CS61B - HOMWORK REVIEW

1. Java Crash Course(HW 0)

In this assignment, we will go through basic Java syntax concepts.

For details please refer HW)

We will skip this section since the students who has previous programming background will feel quite easy for this section.

2. Packages, Interfaces, Generics, Exceptions, Iteration(HW 1)

The original questions and tasks could be found at the following HW1. In this homework, you will learn how to write and use packages, as well as get some hands-on practice with interfaces and abstract classes. We'll also get an opportunity to implement a simple data structure as well as an algorithm that's easy to implement using that data structure. Finally, we'll add support for iteration and exceptions (which we'll cover on Friday) to our data structure.

A. Task1: BoundedQueue

We will start by defining a BoundedQueue interface. The BoundedQueue is similar to our Deque from Project 1, but with a more limited API. Specifically, items can only be enqueued at the back of the queue, and can only be dequeued from the front of the queue. Unlike our Deque, the BoundedDeque has a fixed capacity, and nothing is allowed to enqueue if the queue is full.

My codes of the solution is listed below, please refer that:

```
package synthesizer;

import java.util.Iterator;

public interface BoundedQueue<T> extends Iterable<T> {

    int capacity();     // return size of the buffer

    int fillCount();    // return number of items currently in the buffer

    void enqueue(T x);  // add item x to the end

    T dequeue();        // delete and return item from the front

    T peek();           // return (but do not delete) item from the front

    default boolean isEmpty() { // is the buffer empty (fillCount equals zero)?
        return fillCount() == 0;
    }
```

```
    default boolean isFull() { // is the buffer full (fillCount is same as capacity)?
        return fillCount() == capacity();
    }

    Iterator<T> iterator();
}
```

Please be awared that we could not define any body of a method in the Interface excepts the method with default annotation. The `Iterator<T> iterator` method enables all implementations of `BoundedQueue` to be iterable, i.e. support enhanced `for` loop.

Nothing special in that part, just define the Interface by following the official instructions.

B. Task 2: AbstractBoundedQueue

Methods and classes can be declared as abstract using the abstract keyword. Abstract classes cannot be instantiated, but they can be subclassed using the `extends` keyword. Unlike interfaces, abstract classes can provide implementation inheritance for features other than public methods, including instance variables.

Classes that implement interfaces will inherit all of the methods and variables from that interface. If an implementing class fails to implement any abstract methods inherited from an interface, then that class must be declared abstract.

That is the next hierarchy of this homework:

```
package synthesizer;

public abstract class AbstractBoundedQueue<T> implements BoundedQueue<T> {
    protected int fillCount;
    protected int capacity;

    public int capacity() {
        return capacity;
    }

    public int fillCount() {
        return fillCount;
    }
}
```

This abstract class implements the `BoundedQueue<T>` interface, but does not implement all the methods. It also declares two `protected` variables.

C. Task 3: ArrayRingBuffer and Task 5: Iteration and Exceptions

The `ArrayRingBuffer` class will do all the real work by extending `AbstractBoundedQueue`. That means we can happily inherit `capacity()`, `fillCount()`, `isEmpty()`, and `isFull()` without having to override these, but we'll need to override all of the the abstract methods.

In the following pieces of codes, Kefan implements all the methods to support this data structure and make it available to iterate by adding `private class ArrayRingBufferIterator implements Iterator<T>`.

```
package synthesizer;

import java.util.Iterator;

public class ArrayRingBuffer<T> extends AbstractBoundedQueue<T> {
    /* Index for the next dequeue or peek. */
    private int first;            // index for the next dequeue or peek
    /* Index for the next enqueue. */
    private int last;
    /* Array for storing the buffer data. */
    private T[] rb;

    /**
     * Create a new ArrayRingBuffer with the given capacity.
     */
    public ArrayRingBuffer(int capacity) {
        this.first = 0;
        this.last = 0;
        this.fillCount = 0;
        this.capacity = capacity;
        rb = (T[]) new Object[capacity];
    }

    /**
     * Adds x to the end of the ring buffer. If there is no room, then
     * throw new RuntimeException("Ring buffer overflow"). Exceptions
     * covered Monday.
     */
    public void enqueue(T x) {
        if (fillCount == capacity) {
            throw new IllegalArgumentException("Ring Buffer Overflow");
        }
        rb[last] = x;
        last = (last + 1) % capacity;
        fillCount += 1;
    }

    /**
     * Dequeue oldest item in the ring buffer. If the buffer is empty, then
     * throw new RuntimeException("Ring buffer underflow"). Exceptions
```

```
     * covered Monday.
     */
    public T dequeue() {
        if (fillCount == 0) {
            throw new IllegalArgumentException("Ring Buffer Underflow");
        }
        T returnItem = rb[first];
        rb[first] = null;
        first = (first + 1) % capacity;
        fillCount -= 1;
        return returnItem;
    }

    /**
     * Return oldest item, but don't remove it.
     */
    public T peek() {
        if (fillCount == 0) {
            throw new IllegalArgumentException("Ring Buffer Underflow");
        }
        return rb[first];
    }

    public Iterator<T> iterator() {
        return new ArrayRingBufferIterator();
    }

    private class ArrayRingBufferIterator implements Iterator<T> {
        private int wizPos;
        private int num;

        public ArrayRingBufferIterator() {
            wizPos = first;
            num = 0;
        }

        @Override
        public boolean hasNext() {
            return num < fillCount;
        }

        @Override
        public T next() {
            T returnItem = rb[wizPos];
            wizPos = (wizPos + 1) % capacity;
            num += 1;
            return returnItem;
        }
    }
```

```
}
```

The first thing that is worth to mention is how to change the variables `first` and `last`:

```
last = (last + 1) % capacity;
```

```
first = (last + 1) % capacity;
```

This algorithm is elegant and tidy and it is similar as the algorithm that is implemented in **Proj1a ArrayDeque**. Please be noted that such algorithm is so important in the circular data structure.

The second thing that is worth to mention is how to implement iteration in our data structure.

I define `private class ArrayRingBufferIterator implements Iterator<T>` in which I override `hasNext()` and `next()` method.

Finally, I define `iterator()` which returns Iterator Object to make our data structure iterable.

```
public Iterator<T> iterator() {
        return new ArrayRingBufferIterator();
    }
```

D. Task 4: GuitarString

Finally, we want to flesh out `GuitarString`, which uses an `ArrayRingBuffer` to replicate the sound of a plucked string. We'll be using the Karplus-Strong algorithm, which is quite easy to implement with a BoundedQueue.

