# Assignment 2: Quadtree Compression

***Due: April 8, 2021, 10:00 PM***

Please read the syllabus regarding late policies. This assignment will be completed **individually**.

## Introduction

A quadtree is a tree data structure in which each internal node has exactly four children.

In this assignment, we will be using the quadtree data structure to compress bitmap image files, and then decompress them afterwards.

In this assignment, you will:

- learn about the quadtree data structure
- learn to implement recursive operations on trees
- use inheritance and good OOP design to build an interface for your compression algorithm
- gain some insight into reading binary file data (although we do most of the work for you)

So then, you might be wondering how a quadtree works to compress a file. Given an image file (which is essentially a 2D matrix of pixels), we first split it into four equal quadrants. For each of those quadrants, we further split them into four quadrants until we reach a quadrant that only consists of one pixel. At this point, we can store the data of the pixel in a leaf node in the tree. When we want to retrieve pixel data from the tree, we can very easily traverse the quadtree structure to find out what the colour each pixel is. All we have to do is specify the location of the pixel, and depending on which quadrant the pixel is located in, recursively look into the children of current tree node until we reach a leaf, which we can retrieve the pixel data from.

However, there is one caveat to the approach we just described: it doesn't actually do any compression! If we store every single pixel in a tree structure, this would actually make it take up even more space, because of the extra attributes we would need to store at each node. Fret not, however, as there is a neat trick to fix this, and that is to introduce a **loss level** during our quadtree compression process.

The idea is very simple: when we are splitting up our pixel data into quadrants, we calculate the **standard deviation** and **mean** of the pixel values inside each of the four quadrants that we have just created. If the standard deviation does not exceed our loss level (an integer between 0 and 255, inclusive, and is specified by the user), then that means that the pixel values inside the quadrant is "close enough" to one another. When this happens, we get lazy and say, let's store this entire quadrant as one colour: the mean (average) value of all the colours that were originally in the quadrant. This is where the compression happens. By storing the entire quadrant as a single colour value, we essentially save all the space that we would have originally needed to store the colour values of each pixel in the quadrant. All we have to do now is to remember the height and width of the quadrant so that we can restore the image when needed.

This method of compression also does something quite neat, it preserves the parts of the image which have finer details, because quadrants with finer details results in a higher standard deviation as neighboring pixels are likely to have different colours (there is a lot of statistics and science behind this in fields like image recognition, etc). However, this also means that quadtree compression is a lossy compression algorithm. That is, once an image is compressed, it is impossible to retrieve the original image data from the compressed image file. The good part about this is that the compressed file can be quite small depending on the loss level, which you will see when you implement the assignment.

Here is the recommended order of tasks to do in order to complete this assignment, all the work that you will submit will be in `a2tree.py` (this is the only file you'll need to submit).

Note: The tasks are marked by TODOs in the starter code. In general, you're allowed to add new helper methods if needed; however, you should NOT modify the provided methods that don't have a TODO in it.

## Task 1: Understand the Starter Code

**Download the starter code HERE.**

Below is a quick description of the files in the starter code:

1. **a2tree.py**: This file contains the classes that define the quadtree data structure. This is the file that you'll be working with most of the time for this assignment. You **WILL** submit this file.
2. **a2files.py**: This file contains the classes for the input/output file formats used in this assignment, namely, BMP and QDT. You will **NOT** submit this file, so you need to make sure your `a2tree.py` works correctly with the **original** `a2files.py` in the starter code.
3. **a2main.py**: This is the main file for running the program. It defines the compressor and decompressor classes. You will need to add some code in the main block of this file for the user interface. However, you will **NOT** submit this file, so you need to make sure your `a2tree.py` works correctly with the class defined in the **original** `a2main.py` in the starter code.
4. **a2test_student.py**: This file contains a template for the test suite. You **WILL** be asked to submit this file with some required test cases. See Task 6 for the detail.
5. **dog.bmp**, **toronto.bmp**, and **uoft.bmp**: three example BMP images that you can compress. Feel free to use your own BMP images as well.

