Goals and Overview
The Assignment, Part 1: Inspiration and B…
QuadTrees on a Torus
Entropy
The Assignment, Part 2: Coding
Problem Specification
Implementation Constraints and Advice
Getting the Given Code
Testing your code
Handing in your code
Good luck!

PA3 The Art of Compression

Due: Friday, April 5 at 11:59 PM

UPDATES!

NOTE: Clarifications, updates, and hints to solving this PA can be found on this piazza page: https://piazza.com/class/jpu51kyvbx9ps?cid=1779

Lab machines

We will be grading the PAs for this course using an image similar to the remote Linux machines. Any errors that you run into as a result of developing your code on other machines are your own responsibility.

Goals and Overview

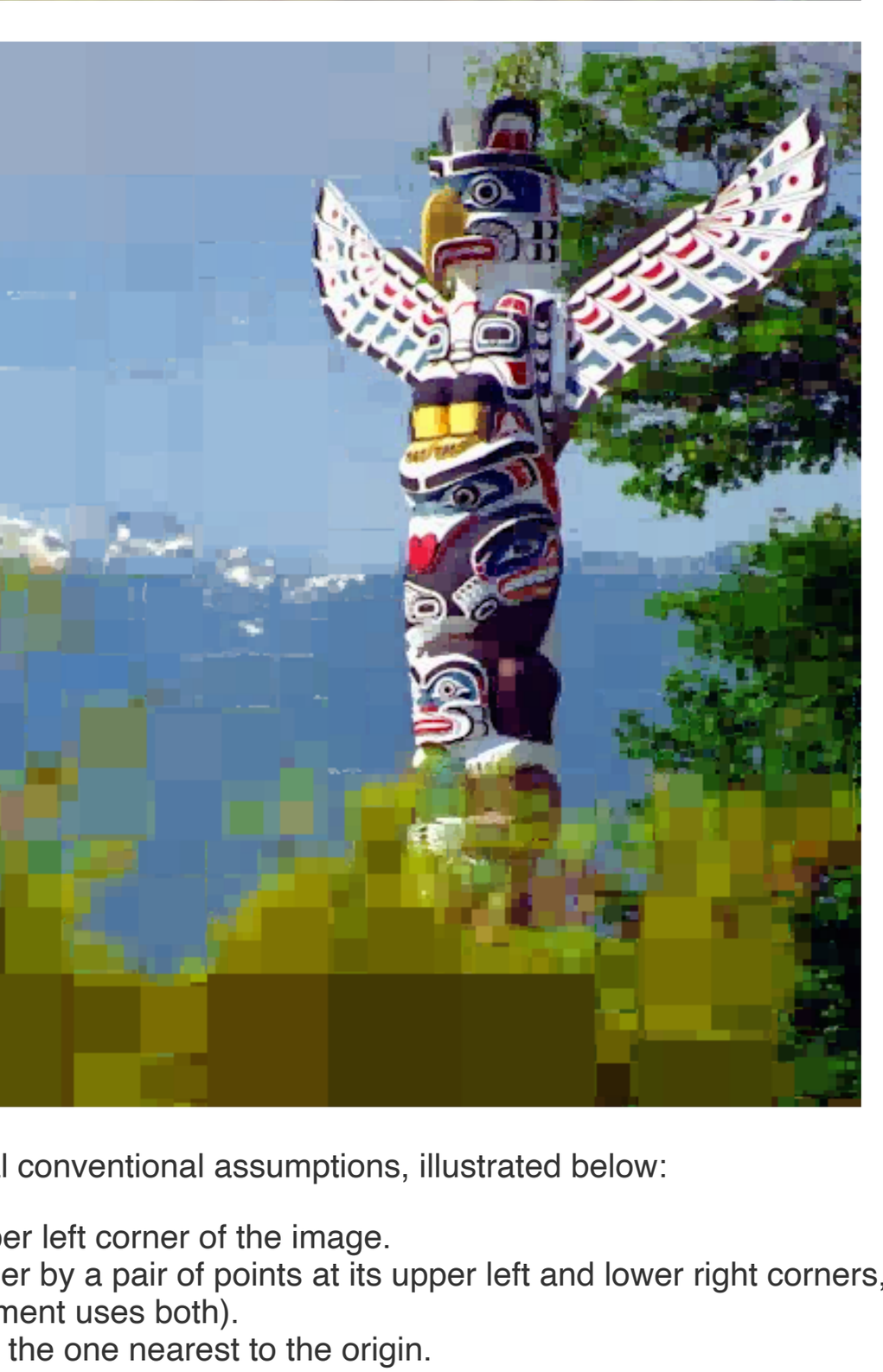
In this PA (Programming Assignment) you will:

- learn about algorithmic art.
- learn about an unusual algorithm for lossy image compression.
- learn about space partitioning trees called Quad Trees.
- learn about clever mechanisms for speeding up statistical algorithms.
- learn to be aware of memory usage in recursive programs.