If you are wondering how the karptlus-Strong algorithm works, please refer GuitarString.

```
package synthesizer;

import java.util.HashSet;

//import javax.print.DocFlavor;

//Make sure this class is public
public class GuitarString {
    /**
     * Constants. Do not change. In case you're curious, the keyword final means
     * the values cannot be changed at runtime. We'll discuss this and other topics
     * in lecture on Friday.
     */
    private static final int SR = 44100;      // Sampling Rate
    private static final double DECAY = .996; // energy decay factor

    /* Buffer for storing sound data. */
    private BoundedQueue<Double> buffer;
```

```
    /* Create a guitar string of the given frequency.  */
    public GuitarString(double frequency) {
        buffer = new ArrayRingBuffer<>((int) Math.round(SR / frequency));
        for (int i = 0; i < buffer.capacity(); i++) {
            buffer.enqueue(.0);
        }
    }

    /* Pluck the guitar string by replacing the buffer with white noise. */
    public void pluck() {
        HashSet<Double> hs = new HashSet<>();
        for (int i = 0; i < buffer.capacity(); i++) {
            double r = Math.random() - 0.5;
            while (hs.contains(r)) {
                r = Math.random() - 0.5;
            }
            buffer.dequeue();
            buffer.enqueue(r);
            hs.add(r);
        }
    }

    /* Advance the simulation one time step by performing one iteration of
     * the Karplus-Strong algorithm.
     */
    public void tic() {
        double front = buffer.dequeue();
        double next = buffer.peek();
        double inserted = DECAY * 0.5 * (front + next);
        buffer.enqueue(inserted);
    }

    /* Return the double at the front of the buffer. */
    public double sample() {
        return buffer.peek();
    }
}
```

Nothing difficult, just following the official guide, but there is one point I would like to mention is using a HashSet to make sure there is no replicated value in the buffer.

E. When to Use Abstract Class and Interface

In practice, in can be a little unclear when to use an interface and when to use an abstract class. One mostly accurate metaphor that might help is that you can think of an interface as defining a “can-do” or an “is-a” relationship, whereas an abstract class should be a stricter “is-a” relationship. The difference can be subtle, and you can often use one instead of the other.

In practice, large Java libraries often have a hierarchy of interfaces, which are extended by abstract classes that provided default implementations for some methods, and which are in turn ultimately implemented by concrete classes. A good example is the Collection interface: It extends Iterable (which is its superinterface), and is implemented by many subinterfaces (i.e. List, Set, Map), which in turn have their own abstract implementations (AbstractList, AbstractSet AbstractMap). However, for smaller programs, the hierarchy is often stubbier, sometimes starting with an abstract class. For example, we could have just started with `AbstractBoundedQueue` at the top of the hierarchy and skipped having a `BoundedQueue` interface altogether.

3. Percolation(HW 2)

The original tasks and tips could be found at HW2. In this homework, you will write a program to estimate the value of the percolation threshold via Monte Carlo simulation. You will be using a disjoint sets implementation provided as part of the Princeton standard library.

> **Percolation.** Given a composite systems comprised of randomly distributed insulating and metallic materials: what fraction of the materials need to be metallic so that the composite system is an electrical conductor? Given a porous landscape with water on the surface (or oil below), under what conditions will the water be able to drain through to the bottom (or the oil to gush through to the surface)? Scientists have defined an abstract process known as percolation to model such situations.

In this task, the main job for us is to implement two classes, one is `Percolation.java` and another is `PercolationStats.java`.

A. Percolation

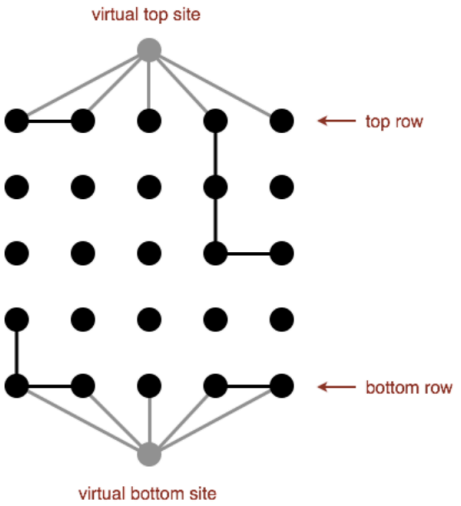
In `Percolation.java`, we transform the 2D matric to 1D array, as the instance variable `private boolean[] items` to store the open status for each point. The most important and tricky part I would like to mention here is the problem of **Virtual Sites and Backwash**.

Since we want to speed the process of `Percolates()` and `isFull(int row, int col)`, we will use the virtual site strategy. The following picture could demonstrate this strategy.

Speeding Things Up With Virtual Sites

One approach to making things fast is to create:

- Virtual top site connected to all open items in top row.
- Similar virtual site at bottom.



To check isFull:

- Check for a connection to top site.

To check percolates:

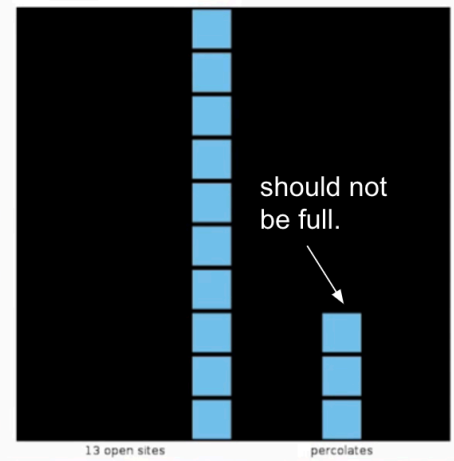
- Check for connection between top and bottom sites.

However, we will definitely encounter the BackWash problem that was caused by the virtual bottom site. In order to fix this dilemma, I will define two instance variables: `private WeightedQuickUnionUF itemsConnected;` and `private WeightedQuickUnionUF itemsConnectedBW;`. The first variable contains both bottom and top virtual sites but the second variable only contains the top virtual site since the bottom site may sometimes causes the BackWash problem.

Virtual Sites and Backwash

Potential downside of these virtual sites is backwash.

- Water should not flow back up through the virtual bottom site.
- Very tricky problem, feel free to discuss amongst yourselves.
  - Solution is only a small amount of code.
  - No spoilers in these slides.



The code for Percolation.java is:

```
public class Percolation {
```

```
    private int numberOfOpenSite;
    private int N;
    private boolean[] items;
    private WeightedQuickUnionUF itemsConnected;
    private WeightedQuickUnionUF itemsConnectedBW;

    // create N-by-N grid, with all sites initially blocked
    public Percolation(int N) {
        if (N <= 0) {
            throw new IllegalArgumentException("N should be larger than 0");
        }
        this.numberOfOpenSite = 0;
        this.N = N;
        this.items = new boolean[N * N];
        for (int i = 0; i < N * N - 1; i++) {
            items[i] = false;
        }
        itemsConnected = new WeightedQuickUnionUF((N * N) + 2);
        itemsConnectedBW = new WeightedQuickUnionUF((N * N) + 1);
    }

    private int xyTo1D(int row, int col) {
        return row * N + col;
    }

    // open the site (row, col) if it is not open already
    public void open(int row, int col) {
        if (row >= N || col >= N) {
            throw new IndexOutOfBoundsException();
        }
        int ith = xyTo1D(row, col);
        if (items[ith]) {
            return;
        }
        items[ith] = true;
        numberOfOpenSite += 1;
        if (row == 0) {
            itemsConnected.union(ith, N * N);
            itemsConnectedBW.union(ith, N * N);
        }
        if (row == N - 1) {
            itemsConnected.union(ith, N * N + 1);
        }
        if (ith % N != 0 && items[ith - 1]) {
            itemsConnected.union(ith, ith - 1);
            itemsConnectedBW.union(ith, ith - 1);
        }
        if ((ith + 1) % N != 0 && items[ith + 1]) {
            itemsConnected.union(ith, ith + 1);
```

```
            itemsConnectedBW.union(ith, ith + 1);
        }
        if ((ith - N) >= 0 && items[ith - N]) {
            itemsConnected.union(ith, ith - N);
            itemsConnectedBW.union(ith, ith - N);
        }
        if ((ith + N) < (N * N) && items[ith + N]) {
            itemsConnected.union(ith, ith + N);
            itemsConnectedBW.union(ith, ith + N);
        }
    }

    // is the site (row, col) open?
    public boolean isOpen(int row, int col) {
        if (row >= N || col >= N) {
            throw new IndexOutOfBoundsException();
        }
        int ith = xyTo1D(row, col);
        return items[ith];
    }

    // is the site (row, col) full? To check whether this site
    // is connected to the top
    public boolean isFull(int row, int col) {
        if (row >= N || col >= N) {
            throw new IndexOutOfBoundsException();
        }
        int ith = xyTo1D(row, col);
        if (!isOpen(row, col)) {
            return false;
        }
        return itemsConnectedBW.connected(ith, N * N);
    }

    // number of open sites
    public int numberOfOpenSites() {
        return numberOfOpenSite;
    }

    // does the system percolate?
    public boolean percolates() {
        if (N == 1 && !items[0]) {
            return false;
        }
        return itemsConnected.connected(N * N, N * N + 1);
    }

    public static void main(String[] args) {
```

```
    }
}
```