Look at the overall structure of the classes and methods in all files of the starter code. You don't need to understand exactly how everything works yet, but it's a good idea to get a general sense of the structure beforehand. No code needs to be written in this task – it's all about the understanding. As a verification that you have understood the code well enough, you should be able to answer the following questions.

1. BMP and QDT are the two file types involved in the input and output of this program. What do they share in common and how do they differ? What is stored in the header and body of a BMP/QDT file?
2. Note that we store a "preorder_list" as the body data of the QDT file. When reading a QDT file, we read the preorder_list from the file, and we can somehow restore the tree structure from the preorder_list. Why is this even possible? Isn't it possible that different trees can have the same preorder traversal? How could we make sure that the tree structure we restored is unique?
3. What are the different classes of quadtree nodes? What does each type of node represent in terms of the image? Why does some class have a "value" while others don't?
4. Why is it necessary to have a class for an "empty" node? What could go wrong if we don't have it?
5. How many children can an internal node in a tree have? Exactly 4, no more than 4, or other? The children are also required to be in a certain order in terms of which quadrants they represent. What's the order?
6. The `Compressor` class does a special thing when compressing a BMP image: it converts an RGB coloured image into a grayscale image. Where in the code does that happen? Also, the pixels list we pass to the `build_quad_tree()` method is different from the raw pixel info stored in the image file – it is about 1/3 of the size of raw pixels list. Can you explain how that works?
7. Why is the loss level between 0 and 255? What does a loss level 0 or 255 mean to the quality of the image after compression?
8. What are the input/output filenames used by the compressor and decompressor? What is ".bmp.qdt" and what is ".bmp.qdt.bmp"?

