Hash Tables – Painting Comparison

**Purpose**

This lab was designed to teach you how to use an efficient data structure to solve a real-world problem.

**Overview**

In this project, you'll implement a kind of hash table and use it in an application: to compare images with each other for similarity of color usage. You'll do some experimenting with two different hashing collision-resolution policies: linear probing and quadratic probing. The high-level idea here is "Comparing Composite Objects."

One approach to comparing any kinds of documents is counting the occurrences of different elements they use and then comparing the counts from one document to another. For example, when comparing written essays, you can count how many times the word "therefore" is used in each essay and perhaps conclude that essay B is more logical in style than essay A because essay B uses the word "therefore" 10 times, whereas essay A uses it once or not at all.

For comparing paintings, we can count how many pixels have a particular color in a digitized version of each painting. The painting that has relatively more bright red pixels might be considered to be more lively or otherwise distinguished from the other. We will use a technique that considers all the colors, rather than any particular color like red by itself.

We will use hash tables to count the number of pixels of each color. Each dictionary entry is a pair, with a "ColorKey" as the key, and a nonnegative integer (a count) as the value. Since the space of possible colors can be large but the number of colors actually used tends to be considerably smaller, hash tables are one natural choice for this problem.

When counting colors, there is an interesting subtlety. Two similar colors, when represented with many bits of precision, may appear identical to the human eye but be off by just a little, so that the computer considers them as distinct. This can make comparing colors problematic. One way to address this is to map each color into a smaller-dimensional color space. How small should that color space be? This can be considered a research question, or if this program is to be an "app" then it is a research and development question. You'll try out multiple "ColorKey" spaces for the hashing, and this will have an effect on what the computer comes up with for its measure of similarity between images.

**Purpose**

You will be using arrays to represent two hash tables. The hash tables will have the usual functionality for inserting and retrieving key-value associations. However, the actual API will be a little unusual, with some extra methods that can help in debugging, automatic testing, and automatic grading.

By doing this assignment, you will learn how to implement a hash table, including collision resolution, and you will see how it can be used in an application such as comparing objects such as images or documents.

**Partnership**

You are encouraged, but not required, to work in a partnership of two students for this project. We expect that you will be in the same partnership in both assignments.

**How to Get Started**

Create a Java project in Netbeans. Put the source files into the default package of the src folder. Put the image files into the project folder but not in the src folder.

**What Files are Involved?**

We are giving you certain files. Add code to those which are not complete. Then test your program as specified. Here are the names of the files and what they are for.

1. MonaLisa.jpg: One of the input files, the Mona Lisa painting by Leonardo Da Vinci.

2. StarryNight.jpg: another input file, Starry Night by Vincent Van Gogh.

3. ChristinasWorld.jpg: The painting "Christina's World" by Andrew Wyeth.

4. ColorKey.java: The definition of a class that represents colors at a variety of possible numbers of bits per pixel. The number of bits per pixel controls the size of the key space that we use in hashing. For example, if the number of bits per pixel is 6, then the size of the key space is 26 = 64. This is a relatively small key space, and for a given hash table size, the likelihood of collisions tends to be lower if the key space is smaller.

5. ImageLoader.java: This file is complete. You don't have to add anything here. It loads in requested images and supports accessing the pixels and returning them in a variety of different numbers of bits per pixel, as instances of the ColorKey class.

6. ColorHash.java: a skeleton for your hash table. Most of your work will be adding code to this file.

7. ResponseItem.java: This class is done for you. An instance of this class will be returned whenever get or put operations are performed in a ColorHash hash table. A ResponseItem instance has four fields. int value: meaningful only for get operations (colorHashGet), where the value associated with the key is returned. int nCollisions: the number of collisons involved in this operation. boolean didRehash: true if this operation caused a rehash due to either exceeding the load factor threshold or running out of space (this can happen with quadratic collision resolution). This can only happen when inserting. boolean didUpdate: true if this operation caused the value associated with an existing key to be overwritten (possibly changed but possibly just rewritten with the same value). If the put operation inserted a new key, then false should be returned.

8. FeatureVector.java: a skeleton for a class of objects that represent vectors of long integers (the counts of the different colors in an image). You will implement methods for retrieving the count values from hash tables and comparing a pair of vectors.

9. ComparePaintings.java: a skeleton for your application. Some of your work will be adding code to this file.

**What to Implement in your Hash Table Class**

In the file ColorHash.java, implement the following methods and make them public...

1. ColorHash(int tableSize, int bitsPerPixel, String collisionResolutionMethod, double rehashLoadFactor). This is a constructor. Here is how your hash table should work. The constructor should create a new hash table implemented as one or more arrays. One approach would be to have separate arrays for the keys and the values. An alternative would be to define an inner class to represent a (key, value) pair and have an array of those pairs. That choice is up to you. The naming of these internal members of class ColorHash is up to you. The constructor should support two collision resolution methods: "Linear Probing" and "Quadratic Probing".