B. PercolationStats

Monte Carlo simulation. To estimate the percolation threshold, consider the following computational experiment:

- Initialize all sites to be blocked.
- Repeat the following until the system percolates:
    - Choose a site uniformly at random among all blocked sites.
    - Open the site.
    - The fraction of sites that are opened when the system percolates provides an estimate of the percolation threshold.

For example, if sites are opened in a 20-by-20 grid according to the snapshots below, then our estimate of the percolation threshold is 204/400 = 0.51 because the system percolates when the 204th site is opened.

By repeating this computation experiment T times and averaging the results, we obtain a more accurate estimate of the percolation threshold. Let xtxt be the fraction of open sites in computational experiment tt. The sample mean μprovides an estimate of the percolation threshold; the sample standard deviation σ measures the sharpness of the threshold.

The implementation of PercolationStats is:

```
public class PercolationStats {
    private int N;
    private int T;
    private double[] items;

    // perform T independent experiments on an N-by-N grid
    public PercolationStats(int N, int T, PercolationFactory pf) {
        if (N <= 0 || T <= 0) {
            throw new IllegalArgumentException("N or T should be larger than 0");
        }
        this.N = N;
        this.T = T;
        items = new double[T];
        for (int i = 0; i < T; i++) {
            Percolation p = pf.make(N);
            while (!p.percolates()) {
                int row = StdRandom.uniform(N);
                int col = StdRandom.uniform(N);
                p.open(row, col);
            }
            items[i] = p.numberOfOpenSites() * 1.0 / (N * N);
        }
    }
```

```
    // sample mean of percolation threshold
    public double mean() {
        return StdStats.mean(items);
    }

    // sample standard deviation of percolation threshold
    public double stddev() {
        return StdStats.stddev(items);
    }

    // low endpoint of 95% confidence interval
    public double confidenceLow() {
        return mean() - (1.96 * stddev()) / Math.sqrt(T);
    }

    // high endpoint of 95% confidence interval
    public double confidenceHigh() {
        return mean() + (1.96 * stddev()) / Math.sqrt(T);
    }
}
```

Nothing tricky, just following the official guide.

4. Hashing(HW 3)

The original tasks and tips could be found at HW 3. In this lightweight HW, we'll work to better our understanding of hash tables. We will self design a `hashCode()` function and `equals(Object o)` in `SimpleOomages` class and find flaws for hashCode() in `ComplexOomages` class.

A. Simple Oomage

our goal in this part of the assignment will be to write an `equals` and `hashCode` method for the `SimpleOomage` class, as well as tests for the `hashCode` method in the `TestSimpleOomage` class.

a. Equals()

Start by running `TestSimpleOomage`. You'll see that you fail the `testEquals` test. The problem is that two `SimpleOomage` objects are not considered equal, even if they have the same `red`, `green`, and `blue` values. This is because `SimpleOomage` is using the default `equals` method, which simply checks to see if the the `ooA` and `ooA2` references point to the same memory location.

Your first task for this homework is to write an `equals` method.

According to the Java language specification, your `equals` method should have the following properties to be in compliance:

- Reflexive: `x.equals(x)` must be true for any non-null `x`.
- Symmetric: `x.equals(y)` must be the same as `y.equals(x)` for any non-null `x` and `y`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)` for any non-null `x`, `y`, and `z`.
- Consistent: `x.equals(y)` must return the same result if called multiple times, so long as the object

referenced by `x` and `y` do not change.
- Not-equal-to-null: `x.equals(null)` should be false for any non-null `x`.

The key philosophy for `equals()` method is to check whether these two object have the same red, green and blue property. The code for `equals()` method is:

```
public class SimpleOomage implements Oomage {
    protected int red;
    protected int green;
    protected int blue;

    private static final double WIDTH = 0.01;
    private static final boolean USE_PERFECT_HASH = true;

    @Override
    public boolean equals(Object o) {
        // TODO: Write this method.
        if (o == this) {
            return true;
        }
        if (o == null || o.getClass() != this.getClass()) {
            return false;
        }
        SimpleOomage comparedItem = (SimpleOomage) o;
        if (this.red == comparedItem.red && this.green == comparedItem.green
                && this.blue == comparedItem.blue) {
            return true;
        } else {
            return false;
        }
    }
```

And the test for this method is:

```
@Test
    public void testEquals() {
        SimpleOomage ooA = new SimpleOomage(5, 10, 20);
        SimpleOomage ooA2 = new SimpleOomage(5, 10, 20);
        SimpleOomage ooB = new SimpleOomage(50, 50, 50);
        assertEquals(ooA, ooA2);
        assertNotEquals(ooA, ooB);
        assertNotEquals(ooA2, ooB);
        assertNotEquals(ooA, "ketchup");
    }
```

b. A Perfect hashCode

In Java, it is critically important that if you override `equals` that you also override `hashCode`.

The Java specification for `equals` mentions this danger as well: "Note that it is generally necessary to override the `hashCode` method whenever the `equals` method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes."

Our next goal for the `SimpleOomage` class will be to write a *perfect* `hashCode` function. By perfect, we mean that two `SimpleOomage`s may only have the same hashCode only if they have the exact same red, green, and blue values.

But before we write it, fill in the `testHashCodePerfect` of `TestSimpleOomage` with code that tests to see if the `hashCode` function is perfect. Hint: Try out every possible combination of red, green, and blue values and ensure that you never see the same value more than once. Feel free to use any data structure you want.

The key philosophy of writing a `hashCode()` method is calculating the possible values for each red, green and blue, and that is 52. So we will use 52 作为底数，总共有三个实例变量，因此红色对应52^2, 绿色对应52^1, 蓝色对应52^0，再分别乘上各自的值除以5(因为都是5的倍数，所以要消除5的影响)。This idea comes from HugBook.