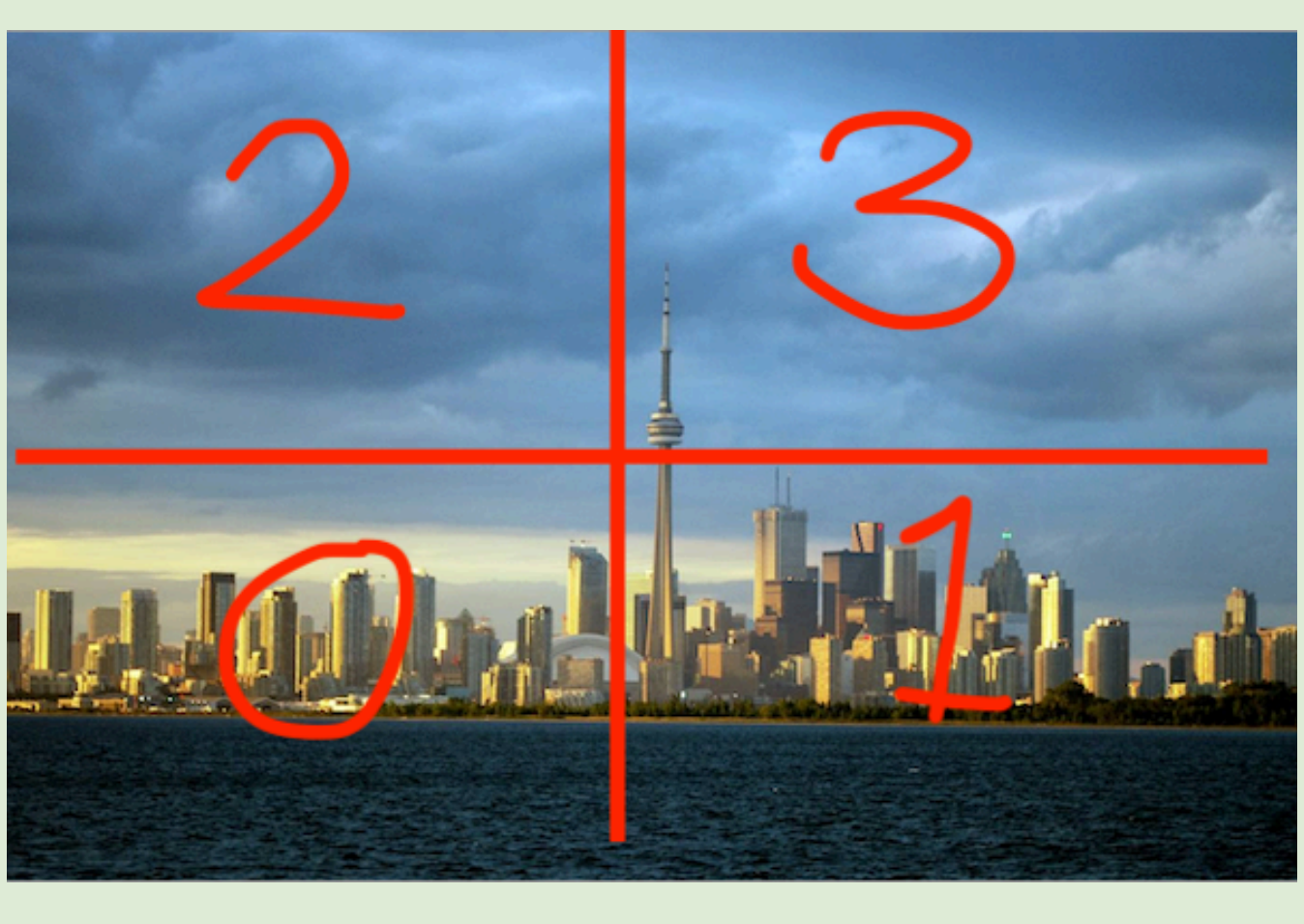
## Task 2: Compression

In this task, you'll implement the compression related methods. The main method used for compression is the `build_quad_tree()` method in the `QuadTree` class, which utilizes a helper method `_build_tree_helper()` that you will implement. The `_build_tree_helper()` should be implemented with recursion. What are the base cases? Make sure to think about this carefully! Also, somewhere while building the tree, you'll need to compare the standard deviation of a quadrant with the loss level and decide whether to turn that quadrant into a leaf node. **IMPORTANT**: the condition for compressing a quadrant is the standard deviation being **LESS THAN OR EQUAL TO (<=)** the loss level. You must implement this condition exactly; otherwise, you could fail some test cases unexpectedly.

Note: The leaf node class requires an integer as its value. You should get this integer by rounding the mean colour of the quadrant using the built-in **round()** method of Python.

**NOTE: The indices of the pixels in a BMP image start from the bottom-left corner (i.e., bottom left is 0,0), going from left to right, and then row by row from the bottom to the top of the image. In other words, the "image" view and the "list" view of the pixels are opposite to each other, i.e., the bottom of the "image" view corresponds to the top of the "list" view. Be very careful here!**

The `_build_tree_helper()` method will use the `_split_quadrants()` method to the split the pixel matrix (a list of lists of ints) into four quadrant with (roughly) equal sizes. Note that this is a static method, which means that it does not have to interact with any class attributes. **IMPORTANT (UPDATED)**: When dividing an **odd** number of entries in half (i.e., they can't be divided evenly and sizes of the two halves have to differ by 1), the **left** half and the **bottom** half (i.e., the half with **lower** indices) must be the smaller one of the two halves. See the docstest of `_split_quadrants()` for a concrete example. Any division that's not following this rule will cause failure of test cases while marking! **ALSO IMPORTANT (UPDATED)**: The return value of `_split_quadrants()` is a list of four list-of-lists representing the four quadrants. They MUST be in a certain order (**bottom-left, bottom-right, top-left, top-right**). Read the docstring of the method carefully for the requirement. See the image below that illustrates the order of the four child quadrants.



Another method needed by the compression is `preorder()`. This method serializes the quad tree into a string representation by performing a preorder traversal on the tree. The string is then written into the QDT file (see `flatten_body()` in the `QuadTreeFile` class). The `preorder()` method in the `QuadTree` class simply calls the `preorder()` method on the root node of the tree. You need to make sure the `preorder()` method is properly implemented for all quad tree node classes. **IMPORTANT**: The return value of `preorder()` has a specific format that must be followed. Make sure to read the docstrings of the `preorder()` method to follow the format precisely.