The Assignment, Part 1: Inspiration and Background

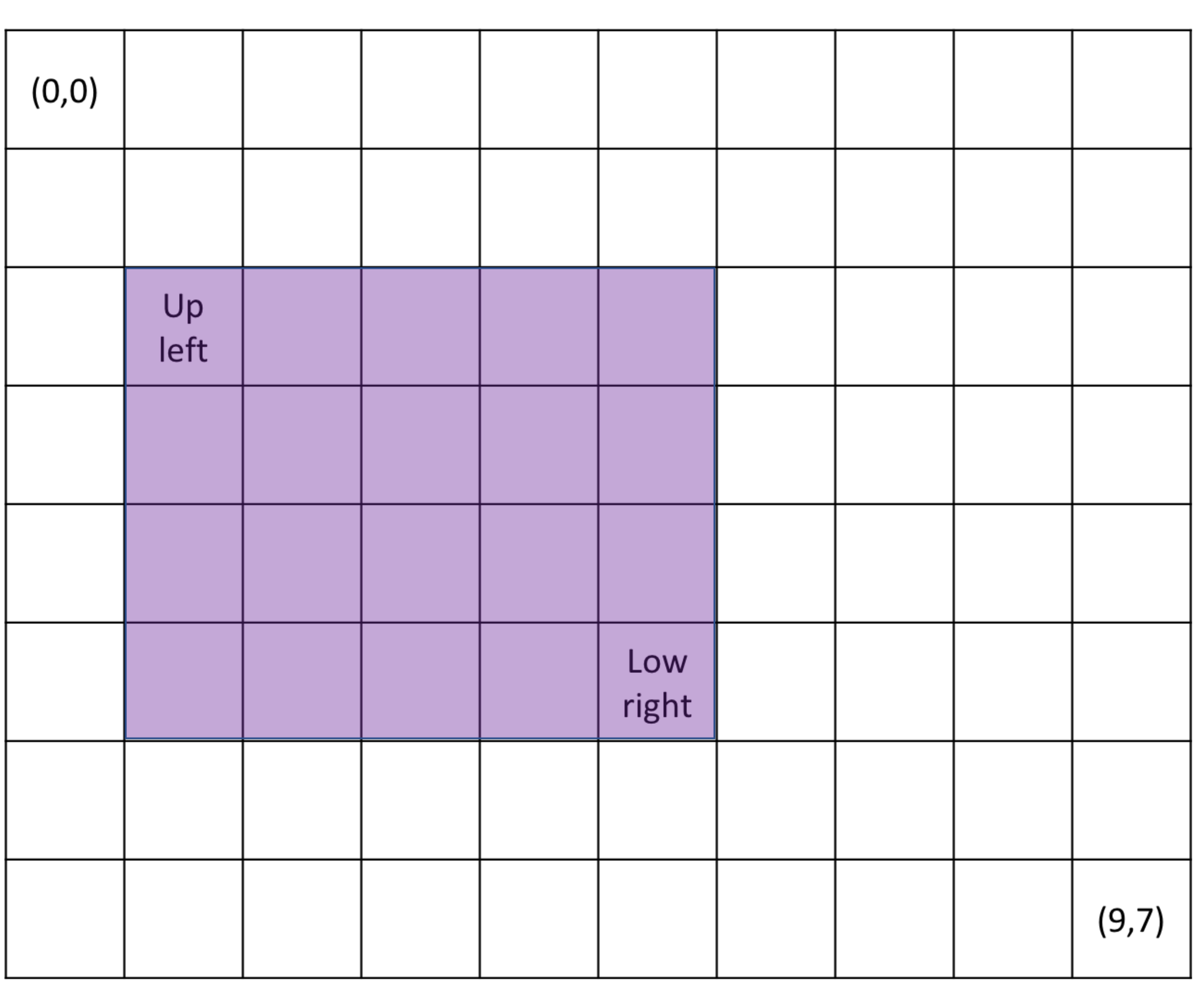
The inspiration for this assignment came from an article about an artist whose work recreates classic portraits with a blocky effect. The exact algorithm he used is not given in the article, but we will create similar images using a strategy for image representation that underlies common lossy image compression algorithms.

The two images below illustrate the result of this assignment. Note that the image on the right sacrifices color detail in rectangles that do not contain much color variability in the original image, but maintains detail by using smaller rectangles in areas of the original image containing lots of variability.



In specifying the algorithm we make several conventional assumptions, illustrated below:

- The origin, position (0,0), is in the upper left corner of the image.
- Rectangles are typically specified either by a pair of points at its upper left and lower right corners, or by an upper left point, together with dimensions (this assingment uses both).
- The upper left corner of a rectangle is the one nearest to the origin.
- Image locations are typically specified as (x,y), where x is a horizontal offset, and y is a vertical offset.