The code for `hashCode()` function is:

```
@Override
    public int hashCode() {
        if (!USE_PERFECT_HASH) {
            return red + green + blue;
        } else {
            // TODO: Write a perfect hash function for Simple Oomages.
            int redCode = (this.red / 5) * (52 * 52);
            int greenCode = (this.green / 5) * 52;
            int blueCode = this.blue / 5;
            return redCode + greenCode + blueCode;
        }
    }
```

The test for that is:

```
 @Test
    public void testHashCodePerfect() {
        /* TODO: Write a test that ensures the hashCode is perfect,
          meaning no two SimpleOomages should EVER have the same
          hashCode UNLESS they have the same red, blue, and green values!
         */
        Set<SimpleOomage> setSimpleOomage = new HashSet<>();
        for (int red = 0; red <= 255; red += 5) {
            for (int green = 0; green <= 255; green += 5) {
                for (int blue = 0; blue <= 255; blue += 5) {
                    setSimpleOomage.add(new SimpleOomage(red, green, blue));
```

```
                }
            }
        }
        Set<Integer> setHashCode = new HashSet<>();
        for (SimpleOomage s : setSimpleOomage) {
            assertTrue(setHashCode.add(s.hashCode()));
        }
    }
```

c. Evaluating the Spread of Perfect hashCode

We define a nice spread as:

- No bucket has fewer than N / 50 oomages.
- No bucket has more than N / 2.5 oomages. , where N is the number of oomages.

In other words, the number of oomages per bucket has to be within the range (N / 50, N / 2.5).

For the purposes of converting an Oomage's hashCode to a bucket number, you should compute `bucketNum = (o.hashCode() & 0x7FFFFFFF) % M`, where `o` is an Oomage. You should not use `Math.abs` or `Math.floorMod`.

The test for evaluating the spread of perfect hashCode is:

```
public class OomageTestUtility {
    public static boolean haveNiceHashCodeSpread(List<Oomage> oomages, int M) {
        /* TODO:
         * Write a utility function that returns true if the given oomages
         * have hashCodes that would distribute them fairly evenly across
         * M buckets. To do this, convert each oomage's hashcode in the
         * same way as in the visualizer, i.e. (& 0x7FFFFFFF) % M.
         * and ensure that no bucket has fewer than N / 50
         * Oomages and no bucket has more than N / 2.5 Oomages.
         */
        HashMap<Integer, Integer> map = new HashMap<>();
        for (Oomage o : oomages) {
            int bucketNum = (o.hashCode() & 0x7FFFFFFF) % M;
            if (map.containsKey(bucketNum)) {
                map.put(bucketNum, map.get(bucketNum) + 1);
            } else {
                map.put(bucketNum, 1);
            }
        }
        Set<Integer> set = map.keySet();
        int size = oomages.size();
        for (int i : set) {
            int numOfEachBucket = map.get(i);
            if (numOfEachBucket > (size / 50) && numOfEachBucket < (size / 2.5)) {
                continue;
            } else {
```

```
                return false;
            }
        }
        return true;
    }
```

```
@Test
    public void testRandomOomagesHashCodeSpread() {
        List<Oomage> oomages = new ArrayList<>();
        int N = 10000;

        for (int i = 0; i < N; i += 1) {
            oomages.add(NiceSpreadOomage.randomNiceSpreadOomage());
        }

        assertTrue(OomageTestUtility.haveNiceHashCodeSpread(oomages, 10));
    }

@Test
    public void testRandomOomagesHashCodeSpread() {
        List<Oomage> oomages = new ArrayList<>();
        int N = 10000;

        for (int i = 0; i < N; i += 1) {
            oomages.add(SimpleOomage.randomSimpleOomage());
        }

        assertTrue(OomageTestUtility.haveNiceHashCodeSpread(oomages, 10));
    }
```

B. Complex Oomage

The `ComplexOomage` class is a more sophisticated beast. Instead of three instance variables representing `red`, `green`, and `blue` values, each `ComplexOomage` has an entire a list of ints between 0 and 255 (not necessarily multiples of 5). This list may be of any length.

This time, you won't change the `ComplexOomage` class at all. Instead, your job will be to write tests to find the flaw in the `hashCode` function.

The original `hashCode()` method in `ComplexOomage` class is:

```
public class ComplexOomage implements Oomage {
    protected List<Integer> params;
    private static final double WIDTH = 0.05;

    @Override
    public int hashCode() {
        int total = 0;
```

```
        for (int x : params) {
            total = total * 256;
            total = total + x;
        }
        return total;
    }

    @Override
    public boolean equals(Object o) {
        if (o.getClass() != this.getClass()) {
            return false;
        }
        ComplexOomage otherComplexOomage = (ComplexOomage) o;
        return params.equals(otherComplexOomage.params);
    }
}
```

In this case, we need to look at the `hashCode` to find the problem. Fill in the test `testWithDeadlyParams` such that the provided `hashCode` function fails due to poor distribution of `ComplexOomage` objects.

Given what we've learned in 61B so far, this is a really tricky problem! Consider how Java represents integers in binary (see lecture 23 for a review). For a hint, see Hint.java.

```
public class Hint {
    public static void main(String[] args) {
        System.out.println("The powers of 256 in Java are:");
        int x = 1;
        for (int i = 0; i < 10; i += 1) {
            System.out.println(i + "th power: " + x);
            x = x * 256;
        }
    }
}
```

```
The powers of 256 in Java are:
0th power: 1
1th power: 256
2th power: 65536
3th power: 16777216
4th power: 0
5th power: 0
6th power: 0
7th power: 0
8th power: 0
9th power: 0
```

The Hint.java tells us the power of 256 will be beyond the largest number that Java could represent, which will cause the poorly distribution of bucket number.

So we write the test which will be failed due to that.

```
@Test
    public void testWithDeadlyParams() {
        List<Oomage> deadlyList = new ArrayList<>();
        // Your code here.
        for (int i = 3; i < 100; i++) {
            List<Integer> parameter = new ArrayList<>();
            for (int j = 0; j < i; j++) {
                parameter.add(255);
            }
            deadlyList.add(new ComplexOomage(parameter));
        }
        assertTrue(OomageTestUtility.haveNiceHashCodeSpread(deadlyList, 10));
    }
```

5. 8 Puzzle(HW 4)

The original questions and tips could be found at HW4.

A. Puzzles Formation and Introduction

In this assignment, we'll be building an artificial intelligence that solves puzzles. Specifically, given an object of type `WorldState`, your solver will take that `WorldState` and find a sequence of valid transitions between world states such that the puzzle is solved.

As an example of one such puzzle, consider the "word ladder" puzzle. In this puzzle, we try to convert one word in English to another by either changing, adding, or removing letters such that every transition results in a valid English word. Suppose we start with the word "horse" and we want to turn it into "nurse". To do this, we could perform the following transitions: horse -> hose -> hole -> cole -> core -> cure -> pure -> purse -> nurse.