## Task 3: Decompression

In this task, you'll implement the decompression related methods. The main method for in the `QuadTree` class is `convert_to_pixels()` which converts the quad tree into the pixel matrix. The `convert_to_pixels()` method in the `QuadTree` class simply calls the `convert_to_pixels()` method on the root node of the tree. You need to make sure the `convert_to_pixels()` method is properly implemented for all quad tree node classes. You'll be putting the quadrants back together, recursively, in this method. Make sure the way you specify the widths and heights of the quadrants is consistent with how you split them in `_split_quadrants()`.

Another method needed by the decompression is `restore_from_preorder()` and you need to implement this method for the `QuadTreeNodeInternal` class. This is a recursive method. All recursive calls share the same preorder list (the preorder string split by commas) as an input parameter, but work on different starting indices of the preorder list. This method has a return value – it returns the number of entries in the preorder list that are processed by the current method call. This is a useful return value. Make sure to use it! **Hint**: What is the base case? You may be able to decide by checking the value of the entry at the a given index of the preorder list.

At this point, it would be a good idea to do some unit testing for `preorder` and `restore_from_preorder`. To test these two methods, you can manually build a tree out of some sample data, and assert that you can rebuild the tree from its preorder traversal.

## Task 4: User Interface

Now you have completed all the compression and decompression functionalities, you might want to actually try it on some real images! The `Compressor` and `Decompressor` classes in `a2main.py` has the methods that you need to run the compression and decompression (namely `run()`). To be able to use these methods conveniently, you want to write a little command line user interface in the `main` block of `a2main.py`. Below is an example of what the UI could look like.

```
>>> python a2main.py

Quad Tree Image Compression
============================
Input 'q' at any point to terminate the app

Command [c -> Compress | d -> Decompress] : c
Loss [between 0-255] : 42
File Name: dog.bmp
Compressing dog.bmp
mirror = False
Built a tree with size: 345
Saving to file: dog.bmp.qdt
Compression Done

>>> python a2main.py

Quad Tree Image Compression
============================
Input 'q' at any point to terminate the app

Command [c -> Compress | d -> Decompress] : d
File Name: dog.bmp.qdt
Decompressing dog.bmp.qdt
loaded a tree with size: 345
Saving to file: dog.bmp.qdt.bmp
Decompression done.
```

Since you won't submit `a2main.py`, this task will NOT be marked. I know, we could have provided you with the UI code, but we decided not to, because we want you to have an opportunity to practice with understanding and properly using the interface of classes that are not written by you. Also, this creates an opportunity for you to be creative: you should try to make the UI the way you think is cool. (Start compression by Meepo pushing a block? Be my guest!)

When you compress a BMP file then decompress it, and you actually see an image that is a lossy version of the original, that's a moment worth some celebration!

Also, it is a great opportunity to do some interesting experiments here. Does the compression actually make the file size smaller? If so, how much smaller? How does the loss level affect the size of the compressed file? Does certain kind of images (in terms of the "colour distribution") benefit more from quadtree compression than others? Can you explain why?

## Task 5: Additional Operations

We're not done yet. In this task, we will implement some additional operations for the quad tree, for more practice, and more fun.