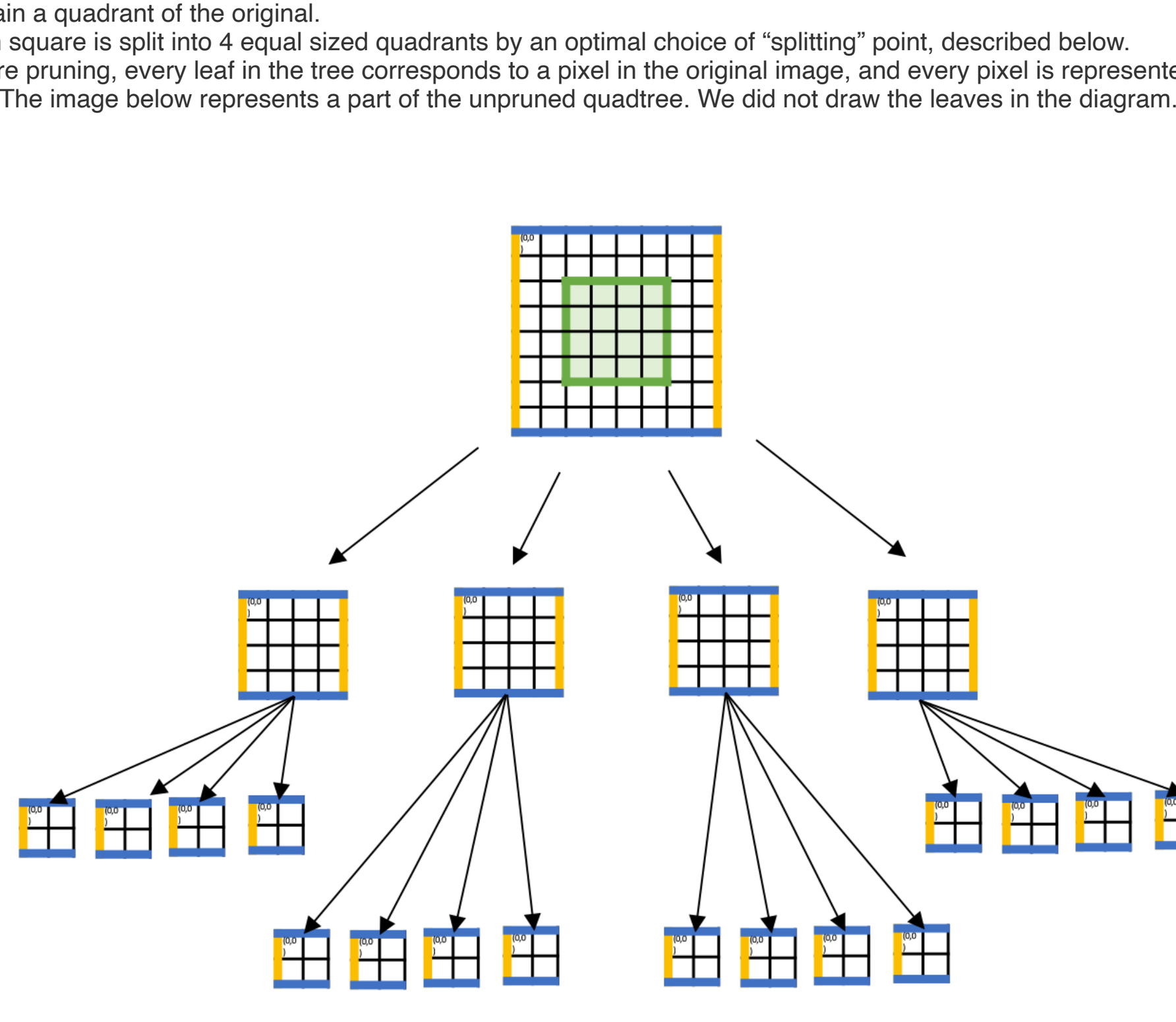
QuadTrees on a Torus

The original image will be represented in memory as a 4-ary tree whose nodes contain information about square subsets of the image.

- Every node is the root of a subtree that represents a subset of the original image. Note that the region might not be contiguous in the original image.
- Every node also contains a pixel that represents the average color over the square in the original image.
- Every node contains a dimension that determines the size of the square it represents. If dimension == d, then the node represents a

  $$2^d \times 2^d$$

  square.
- When a square is split into four smaller squares, the parent node represents the whole square, and the 4 children each contain a quadrant of the original.
- Each square is split into 4 equal sized quadrants by an optimal choice of “splitting” point, described below.
- Before pruning, every leaf in the tree corresponds to a pixel in the original image, and every pixel is represented by some leaf. The image below represents a part of the unpruned quadtree. We did not draw the leaves in the diagram.