As another example, consider the 8-puzzle problem. The 8-puzzle is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square tiles labeled 1 through 8 and a blank square. In this puzzle, the goal is to rearrange the tiles so that they are in order, using as few moves as possible. The player is permitted to slide tiles horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board (left) to the goal board (right). Each of these puzzles is considered a valid "WorldState".

```
   1  3        1     3        1  2  3        1  2  3        1  2  3
4  2  5   =>   4  2  5   =>   4     5   =>   4  5     =>   4  5  6
7  8  6        7  8  6        7  8  6        7  8  6        7  8

initial        1 left          2 up          5 left          goal
```

In this assignment, not only must your artificial intelligence find a solution to **any WorldState puzzle**, but it must also be the shortest possible solution. While this may seem daunting, read on, and you'll see that we can solve this problem with a clever use of a priority queue.

B. Best-First Search

Your AI will implement the A* search algorithm.

Before we describe this algorithm, we'll first define a **search node** of the puzzle to be:

- a `WorldState`.
- the number of moves made to reach this world state from the initial state.
- a reference to the previous search node.

The first step of the algorithm is to create a **priority queue** of search nodes, and insert an "initial search node" into the priority queue, which consists of:

- the initial world state.
- 0 (since no moves have been made yet).
- null (since there is no previous search node).

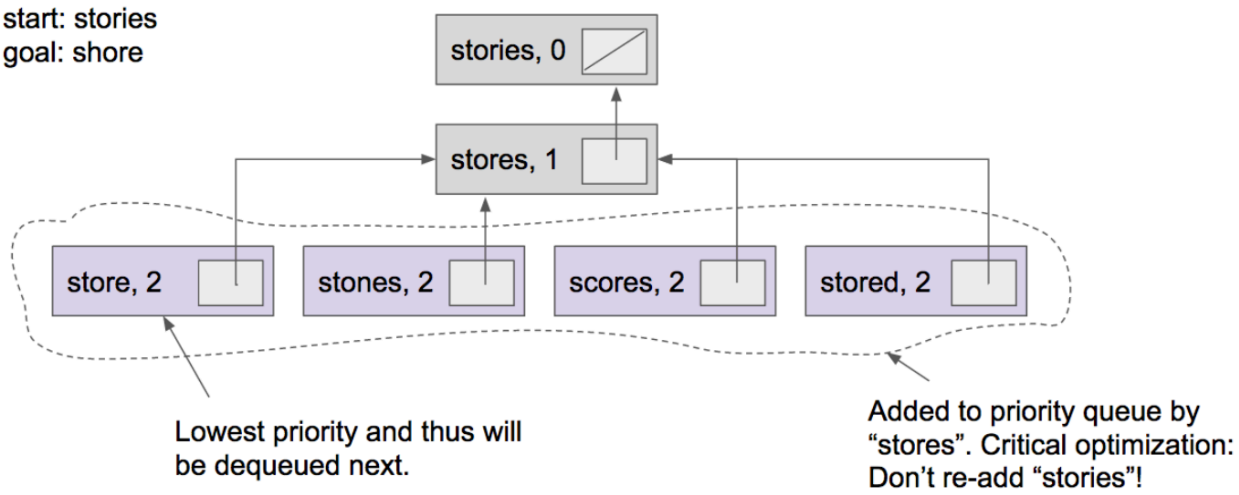
The algorithm then proceeds as follows:

- Remove the search node with minimum priority. Let's call this node X. If it is the goal node, then we're done.
- Otherwise, for each neighbor of X's world state, create a new search node that obeys the description above and insert it into the priority queue.

We can think of each search node as having a priority equal to the sum of (the number of moves made to reach this world state from the initial state + the `WorldState`'s `estimatedDistanceToGoal`). In other words, if we have two search nodes on the priority queue, with the first M1 moves away from the initial state and E1 as its estimated distance to goal, and the other having M2 moves so far and E2 as its estimated distance to goal, then the former search node will have a lower priority if (M1 + E1) < (E2 + M2).

The `A* algorithm` can also be thought of as "Given a state, pick a neighbor state such that (distance so far + estimated distance to goal) is minimized. Repeat until the goal is seen".

As an example, consider the problem of converting the word “stories” into “shore”. The diagram below shows the six search-nodes created after two removals, i.e. after “stories” has had a chance to be X, and “stores” has had a chance to be X. At this point in time, the priority queue contains four search nodes, and the next node to be dequeued will be “store”, for which M = 2 and E = 1. The critical optimization mentioned in the figure is described below under “Optimizations”.



To see an example of this algorithm in action, see this video or these slides.

C. Solver

Now we will start to implement `Solver` class.

```
import java.util.ArrayDeque;

public class Solver {

    /* Store the SeachNode */
    private MinPQ<SearchNode> pq;

    /* Store each state of WorldState to final goal */
    private ArrayDeque<WorldState> solution;

    /* Store the last SearchNode to the final goal */
    private SearchNode finishNode;

    private class SearchNode implements Comparable<SearchNode> {

        /* Previous SearchNode */
        private SearchNode pre;

        /* The number of steps from initial state */
        private int moves;
```

```
    /* The number of steps to final state obtained from word */
    private int estimate;

    /* Store the word object */
    private WorldState word;

    public SearchNode(SearchNode pre, int moves, WorldState word) {
        this.pre = pre;
        this.moves = moves;
        this.estimate = word.estimatedDistanceToGoal();
        this.word = word;
    }

    @Override
    public int compareTo(SearchNode s) {
        return (this.moves + this.estimate)
                - (s.moves + s.estimate);
    }
}

public Solver(WorldState initial) {
    pq = new MinPQ<>();
    SearchNode firstNode = new SearchNode(null, 0, initial);
    pq.insert(firstNode);
    while (!pq.isEmpty()) {
        SearchNode x = pq.delMin();
        if (x.estimate == 0) {
            finishNode = x;
            return;
        }
        Iterable<WorldState> neighbors = x.word.neighbors();
        for (WorldState n : neighbors) {
            if (x.pre == null || !n.equals(x.pre.word)) {
                pq.insert(new SearchNode(x, x.moves + 1, n));
            }
        }
    }
}

public int moves() {
    return finishNode.moves;
}

public Iterable<WorldState> solution() {
    solution = new ArrayDeque<>();
    SearchNode tmp = finishNode;
    while (tmp != null) {
        solution.addFirst(tmp.word);
```

```
            tmp = tmp.pre;
        }
        return solution;
    }
}
```

We need first to define a private nested class called `SearchNode` which implements `Comparable<SearchNode>` interface. Please be noted that we must override the `compareTo(SearchNode s)` otherwise the MinPQ will not work fine.

The constructor `public Solver(WorldState initial)` will solve the puzzle, computing everything necessary for `moves()` and `solution()`. Just following the instructions above to use the A* algorithm to implement the constructor.

We just use `finishNode` to get the number of steps and the `prev` to get an iterable solution from initial state to final state.

D. Board

For the second part of this assignment, you'll implement the Board class, allowing your Solver from part 1 to solve the 8-puzzle.

Unlike `word` class, there are two goal distance estimates ways:

- *Hamming estimate*: The number of tiles in the wrong position.
- *Manhattan estimate*: The sum of the Manhattan distances (sum of the vertical and horizontal distance) from the tiles to their goal positions.