2. ResponseItem colorHashPut(ColorKey key, long value). The method colorHashPut should normally insert the key into the hash table and enter the value associated with it. If there already exists an entry for this key, then the new value should be stored, overwriting the old value. (We will be using the values here to represent counts of the corresponding colors.) The response item should be returned that gives all the relevant status of this operation.

3. ResponseItem increment(Colorkey key). The method increment should do the following: If the key is already in the table, get the value, add one to it, and replace the old value with the new value. If the key is not in the table, insert it and store the value 1 with it. Implement this method so that the "find" portion of the operation is performed only once. This should cut the number of collisions to half the number that would be required if it were performed with separate get and put operations. The returned ResponseItem should contain the number of collisions involved in the initial find operation.

4. ResponseItem colorHashGet(ColorKey key) throws MissingColorKeyException. The method colorHashGet should look up the key and return its value. If the key is not there, it should throw an exception with the new type MissingColorKeyException.

5. long getCount(ColorKey key). The method getCount should return 0 if the key is not in the table. Otherwise, it should return the value associated with the key.

6. ColorKey getKeyAt(int tableIndex). The method getKeyAt should return whatever key is currently being stored at location tableIndex in your array. This is not a standard hashing method, but is used in debugging and compliance checking.

7. long getValueAt(int tableIndex). The method getValueAt should return whatever value is currently being stored at location tableIndex in your array. Like getKeyAt, this is not a standard hashing method, but is used in debugging and compliance checking. (Clarification: the question came up of how to handle cases where the table has no entry at the given tableIndex. We suggest returning -1L. However, it will also be acceptable to throw a RuntimeException. The new JUnit test script should allow both of these to pass, within the test of quadratic probing.)

8. double getLoadFactor(). The method getLoadFactor should return the current value of the load factor of your hash table.

9. int getTableSize(). The method getTableSize should return the size of your hash table. Since resizing can take place during a put operation, the size might not stay the same during a session.

10. void resize(). The method resize should allocate a new array (or arrays if you need more than one new one) to build a bigger hash table and then scan the old table from location 0 to location getTableSize()-1, inserting the key-value pairs into the new table using the same approach as with colorHashPut. The new size should be the smallest prime number that is at least double the old table size. You may use the method isPrime in the file IsPrime.java to find out if a number is prime. Before returning from resize, the old array(s) should be replaced by the new one(s), so that they can be garbage-collected.

Note that you are not responsible for implementing any code for the deletion of dictionary items in this assignment.

You may implement any additional methods you like to serve as helper methods or debugging aids. Provide comments for each of these additional methods.

**What to Implement in Your FeatureVector Class**

1. void getTheCounts(ColorHash ch): It will go through all possible key values in order, get the count from the hash table and put it into this feature vector.

2. double cosineSimilarity(FeatureVector other): This will return a double in the range 0.0 to 1.0 containing the result of doing the cosine similarity computation with the current FeatureVector ("this") and the additional one ("other"). You may call these A and B in the code if you wish.

**What to Implement in Your ComparePaintings Class**

The file ComparePaintings.java will be your main application file. It already has some code in it.

There is a separate file FeatureVector.java with a class FeatureVector which has a constructor. You should add the implementation of the two incomplete methods there -- one is for getting the counts of the colors from a hash table. The other is computing the "cosine similarity" between a pair of feature vectors. This is a standard technique in information retrieval and document analysis, and it is often used for comparing web pages.

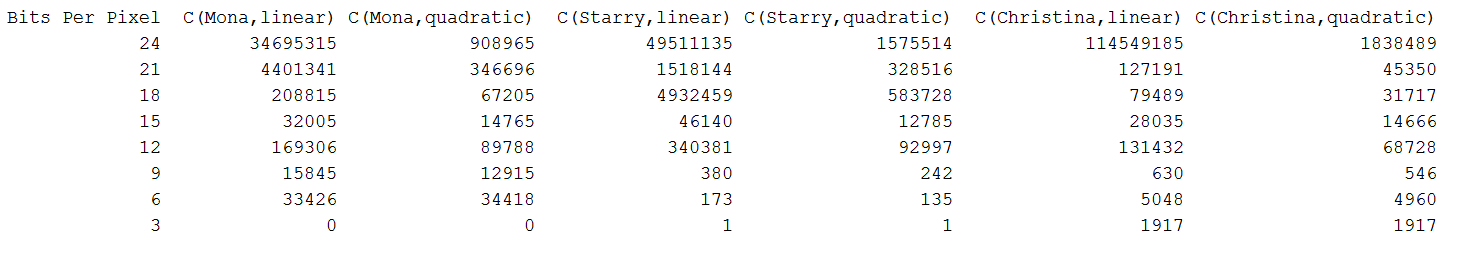
The ComparePaintings.java file implements a class whose instances handle analysis of one or more paintings. A basic constructor method is already provided. You should implement, for class ComparePaintings the following methods:

1. ColorHash countColors(String filename, int bitsPerPixel): Load the image, construct the hash table, count the colors.

2. double compare(ColorKeyHash painting1, ColorKeyHash painting2) Starting with two hash tables of color counts, compute a measure of similarity based on the cosine distance of two vectors.