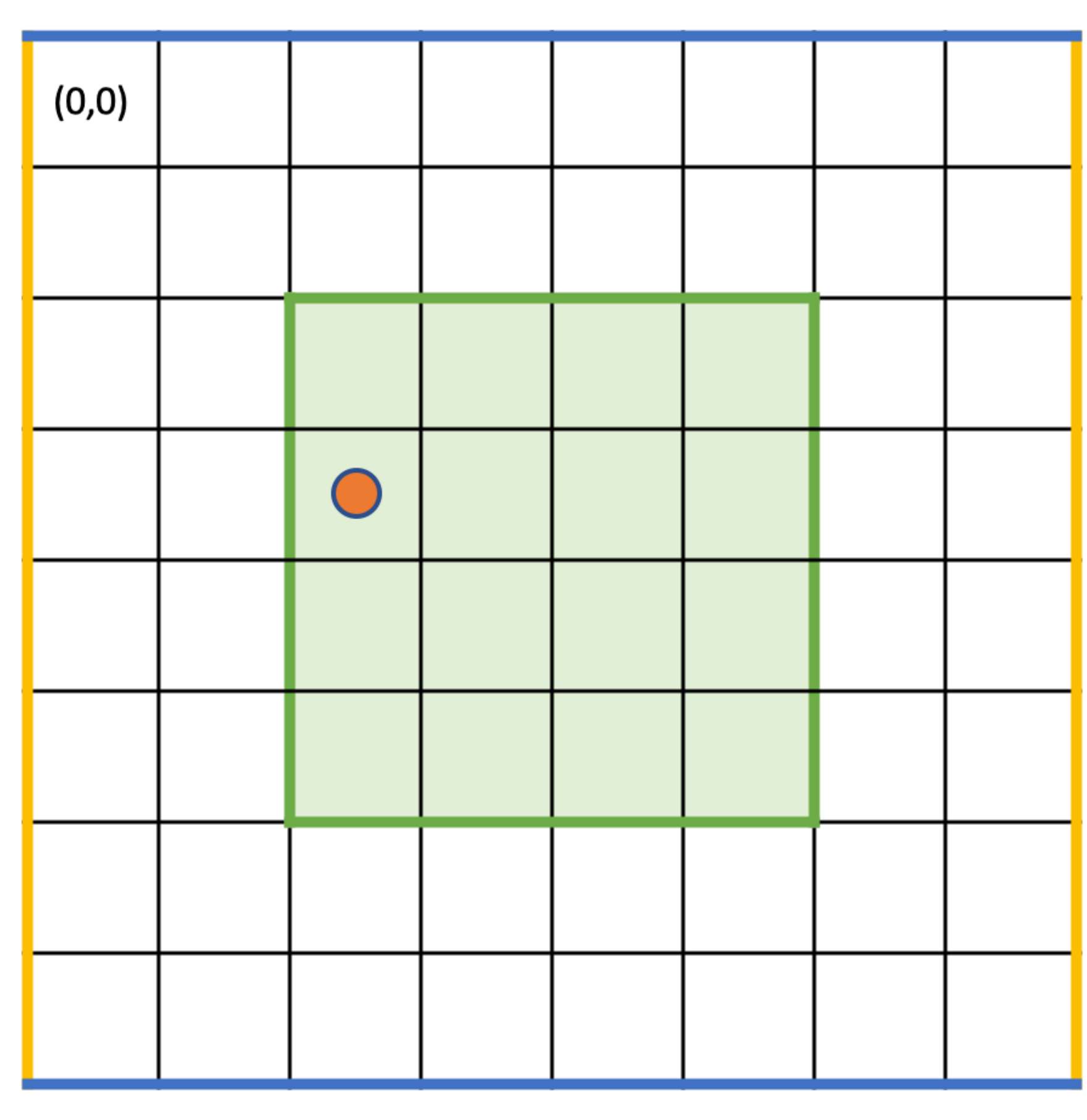
Consider every node to represent a square image that you will subdivide into 4 children nodes. In order to determine how to partition the square, you will find a splitting point from among the points in the light green region in the image below (in general, this will be a