For example, the Hamming and Manhattan estimates of the board below are 5 and 10, respectively.

```
8  1  3        1  2  3     1  2  3  4  5  6  7  8    1  2  3  4  5  6  7  8
4     2        4  5  6     ----------------------    ----------------------
7  6  5        7  8        1  1  0  0  1  1  0  1    1  2  0  0  2  2  0  3

initial          goal         Hamming = 5 + 0          Manhattan = 10 + 0
```

The implementation of `Board` class is:

```
import edu.princeton.cs.algs4.Queue;

public class Board implements WorldState {

    private static final int BLANK = 0;
    private final int[][] tiles;
    private int size;

    /**
     * Constructor of Board using tiles
     */
    public Board(int[][] tiles) {
```

```
        this.size = tiles.length;
        this.tiles = new int[size][size];
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                this.tiles[i][j] = tiles[i][j];
            }
        }
    }

    /**
     * Returns value of tile at row i, column j (or 0 if blank)
     */
    public int tileAt(int i, int j) {
        if (i < 0 || i > size - 1) {
            throw new IndexOutOfBoundsException();
        }
        if (j < 0 || j > size - 1) {
            throw new IndexOutOfBoundsException();
        }
        return tiles[i][j];
    }

    /**
     * Returns neighbors of this board.
     * SPOILERZ: This is the answer.
     * Copy form Hug's solution
     */
    @Override
    public Iterable<WorldState> neighbors() {
        Queue<WorldState> neighbors = new Queue<>();
        int hug = size();
        int bug = -1;
        int zug = -1;
        for (int rug = 0; rug < hug; rug++) {
            for (int tug = 0; tug < hug; tug++) {
                if (tileAt(rug, tug) == BLANK) {
                    bug = rug;
                    zug = tug;
                }
            }
        }
        int[][] ili1li1 = new int[hug][hug];
        for (int pug = 0; pug < hug; pug++) {
            for (int yug = 0; yug < hug; yug++) {
                ili1li1[pug][yug] = tileAt(pug, yug);
            }
        }
        for (int l11il = 0; l11il < hug; l11il++) {
            for (int lil1il1 = 0; lil1il1 < hug; lil1il1++) {
```

```
            if (Math.abs(-bug + l11il) + Math.abs(lil1il1 - zug) - 1 == 0) {
                ili1li1[bug][zug] = ili1li1[l11il][lil1il1];
                ili1li1[l11il][lil1il1] = BLANK;
                Board neighbor = new Board(ili1li1);
                neighbors.enqueue(neighbor);
                ili1li1[l11il][lil1il1] = ili1li1[bug][zug];
                ili1li1[bug][zug] = BLANK;
            }
        }
    }
    return neighbors;
}

/**
 * Returns the board size N
 */
public int size() {
    return size;
}

/**
 * Hamming estimate
 */
public int hamming() {
    int hamDistance = 0;
    int goal = 1;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i == size - 1 && j == size - 1) {
                break;
            }
            if (tiles[i][j] != goal) {
                hamDistance += 1;
            }
            goal += 1;
        }
    }
    return hamDistance;
}

/**
 * Private helper method to locate tile
 */
private int[] locate(int target) {
    int[] outcome = new int[2];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (tiles[i][j] == target) {
                outcome[0] = i;
```

```
                outcome[1] = j;
                break;
            }
        }
    }
    return outcome;
}

/**
 * Manhattan estimate
 */
public int manhattan() {
    int manDistance = 0;
    int goal = 1;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i == size - 1 && j == size - 1) {
                break;
            }
            int[] location = locate(goal);
            int xDistance = location[0] - i;
            if (xDistance < 0) {
                xDistance = -xDistance;
            }
            int yDistance = location[1] - j;
            if (yDistance < 0) {
                yDistance = -yDistance;
            }
            manDistance += xDistance;
            manDistance += yDistance;
            goal += 1;
        }
    }
    return manDistance;
}

@Override
public int estimatedDistanceToGoal() {
    return this.manhattan();
}

/**
 * Returns the string representation of the board.
 * Uncomment this method.
 */
public String toString() {
    StringBuilder s = new StringBuilder();
    int N = size();
    s.append(N + "\n");
```

```
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                s.append(String.format("%2d ", tileAt(i, j)));
            }
            s.append("\n");
        }
        s.append("\n");
        return s.toString();
    }

    @Override
    public boolean equals(Object O) {
        if (this == O) {
            return true;
        }
        if (O == null || O.getClass() != this.getClass()) {
            return false;
        }
        Board B = (Board) O;
        if (this.size != B.size) {
            return false;
        }
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (tiles[i][j] != B.tileAt(i, j)) {
                    return false;
                }
            }
        }
        return true;
    }

    @Override
    public int hashCode() {
        int code = 0;
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                code += tiles[i][j] * 31;
            }
        }
        return code;
    }
}
```

The implementation process is quite straightforward, in order to make our code more compact, I added a private helper method `locate()` to get the location of a target value, which is helpful when we implement `public int manhattan()`.

In addition, when you override `equals`, don't forget to define a new `hashCode()` method.

In the end, we could edit configurations in Run toor bar in Intellij IEDA to set the command line arguments.

6. Seam Carving(HW 5)

The original instructions and tips could be found at HW5.

> Seam-carving is a content-aware image resizing technique where the image is reduced in size by one pixel of height (or width) at a time. A vertical seam in an image is a path of pixels connected from the top to the bottom with one pixel in each row. (A horizontal seam is a path of pixels connected from the left to the right with one pixel in each column.) Below is the original 505-by-287 pixel image; further below we see the result after removing 150 vertical seams, resulting in a 30% narrower image. Unlike standard content-agnostic resizing techniques (e.g. cropping and scaling), the most interesting features (aspect ratio, set of objects present, etc.) of the image are preserved.

In this assignment, you will create a data type that resizes a W-by-H image using the seam-carving technique.

A. Energy Calculation

The first step is to determine which pixel in the picture shoud be removed. Our strategy is to calculate the energy of each pixel which is determined by energy function in the picture. The defination of energy function is that the square of the absolute value in differences of red, blue and green components in both X and Y direction.

The implementation of Energy Calculation is:

```
// energy of pixel at column x and row y
    public double energy(int x, int y) {

        if (x >= width || y >= height || x < 0 || y < 0) {
            throw new IndexOutOfBoundsException("Please enter a valid x or y");
        }

        int leftX = x - 1 < 0 ? width - 1 : x - 1;
        int rightX = x + 1 >= width ? 0 : x + 1;
        double energyX = calcEnergy(picture.get(leftX, y), picture.get(rightX, y));

        int upY = y - 1 < 0 ? height - 1 : y - 1;
        int downY = y + 1 >= height ? 0 : y + 1;
        double energyY = calcEnergy(picture.get(x, upY), picture.get(x, downY));
        return energyX + energyY;
    }

    private double calcEnergy(Color c1, Color c2) {
        int red = c1.getRed() - c2.getRed();
        int blue = c1.getBlue() - c2.getBlue();
        int green = c1.getGreen() - c2.getGreen();
        return (double) red * red + blue * blue + green * green;
    }
```

Nothing tricky, but please be noted that the following statements are quite useful when we are handling multiply cases such as whether `x` or `y` is located in the corner of picture.

```
int leftX = x - 1 < 0 ? width - 1 : x - 1;
int rightX = x + 1 >= width ? 0 : x + 1;
```

B. findVerticalSeam(): Finding a Minimum Energy Path

The `findVerticalSeam()` method should return an array of length H such that entry x is the column number of the pixel to be removed from row x of the image.