### Task 5.1: Tree Size

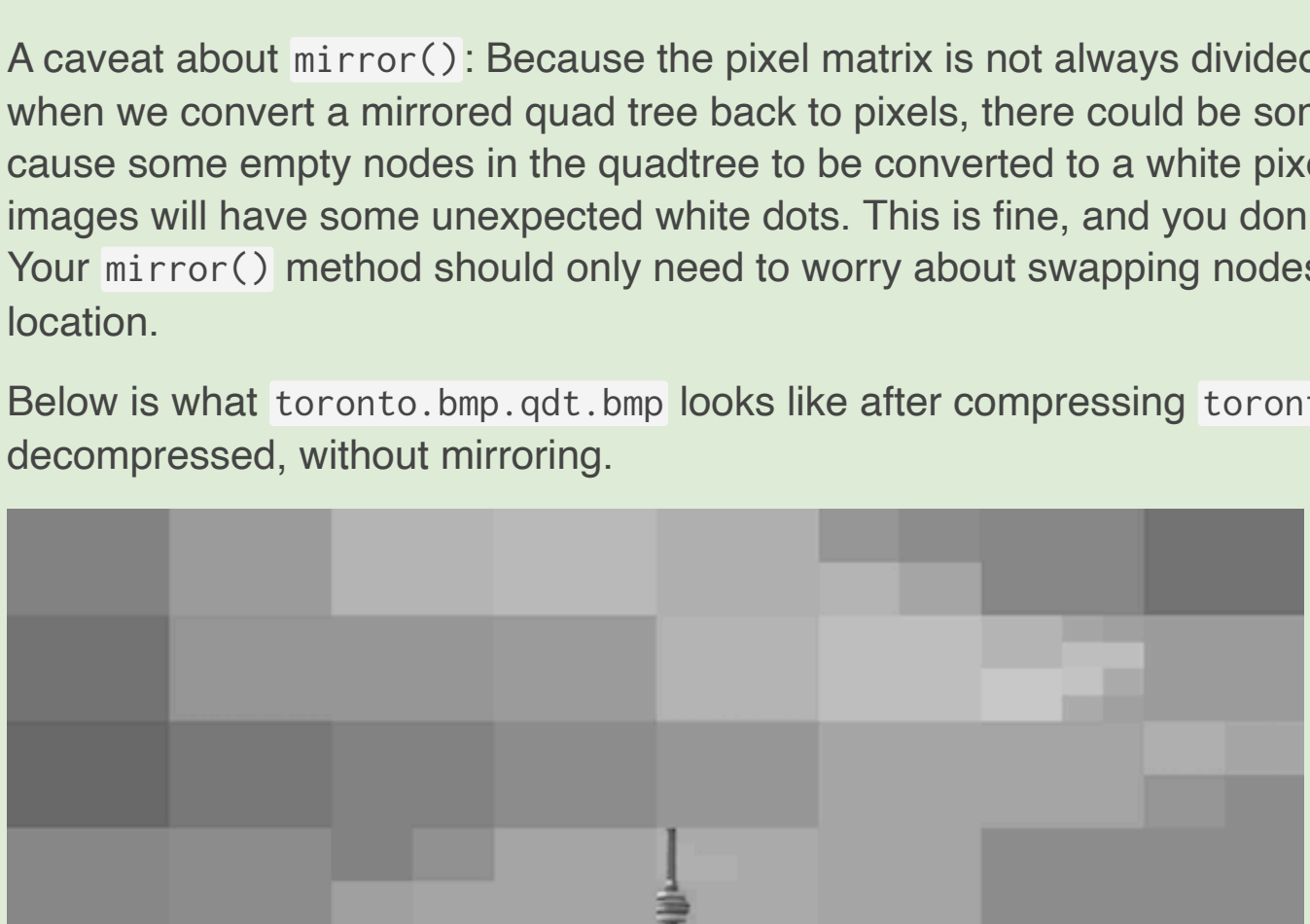
The `tree_size()` method returns the number of nodes in a subtree, including all the Empty, Leaf, and Internal nodes. This method should be fairly straightforward to implement when done with recursion. You will write the `tree_size()` method for all the quadtree node classes.