$$2^{k-1} \times 2^{k-1}$$

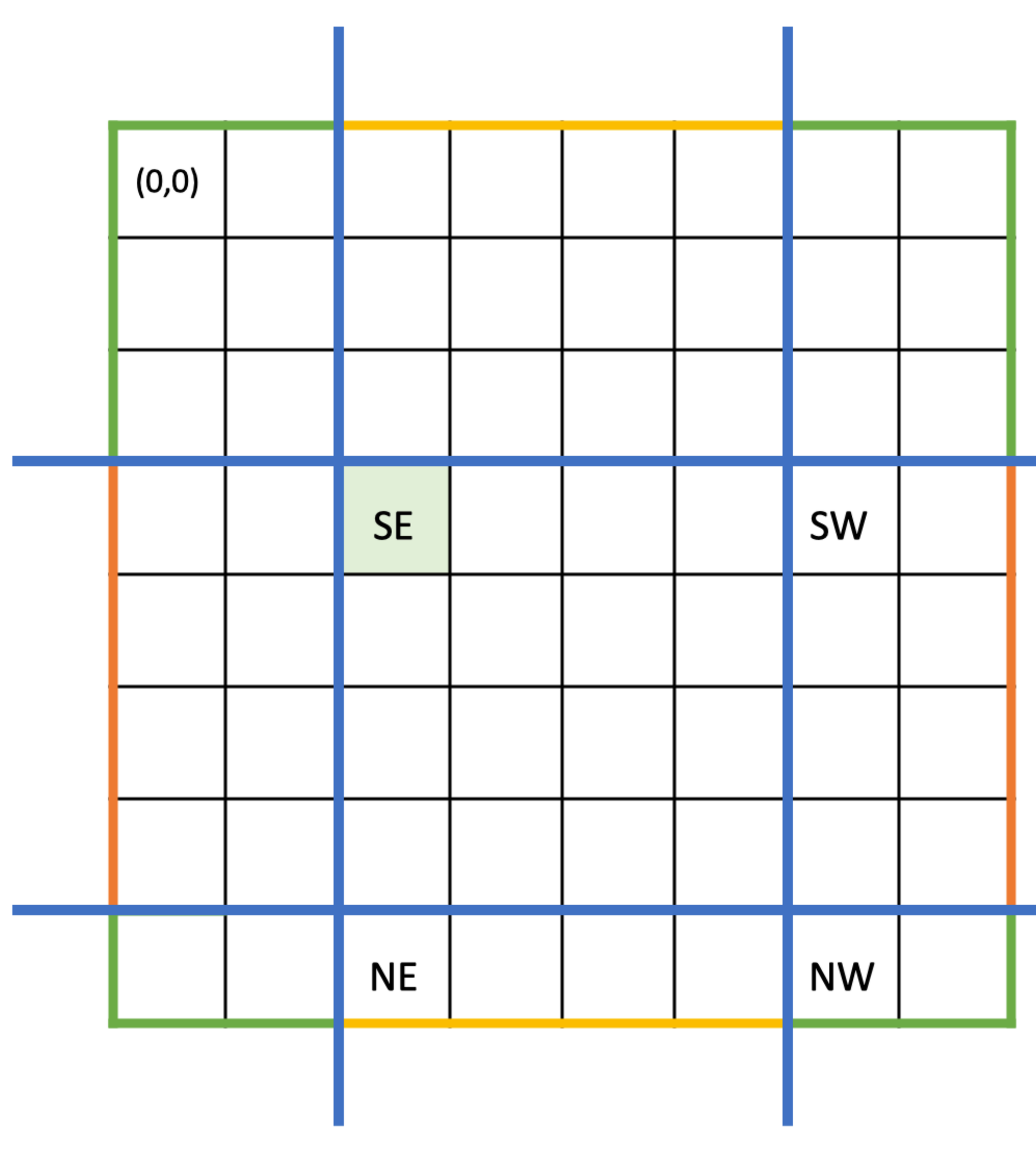
square in the middle of the

$$2^k \times 2^k$$

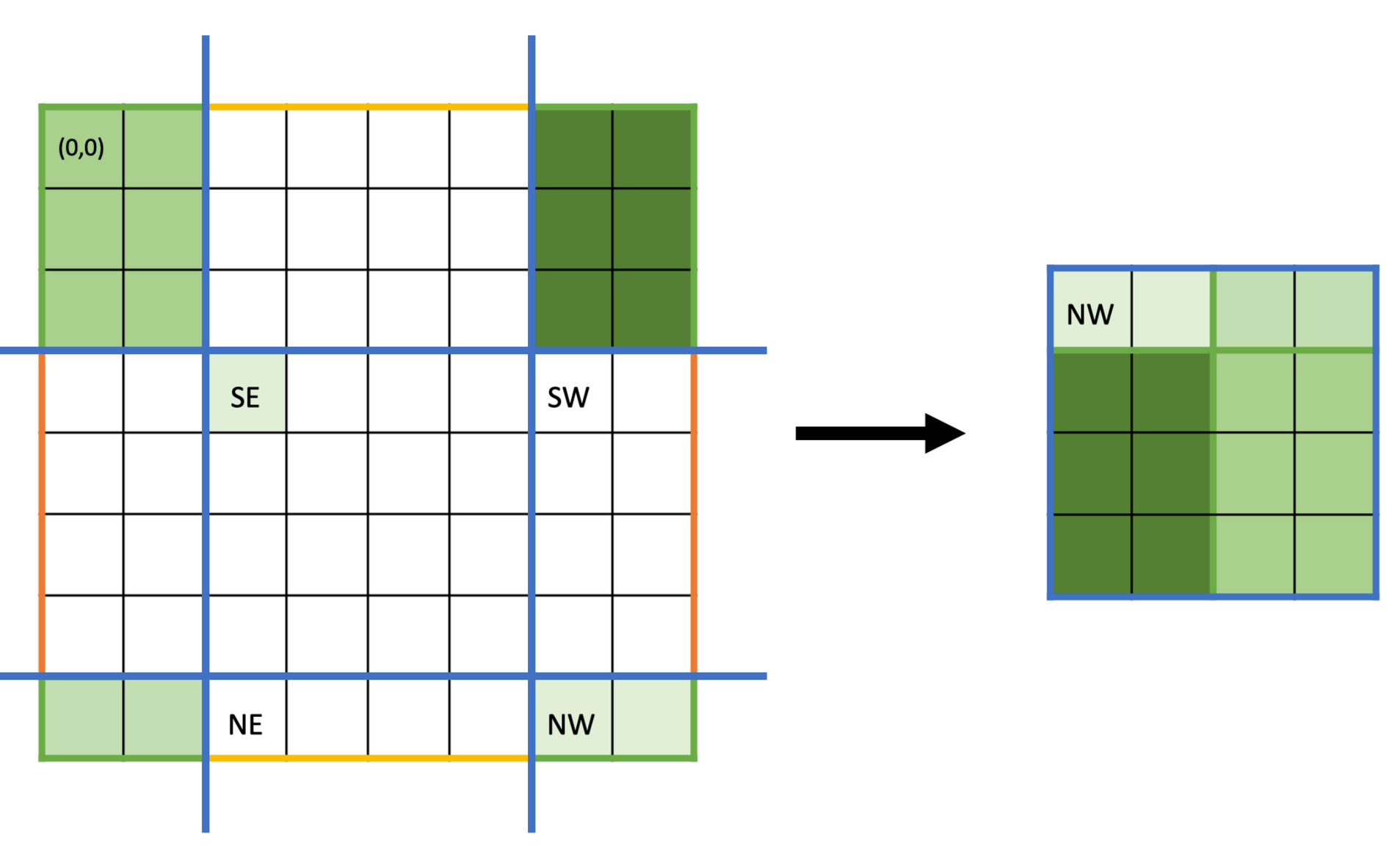
image:



Traverse the pixels in the green square, looking for the one which minimizes the average entropy (described below) over the 4 squares it determines. In this example, we suppose the pixel containing the orange dot gives the minimum entropy score. That orange dot pixel will become the upper left corner of the SE quadrant of the 4-way split.