For example, consider the 6-by-5 image below (supplied as 6x5.png).

(78,209, 79)	**(63,118,247)**	**(92,175, 95)**	**(243, 73,183)**	**(210,109,104)**	**(252,101,119)**
(224,191,182)	(108, 89, 82)	(80,196,230)	(112,156,180)	(176,178,120)	(142,151,142)
(117,189,149)	(171,231,153)	(149,164,168)	(107,119, 71)	(120,105,138)	(163,174,196)
(163,222,132)	(187,117,183)	(92,145, 69)	(158,143, 79)	(220, 75,222)	(189, 73,214)
(211,120,173)	(188,218,244)	(214,103, 68)	(163,166,246)	(79,125,246)	(211,201, 98)

The corresponding pixel energies are shown below, with a minimum energy vertical seam highlighted in pink. In this case, the method `findVerticalSeam()` returns the array `{ 3, 4, 3, 2, 2 }`.

57685.0	**50893.0**	**91370.0**	**25418.0**	**33055.0**	**37246.0**
15421.0	56334.0	22808.0	54796.0	**11641.0**	25496.0
12344.0	19236.0	52030.0	**17708.0**	44735.0	20663.0
17074.0	23678.0	**30279.0**	80663.0	37831.0	45595.0
32337.0	30796.0	**4909.0**	73334.0	40613.0	36556.0

When there are multiple vertical seams with minimal total energy, your method can return any such seam.

Your `findVerticalSeam` method should work by first solving a base case subproblem, and then using the results of previous subproblems to solve later subproblems. Suppose we have the following definitions:

- M(i,j)M(i,j) - cost of minimum cost path ending at (i, j)
- e(i,j)e(i,j) - energy cost of pixel at location (i, j)

Then each subproblem is the calculation of M(i,j)for some i and j. The top row is trivial, M(i,0) is just e(i,0) for all i. For lower rows, we can find M(i,j)simply by adding the e(i,j) to the minimum cost path ending at its top left, top middle, and top right pixels, or more formally: M(i,j)=e(i,j)+min(M(i−1,j−1),M(i,j−1),M(i+1,j−1)).

Addendum: The Java language does not deal well with deep recursion, and thus a recursive approach will almost certainly not be able to handle images of largish size (say 500x500). We recommend writing your code iteratively.

My implementation is to define a instance variable `minCost` as a 2D matric to store the minCost of each pixel in the picture.

```
private double getMinUpper(double x, double y, double z) {
        double min1 = Math.min(x, y);
        return Math.min(min1, z);
    }

    private void setMinCost() {
        minCost = new double[width][height];
        double minLeft, minMiddle, minRight;
        for (int i = 0; i < width; i++) {
            minCost[i][0] = energy[i][0];
        }
        if (width == 1) {
            return;
```

```
        }
        for (int i = 1; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (j == 0) {
                    minMiddle = minCost[j][i - 1];
                    minRight = minCost[j + 1][i - 1];
                    if (minMiddle <= minRight) {
                        minCost[j][i] = energy[j][i] + minMiddle;
                    } else {
                        minCost[j][i] = energy[j][i] + minRight;
                    }
                } else if (j == width - 1) {
                    minLeft = minCost[j - 1][i - 1];
                    minMiddle = minCost[j][i - 1];
                    if (minMiddle <= minLeft) {
                        minCost[j][i] = energy[j][i] + minMiddle;
                    } else {
                        minCost[j][i] = energy[j][i] + minLeft;
                    }
                } else {
                    minLeft = minCost[j - 1][i - 1];
                    minMiddle = minCost[j][i - 1];
                    minRight = minCost[j + 1][i - 1];
                    minCost[j][i] = energy[j][i] + getMinUpper(minLeft,
minMiddle,minRight);
                }
            }
        }
    }
```

After we set up the instance variable `minCost`, we will start to find the minimum energy path in `minCost`. I will define a variable row and minIndex to track the trace of minimum energy path. We start at the bottom of `minCost` since we have already knew the which bottom point has the minimum energy.

```
// sequence of indices for vertical seam
    public int[] findVerticalSeam() {
        int[] verticalSeam = new int[height];
        int minIndex = 0;
        double minCostBottom = Double.MAX_VALUE;
        for (int i = 0; i < width; i++) {
            if (minCost[i][height - 1] < minCostBottom) {
                minCostBottom = minCost[i][height - 1];
                minIndex = i;
            }
        }
        int row = height - 1;
        verticalSeam[row] = minIndex;
        while (row >= 0) {
            row -= 1;
```

```
            if (row < 0) {
                break;
            }
            // int minLeftIndex, minMiddleIndex, minRightIndex;
            int minLeftIndex = minIndex - 1 < 0 ? 0 : minIndex - 1;
            int minMiddleIndex = minIndex;
            int minRightIndex = minIndex + 1 >= width ? width - 1 : minIndex + 1;
            if (minCost[minLeftIndex][row] <= minCost[minMiddleIndex][row]
                    && minCost[minLeftIndex][row] <= minCost[minRightIndex][row]) {
                verticalSeam[row] = minLeftIndex;
                minIndex = minLeftIndex;
            } else if (minCost[minMiddleIndex][row] <= minCost[minLeftIndex][row]
                    && minCost[minMiddleIndex][row] <= minCost[minRightIndex][row]) {
                verticalSeam[row] = minMiddleIndex;
                minIndex = minMiddleIndex;
            } else if (minCost[minRightIndex][row] <= minCost[minLeftIndex][row]
                    && minCost[minRightIndex][row] <= minCost[minMiddleIndex][row]) {
                verticalSeam[row] = minRightIndex;
                minIndex = minRightIndex;
            }
        }
        return verticalSeam;
    }
```

C. findHorizontalSeam(): Avoiding Redundancy

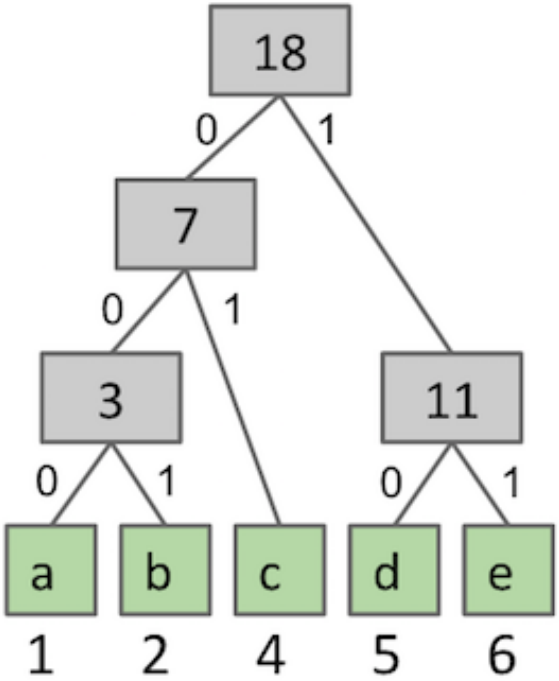
> The behavior of `findHorizontalSeam()` as analogous to that of `findVerticalSeam()` except that it should return an array of W such that entry y is the row number of the pixel to be removed from column y of the image. Your `findHorizontalSeam` method should NOT be a copy and paste of your `findVerticalSeam` method! Instead, considering transposing the image, running `findVerticalSeam`, and then transposing it back.