### Task 5.2: Mirror

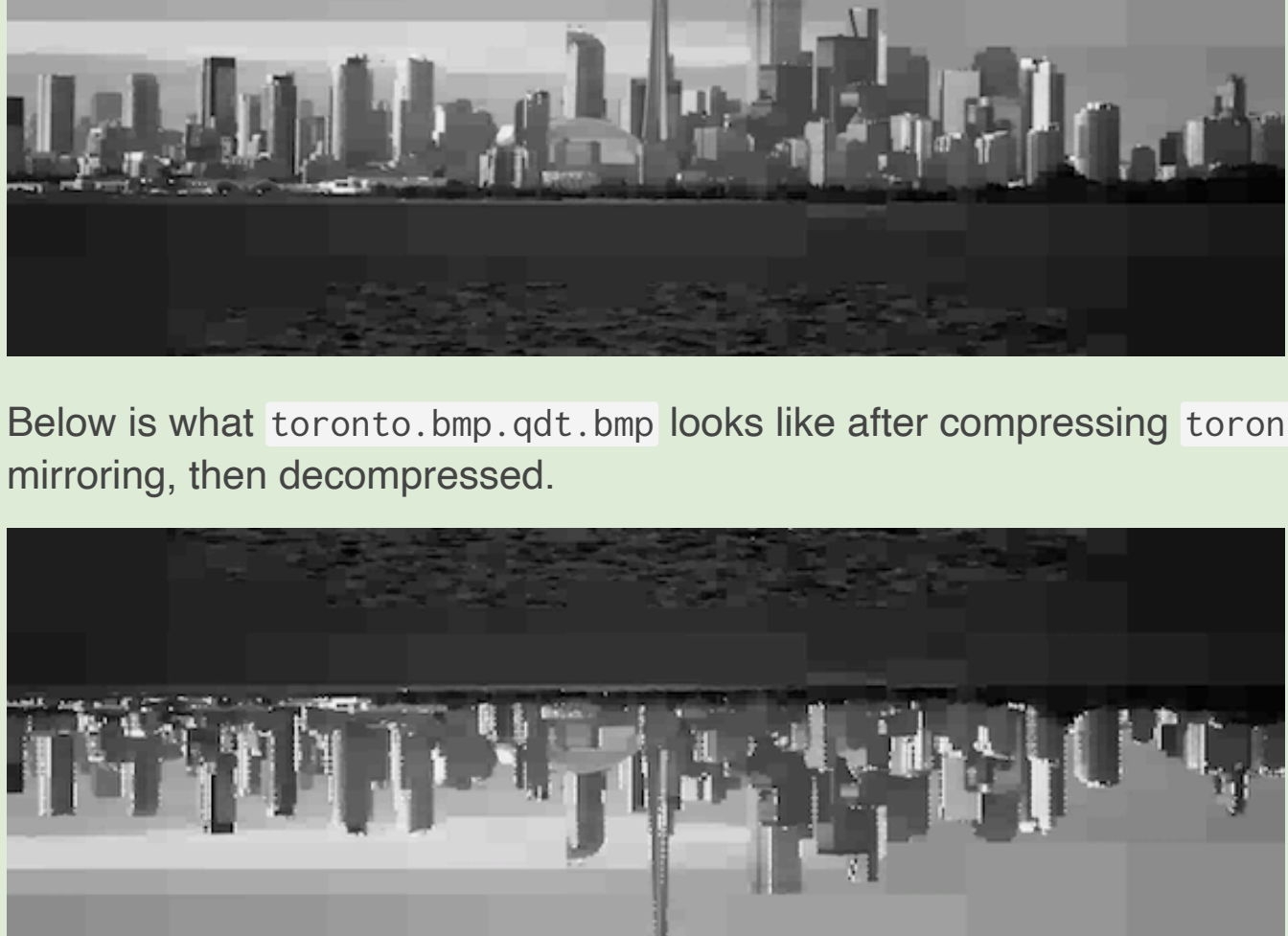
The `mirror()` method rearranges the nodes in the quadtree so that the image will appear to be the bottom half mirrored over the top half. See the two pictures below for an example. You will implement this method in the `QuadTreeNodeInternal` class. During compression, after building the quadtree, this method will be called on the root of the quadtree if the global variable `MIRROR_IMG` is set to True.

A caveat about `mirror()`: Because the pixel matrix is not always divided evenly between the quadrants, when we convert a mirrored quad tree back to pixels, there could be some size mismatches that will cause some empty nodes in the quadtree to be converted to a white pixel. As a result, the mirrored images will have some unexpected white dots. This is fine, and you don't need to worry about fixing it. Your `mirror()` method should only need to worry about swapping nodes/subtrees to the desired location.

Below is what `toronto.bmp.qdt.bmp` looks like after compressing `toronto.bmp` with loss level 15 then decompressed, without mirroring.



Below is what `toronto.bmp.qdt.bmp` looks like after compressing `toronto.bmp` with loss level 15 and mirroring, then decompressed.