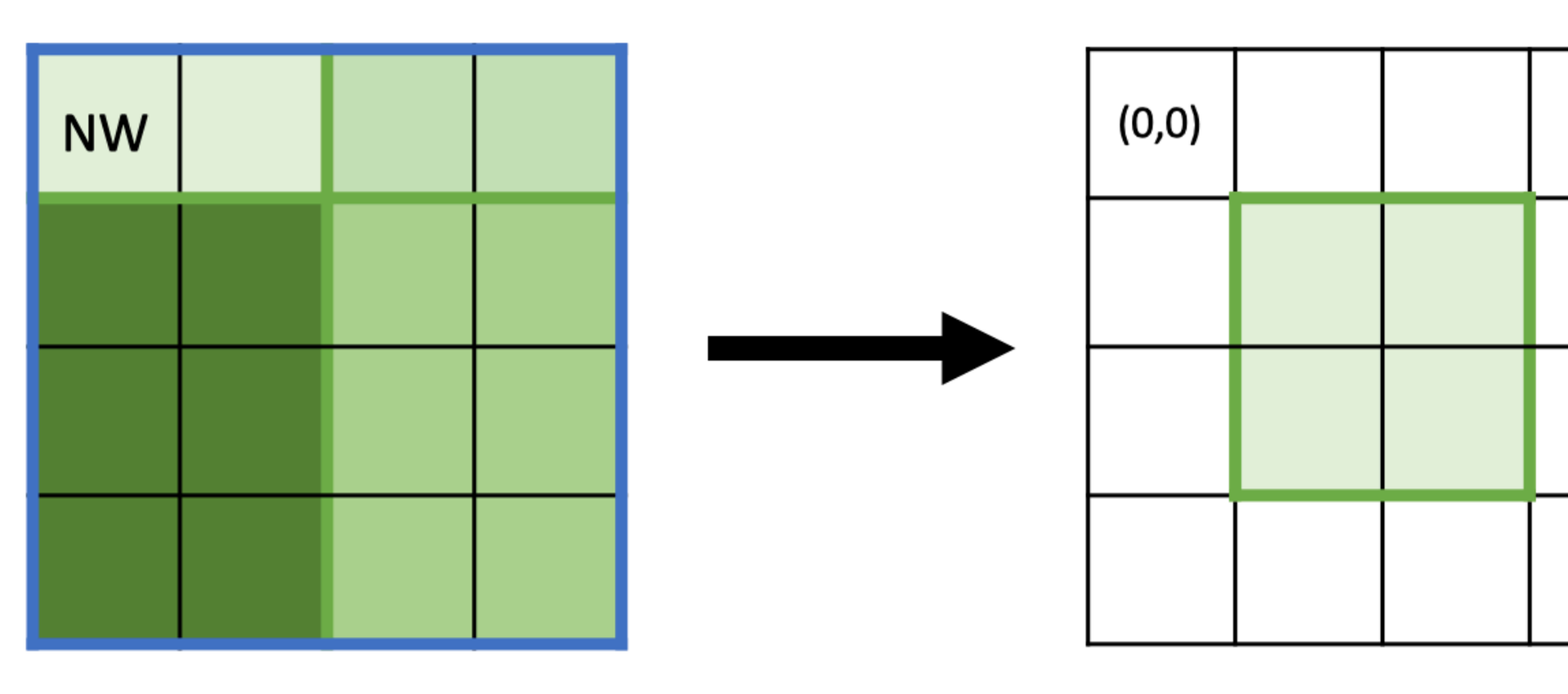


The center found in the illustration will be the upper left corner of the square that we will call SE, for Southeast because it is below and to the right of the optimal splitting point. The squares marked NW, SW, and NE are the upper left corners of the 3 other squares determined by the split. You will need to compute these locations, given the optimal splitting point, and considering that the edges of the square image wrap around to meet.



Once you have an optimal splitting point, and the upper left corner of all 4 squares, you will explicitly create the 4 squares as the bases for the current node’s children. The illustration shows you how to build the NW child.

Note that in the general case, one of the 4 squares will be a whole square subset of the current image, one will be a horizontal shift into segments where part of the square is on the right of the image, and part is on the left, corresponding to SW in the example above (note that the number of shifted locations might be 0), one will be a similar vertical shift (again, where the shift might be 0), and one will be both a horizontal and a vertical shift (again, where one or both directions’ shifts might be 0).



Finally, the squares you reconstruct out of a partition of the current square are (recursively) the basis for the child nodes.