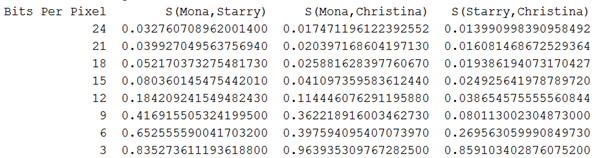
3. void basicTest(String filename): A basic test for the compare method: S(x,x) should be 1.0, so you should compute the similarity of an image with itself and print out the answer. If it comes out to be 1.0, that is a good sign for your implementation so far. For this method, you may use any initial tableSize you like, any value of rehashLoadFactor that works (you are welcome to experiment here), and either linear probing or quadratic probing.

4. void CollisionTests(): Using the three given painting images and a variety of bits-per-pixel values, compute and print out a table of collision counts in the following format(actual data):



In order to determine the number of collisions, your method countColors should keep a running total of the number of collisions as the pixel colors are counted. The number of collisions for each get and put operation are to be communicated from the ColorHash class back to the caller via the ResponseItem objects. For purposes of this assignment, we will define a collision to occur when either a put operation or a get operation arrives via the hash function or via collision resolution at a location that is occupied and that has the wrong key. (A put operation arriving at a location with the same key is an update operation and this is not a collision.) You may implement the logic for totaling up the numbers of collisions in either FeatureVector.java, ComparePaintings.java, or both. (late addition to the spec: For the collisionTests method, use an initial tableSize of 3 and a rehashLoadFactor value of 0.5.)

14. void fullSimilarityTests(): Using the three given painting images and a variety of bits-per-pixel values, compute and print out a table of similarity values in the following format(actual data):



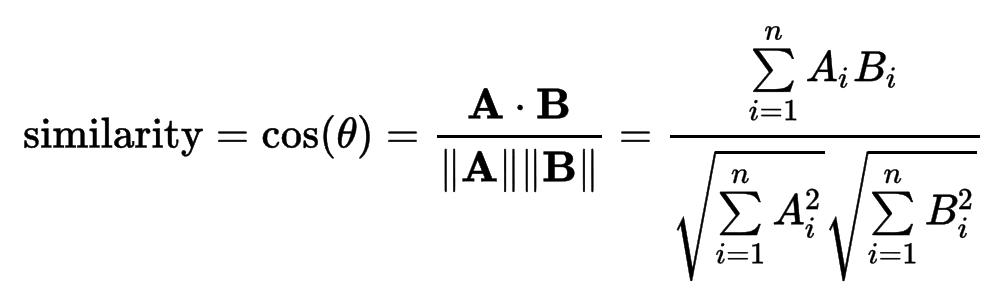
As bits per pixel go down, the S values for S(x,y) can be expected to rise towards 1.0.

**How to Compute a Cosine Similarity Value**

There are several different interpretations of "vector". For example, a vector is a sequence of numbers. An n-dimensional vector has n numbers. A vector can also be thought of as an object that has direction and magnitude. In physics, a vector may be used to represent velocity of motion; the direction represents which way the object in motion is going, and the magnitude tells how fast it is going.

A feature vector is a sequence of numbers that describes the features of some object. For us, the feature vector describes how much of each color is used in a particular painting. We want to measure the similarity of two paintings in terms of these feature vectors. The magnitude of a vector is the square root of the sum of the squares of the vector's elements. But the direction gives an indication of the relative values of the different elements -- relative to each other. Since two paintings could look very similar, but due to different sizes or digital resolution, could have vastly different counts of the pixels of each color, we want to avoid using magnitude of the vectors in our similarity comparison. Rather we will pay attention only to the relative directions. If the two vectors have the same direction, then the angle between them is zero, and we will call their similarity "high" and equal to 1.0. If the directions are as different as possible, the angle between them will be Pi over 2. The directions of our feature vectors cannot be negative due to the fact that all the counts of pixels are 0 or higher. So, we cannot have directional differences of pi radians. We will consider 0.0 to be the lowest similarity value, and it will occur when the directions of the vectors are perpendicular to each other (in whatever high-dimensional space we are using).

Let A and B be two n-dimensional vectors. Then the cosine similarity value between them is computed using the following formula.



The numerator here is called the dot product of A and B. The denominator here is a scalar product of two magnitude values: the magnitude of A times the magnitude of B. When you implement this, you may wish to implement separate "helper" methods for magnitude and for dot product. If you do this, you are free to name those and design them however you like.

**Assistance with Testing**

A handful of tests have been created for you in JUnit for ColorHash.java. You are encouraged to use those and add to them.