0},{4 6 2 255},{6 4 2 255},{2 6 1 0},{2 7 1 255},{3 7 1 255},{3 6 1 255},{6 6 1 255},{6 7 1 255},{7 7 1 0},{7 6 1 255}
```

`int compareTo(QuadTreeImage other)`

Class `QuadTreeImage` implements the `Comparable` interface and, therefore, it must provide this method. If the two trees are not the same, it returns the difference in the brightness of the two images. We measure the brightness as the sum of all the Pixel values in an image. A brighter image is considered larger. Please note that the sum should take into account all pixels in the image not just the ones in the leafs of the tree.

Time-complexity restriction: it must run in O(*m*) where *m* is the number of nodes in the tree; not in O(*n*) where *n* is the number of pixels in the image. Practically, this means that you can't use `getColor` or any other approach to recreate the full image array.

`QuadTreeImage intersectionMask(QuadTreeImage other)`

Returns a new image that represents the intersection of the current image with another one as a binary mask. This means that the returned image will only have values of 0 and 255. All pixels that match in the two images will generate a white (255) pixel in the output mask, while the non-matching ones will generate a black (0) pixel. Instead of doing this computation pixel by pixel, we can compute this mask efficiently by leveraging the quadtree's ability to represent multiple pixels with a single node. You are not allowed to use `getColor` or any other similar functionality.

Time-complexity restriction: it must calculate the mask in O(*m*) where *m* is the number of nodes in the tree; not in O(*n*) where *n* is the number of pixels in the image.

`Iterator<ImageBlob<Pixel>> iterator()`

It creates a `QuadTreeImageIterator` that is iterating over the quadtree nodes (i.e. `ImageBlob`) by a **breadth-first-traversal** order (and **clockwise** within each level). **Every node** should be included in the iteration, regardless if it's an inner or a leaf. When `next()` is called on a leaf, the return value is the `Pixel value` of the leaf. If `next()` is called on an inner node,

the return value is `null`. The following is the output of the for-each loop when run on the `QuadTreeImage` generated from the image in Figure 2 (assuming it's a proper PGB where black is represented with a 0 and white is represented with a 255). The 21 values correspond to the 21 nodes of the tree.

```
null 255 null null 0 255 255 null 0 0 255 null 255 0 255 255 255 255 255 0 255
```

class ImageBlob<Pixel>

This class represents a single node in the quadtree structure.

Pixel value

The value of the pixel if the node is a leaf or `null` if it's an inner node.

ImageBlob NW, NE, SE, SW

The four pointers to the children nodes (in clockwise order).

class Queue<T>

An **array implementation of a circular queue**. It's required for certain tree traversals. This is **the only section** of your code that you're allowed to use the annotation `@SuppressWarnings`

boolean isEmpty()

Returns `true` if the queue is empty, `false` otherwise

void enqueue(T value)

Adds an item to the back of the queue.

T dequeue()

Removes the front item from the queue.

T peek()

Inspects the front item in the queue without removing it.

class QuadTreeImageIterator

It creates an iterator of the quadtree that traverses the nodes in a breadth-first-search approach. Also, this is the iterator that an enhanced for-loop on `QuadTreeImage` will invoke automatically.

boolean hasNext()

Returns `true` if there are more nodes to be traversed, `false` otherwise

ImageBlob<Pixel> next()

Returns the next node in the quadtree according to a breadth-first-traversal.

Extra work… but not extra credit!

If you've finished with the project and you're not fed up with it, you can try extending your code to make it work with color images. The basic idea is the same but the pixels will now consist of RGB triplets instead of scalars. Check this link https://en.wikipedia.org/wiki/Netpbm for the PPM format that is very similar to PGM.

Setup

- Download the `p3.zip` and unzip it. This will create a folder `section-UserName-p3`
- Rename the folder replacing `section` with the `DL1, DL2, DL3`, etc. based on the lecture section you are in.
- Rename the folder replacing `UserName` with your netID (that's the first part of your GMU email address).
- After renaming, your folder should be named something like: `DL1-sdimitr-p3`
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files. You should also remove the **`Project3.java`** and the **`.pgm`** files we provided for testing purposes.
- You should submit the remaining **`.java`** files and your **`readme.txt`**
- Zip your user folder (not just the files) and name the zip `section-UserName-p3.zip` (no other type of archive) following the same rules for `section` and `UserName` as described above. The submitted file should look something like this:

```
DL1-sdimitr-p2.zip → DL1-sdimitr-p3 →    JavaFile1.java
                                         JavaFile2.java
                                         JavaFile3.java
                                         ...
```

- Submit to Blackboard

  DOWNLOAD AND VERIFY WHAT YOU HAVE UPLOADED IS THE RIGHT THING
  Submitting the wrong files will result in a 0 on the assignment!

Grading Rubric

Disclaimer: The professors reserve the right to adjust this rubric due to unforeseen circumstances.

How will my assignment be graded?

- Grading will be divided into two portions:
  - Automatic Testing: To assess the correctness of programs.
  - Manual Inspection: For features your programs should exhibit that are not easily tested with code (such as Big-O). You **cannot** get points (even style/manual-inspection points) for code that doesn't compile or for submitting just the files given to you.
- Extra credit for early submissions:
  - 1% extra credit rewarded for every 24 hours your submission made before the due time
  - Up to 5% extra credit will be rewarded
  - Your latest submission before the due time will be used for grading and extra credit checking. You **cannot** choose which one counts.
- No credit (a 0 for the entire assignment) will be given for any of the following:
  - Non submitted assignments
  - Assignments late by more than 48 hours
  - Non compiling assignments
  - Code that violates and restrictions or “you may not” mandates (such as using array where forbidden or library imports)
  - "Hard coded" solutions
  - Code that would win an obfuscated code competition with the rest of CS310 students.
  - Non-independent work (this will also be referred to the honor court, with a sanction of “course failure”).

Earned Points for Project Completion to Specification (Manual and Automatic Testing)

For this assignment a portion of the automated testing will be based on JUnit tests and/or a manual run of your program. The JUnit tests used for grading will **not** be provided for you (you need to test your own programs!), but the tests will be based on what has been specified in the project description and the comments in the code templates. A breakdown of the point allocations is given below:

10 pts	Utilities
60 pts	QuadTreeImage
5 pts	ImageBlob
15 pts	Queue
10 pts	QuadTreeImageIterator
up to +5 pts	Extra credit for early submission
up to -5 pts	Submission folder format
-5 pts	Not passing code style check
-5 pts	Not passing JavaDoc style check
-5 pts	Compiler Warnings OR using @SuppressWarnings in non-permitted places

Note: Points for Big-O requirements are included in the above numbers.