Entropy

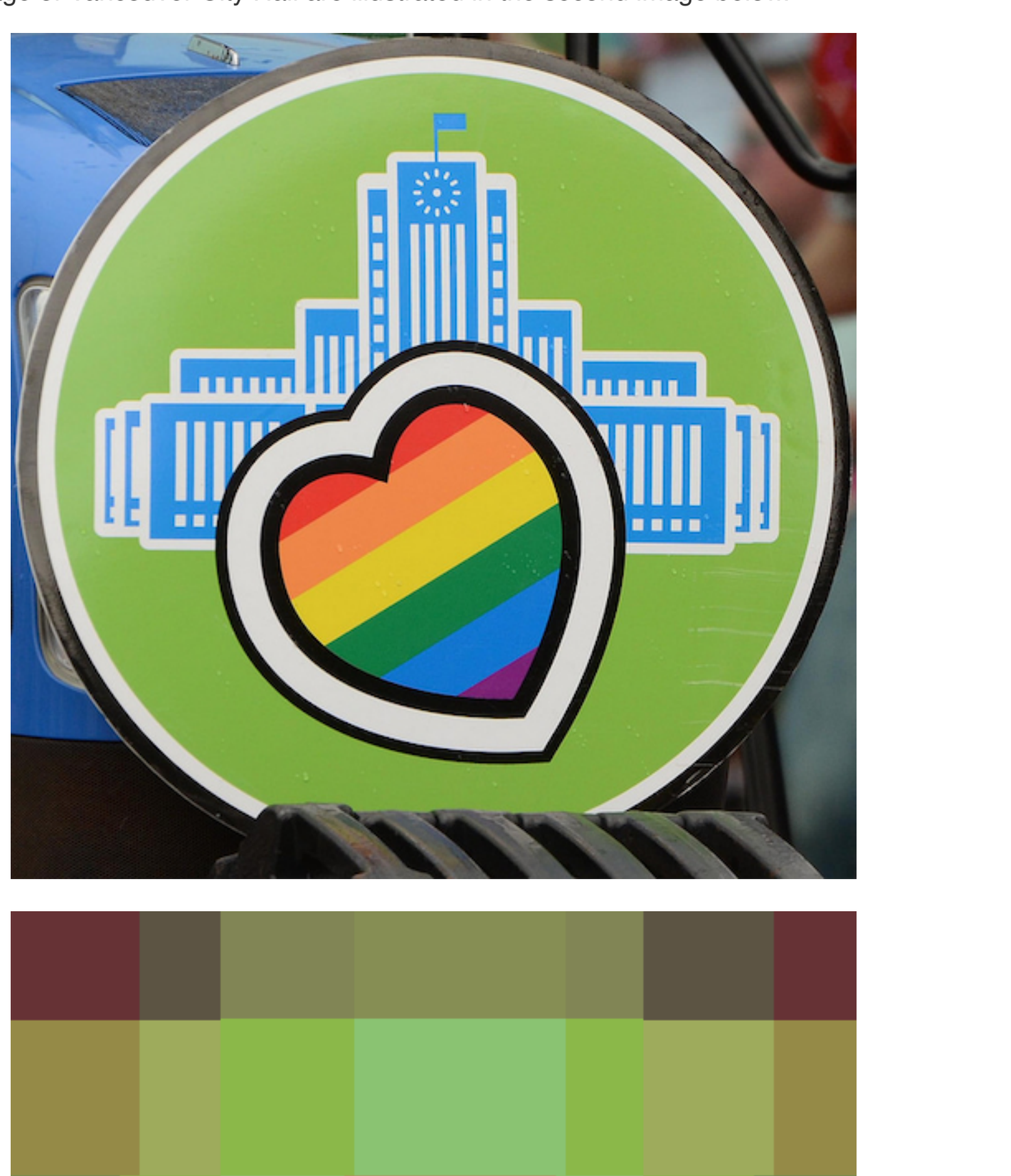
Splits are performed so as to minimize hue variability across the resulting squares. The variability measurement we use is the *entropy*, which is computed as a function of the distribution of colors in a rectangle as follows: Given a relative frequency table of the colors over some rectangle, where p_i represents the fraction of pixels whose hue, h, is $10 \cdot i \leq h < 10 \cdot (i+1)$, the entropy for the rectangle is