### Task 5.3: Maximum Loss

Suppose you're given two quadtrees built from the same image: one with loss level 0 (original), and the other with some unknown loss level (compressed). We want to estimate the loss level used by the compressed quadtree. How do we do this?

The `maximum_loss()` function will take 2 quadtree **roots** (one of the original image and one of the compressed) and simultaneously traverse both trees looking for signs (and magnitude) of compression. The way one would detect compression is actually pretty simple. Think about what it means if you run into a leaf in the compressed tree but not in the original one.

Let's say we find a node which is a leaf in the compressed image but not in the original one. This means that in the compressed image, this node has a single constant colour whereas in the original image, there are multiple colours. Now you may be wondering how does knowing this give us the loss level. Well, it is actually really simple. The loss level for that quadrant in the compressed tree is simply the standard deviation of the colours in the same quadrant in the original version of the image (*Spend some time thinking about why that is the case*).

We know that the loss level used to generate the compressed quadtree must be **greater than or equal to** the loss level of any compressed quadrant. Therefore, the **maximum** loss level across all quadrants of the image will give us a good estimate of the loss level that was used initially.

Note: We return **float** values for the loss level that we compute for each quadrant for better precision in testing.

Note2: When testing this method, to make sure the original tree is uncompressed, you may initialize it with a negative loss level. This is because the tree can be compressed even when the loss level is 0 (why?).

## Task 6: Test Cases

You're required to submit a few test cases for this assignment. Specifically, you'll need to submit the following:

- 3 test cases for `split_quadrants()`
- 3 test cases for `restore_from_preorder()`

The starter file `a2test_student.py` provides a template for these 6 required test cases. You may simply populate them with valid test cases. Of course, you may, and should, include more test cases to test your code more thoroughly.

When we mark your test cases, you'll be given full marks as long as the above six test cases are provided and are valid.

## Task 7: Plagiarism Acknowledgment

You **must** read the following acknowledgment **carefully** and submit a `plagiarism.txt` file, with the following paragraphs:

I, ________ *[FULL NAME HERE]* ________ declare that I have not committed an Academic Offence on this assignment.

Specifically:

- I have read the University of Toronto's Code of Behaviour on Academic Matters and am aware of what constitutes plagiarism.
- No code other than my own (plus starter code) has been submitted.
- I have not received any pieces of code from others (including test cases).
- I have not obtained pieces of code available publicly, nor modified such code to pass as my own work.
- I have not shared any parts of my code with others (including test cases), nor shared specific details on how others may reproduce similar code.
- I did not instruct another classmate on what to write in their assignment code.
- I did not receive instruction on specifically what to write in my code, and have come up with the solution myself
- I did not look for assignment solutions online.
- I did not attend private tutoring sessions (including in other languages) which are not explicitly sanctioned by UTM where the assignment was discussed.
- I did not post my own code publicly online on places like GitHub, pastebin, StackOverflow, etc.
- I acknowledge that this declaration is truthful and does not include any misrepresentation. I acknowledge that not being able to explain in detail my own work will result in a zero on this assignment, and that if the code is detected to be plagiarized, severe academic penalties will be applied when the case is brought forward to the Dean.

**Failure to include this file will result in a grade of 0 for the assignment.**

## Submission instructions

1. Login to MarkUs and navigate to the Assignment 2 submission page.
2. **DOES YOUR CODE RUN**?! Does it pass your thorough your test suite? Make sure you're testing your code using the correct version of the software (e.g., Python 3.8).
3. Submit the files `a2tree.py`, `a2test_student.py`, and `plagiarism.txt`. You do not need to submit any other files.

## Marking Scheme

Below is the tentative marking scheme of this assignment:

- Compression (autotesting): 30%
- Decompression (autotesting): 30%
- Additional operations (autotesting): 30%
- Test cases in `a2test_student.py`: 10%

We will NOT mark by PyTA errors for this assignment. However, you should still try your best to write your code in a good style, and running PyTA yourself can help.

## Congratulations!

You have finished Assignment 2 of CSC148! Go create some arts with your quadtree compressor!