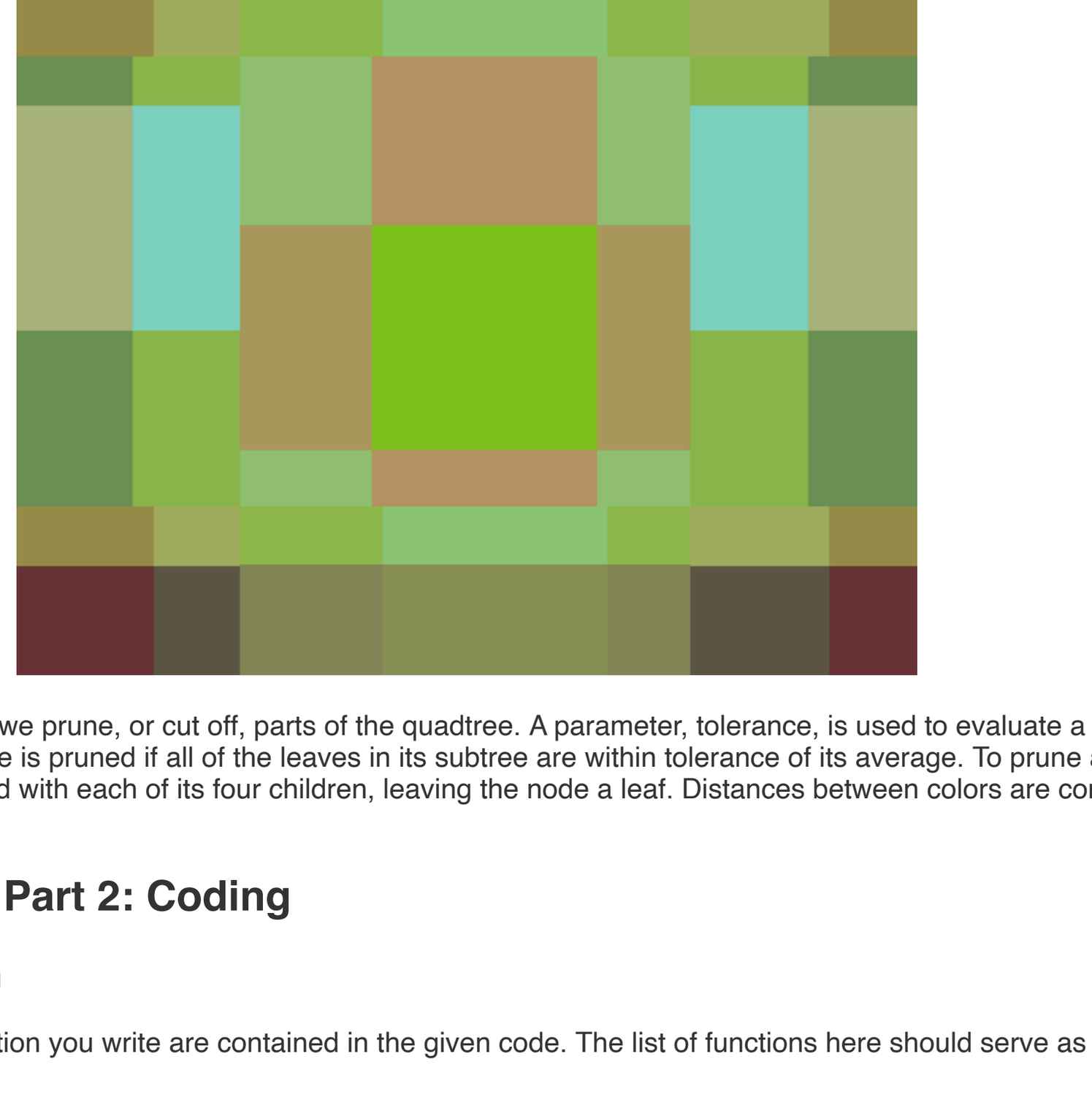
$$-\Sigma_{i=0}^{35} p_i \log_2(p_i).$$

The entropy of a *split* is the average of the entropy of the 4 resulting squares. (In the case of ties, we choose the split which occurs first in a column-wise navigation of the feasible region of splits.)

As part of this assignment, we build a data structure that allows us to find the histogram of colors for any rectangle, in constant time. To that end we build a 2D vector structure whose cells, (i,j), each contain a frequency table for the rectangle with upper left corner (0,0), and lower right corner (i,j).

The first two splits of the image of Vancouver City Hall are illustrated in the second image below.





To achieve an artistic effect we prune, or cut off, parts of the quadtree. A parameter, tolerance, is used to evaluate a subtree’s suitability for pruning. A node is pruned if all of the leaves in its subtree are within tolerance of its average. To prune a node, simply clear the memory associated with each of its four children, leaving the node a leaf. Distances between colors are computed using the HSLAPixel dist function.

The Assignment, Part 2: Coding

Problem Specification

Specifications for each function you write are contained in the given code. The list of functions here should serve as a checklist for completing the exercise.

In class stats:

- stats(PNG & im)
- HSLAPixel getAvg(pair<int,int> ul, pair<int,int> lr)
- long rectArea(pair<int,int> ul, pair<int,int> lr)
- double entropy(pair<int,int> ul, pair<int,int> lr)
- double entropy(vector<int> distn, int area)

In class toqutree:

- void clear(Node * & croot)
- Node * copy(const Node * croot)
- toqutree(PNG & imIn, int dim)
- PNG render()
- void prune(int tol)
- int size()
- Node * buildTree(PNG * im, int k)

Implementation Constraints and Advice

We will be grading your work on functionality, efficiency, and memory use.

The stats class is used only for constructing a node of the toqutree. The choice of optimal split requires the computation of statistics over each candidate split, and these are simply too expensive to compute in real time, especially given the number of squares in our recursively defined tree! Instead, we precompute support structures that allow the necessary statistics to be computed in constant time. We request a particular set of structures, but we leave it as a puzzle for you to figure out how to use them to get the info you need.

The toqutree class is a standard quadtree class, with a twist. We consider a square image to be a torus or a donut, in the sense that the right side of the image is considered to be adjacent to the left, and the top of the image is considered to be adjacent to the bottom. We’ve defined the Node class for you, and a few other necessary memory management functions, but everything else is left to you. Since we will be grading both the .h and .cpp files, you are welcome to add helper functions to the class.

Getting the Given Code

Download the source files from pa3.zip, and follow the procedures you learned in lab to move them to your home directory on the remote linux machines.

Testing your code

We have provided two avenues for your exploration and testing, neither of which is sufficient to completely verify the correct implementation of all of your functions. You should add additional tests yourself!

When you type make two executables are produced. One depends on the given main.cpp, and the other depends on the file testComp.cpp which is a small collection of test cases using the catch framework (as in previous assignments).

Handing in your code

To facilitate anonymous grading, **do not** include any personally-identifiable information (like your name, your University ID number, or your cwl) in any of your source files. Instead, before you hand in this assignment, create a file called partners.txt that contains only the CSIDs of people in your collaboration partnership (if one exists), one per line. If you worked alone, include only your own CSID in this file. We will be automatically processing this information, so do not include anything else in the file. As always, if you’re working in a group, each group member must hand in the assignment. (Failure to cite collaborators violates our academic integrity policy;)

The following files are used to grade PA3:

- stats.h
- stats.cpp
- toqutree.h
- toqutree.cpp
- partners.txt

All other files will not be used for grading.

You will submit your work to gradeScope as you did for PA2.

Good luck!