The ideas behind `findHorizontalSeam()` is quite straightforward, we just reverse hte origin pucture and use `findVerticalSeam()` to that picture.

```
// sequence of indices for horizontal seam
   public int[] findHorizontalSeam() {
        Picture pictureReverse = new Picture(height, width);
        for (int col = 0; col < width; col++) {
            for (int row = 0; row < height; row++) {
                Color pixel = picture.get(col, row);
                pictureReverse.set(row, col, pixel);
            }
        }
        SeamCarver sc = new SeamCarver(pictureReverse);
        int[] horizontalSeam = sc.findVerticalSeam();
        return horizontalSeam;
    }
```

7. Huffman Coding(HW 7)

> The original tips and requirements could be found at HW 7.
>
> In this homework, you'll implement a Huffman encoder and decoder, as described in lecture 38.
>
> The majority of the work will be in building the Huffman decoding trie. For the purposes of this homework, the "less frequent" branch of your Huffman coding trie should always be the '0' side, and the more common side should always be the '1' side.
>
> For example, suppose we have the file below:
>
> ```
> abbccccddddeeeeee
> ```
>
> This file has 1 a, 2 b's, 4 c's, 5 d's, and 6 e's. The unique Huffman decoding trie for this file is as shown below. For example, the letter b corresponds to the binary seqeuence `001`.



A. BinaryTrie

The first and the most important part of this HW is to implement `BinaryTrie`, which is similar as what we have learned in lecture.

The `BinaryTrie` class shoud obey the following API.

```
public class BinaryTrie implements Serializable {
    public BinaryTrie(Map<Character, Integer> frequencyTable)
    public Match longestPrefixMatch(BitSequence querySequence)
    public Map<Character, BitSequence> buildLookupTable()
}
```

Constructor. Given a frequency table which maps symbols of type V to their relative frequencies, the constructor should build a Huffman decoding trie according to the procedure discussed in class. You may find implementations of Huffman codes on the web useful for inspiration, e.g. this implementation.

longestPrefixMatch. The `longestPrefixMatch` method finds the longest prefix that matches the given `querySequence` and returns a Match object for that Match. The Match class is a simple container class with the following API:

```
public class Match {
    public Match(BitSequence sequence, char symbol)
    public char getSymbol()
    public BitSequence getSequence()
}
```

The `longestPrefixMatch` method takes as an argument objects of type `BitSequence`, described in more detail below.

For example, for the example Trie given in the introduction, if we call `trie.longestPrefixMatch(new BitSequence("0011010001"))`, then we will get back a Match object containing `b` as the symbol and `001` as the BitSequence. The method is called `longestPrefixMatch` because `001` is the longest prefix of `0011010001` that is a match inside our decoding binary trie.

buildLookupTable. The `buildLookupTable` method returns the inverse of the coding trie. For example, for the example Trie given in the introduction, this method should return the same map as:

```
HashMap<Character, BitSequence> expected = new HashMap<Character, BitSequence>();
expected.put('a', new BitSequence("000"));
expected.put('b', new BitSequence("001"));
expected.put('c', new BitSequence("01"));
expected.put('d', new BitSequence("10"));
expected.put('e', new BitSequence("11"));
```

This is because the character `a` corresponds to the bitSequence `000`, the character `b` corresponds to the bitSequence `001` and so forth.

The implementation of `BinaryTrie` is:

```
import java.io.Serializable;

import java.util.HashMap;
import java.util.Map;

import edu.princeton.cs.algs4.MinPQ;

public class BinaryTrie implements Serializable {

    private Node root;
    private Map<Character, BitSequence> lookUpTable;
```

```
    private static class Node implements Comparable<Node>, Serializable {
        private final char value;
        private final int freq;
        private final Node left, right;

        Node(char value, int freq, Node left, Node right) {
            this.value = value;
            this.freq = freq;
            this.left = left;
            this.right = right;
        }

        // Is the node a leaf node?
        private boolean isLeaf() {
            assert ((left == null) && (right == null)) || ((left != null) && (right !=
null));
            return (left == null) && (right == null);
        }

        // Compare, based on frequency
        @Override
        public int compareTo(Node that) {
            return this.freq - that.freq;
        }
    }

    public BinaryTrie(Map<Character, Integer> frequencyTable) {
        root = buildTrie(frequencyTable);
        lookUpTable = new HashMap<>();
        buildCode(root, "");
    }

    private Node buildTrie(Map<Character, Integer> frequencyTable) {
        // Initialize priority queue with singleton trees
        MinPQ<Node> pq = new MinPQ<Node>();
        for (Character key : frequencyTable.keySet()) {
            pq.insert(new Node(key, frequencyTable.get(key), null, null));
        }
        // Merge two smallest trees
        while (pq.size() > 1) {
            Node left = pq.delMin();
            Node right = pq.delMin();
            Node parent = new Node('\0', left.freq + right.freq, left, right);
            pq.insert(parent);
        }
        return pq.delMin();
    }

    // Make a lookup table from symbols and their encodings
```

```
    private void buildCode(Node x, String s) {
        if (!x.isLeaf()) {
            buildCode(x.left, s + '0');
            buildCode(x.right, s + '1');
        } else {
            lookUpTable.put(x.value, new BitSequence(s));
        }
    }

    public Map<Character, BitSequence> buildLookupTable() {
        return lookUpTable;
    }

    private Node get(Node x, BitSequence querySequence, int d) {
        if (x == null) {
            return null;
        }
        if (d == querySequence.length()) {
            return x;
        }
        int i = querySequence.bitAt(d);
        if (i == 0) {
            return get(x.left, querySequence, d + 1);
        } else {
            return get(x.right, querySequence, d + 1);
        }
    }

    private char get(BitSequence querySequence) {
        Node x = get(root, querySequence, 0);
        if (x == null) {
            return '\0';
        } else {
            return x.value;
        }
    }

    public Match longestPrefixMatch(BitSequence querySequence) {
        int length = search(root, querySequence, 0, 0);
        return new Match(querySequence.firstNBits(length),
get(querySequence.firstNBits(length)));
    }

    private int search(Node x, BitSequence querySequence, int d, int length) {
        if (x == null) {
            return length;
        }
        if (x.value != '\0') {
            length = d;
```

```
        }
        if (d == querySequence.length()) {
            return length;
        }
        int i = querySequence.bitAt(d);
        if (i == 0) {
            return search(x.left, querySequence, d + 1, length);
        } else {
            return search(x.right, querySequence, d + 1, length);
        }
    }
}
```

I refer a lot of materials from Huffman. What's more, you could refer a quite amount of inspirations from *Algorithm book*.

Some ideas are quite interesting such as using `MinPQ` to get the less frequency symbol.

B. HuffmanEncoder

Once you've written AND tested your BinaryTrie, implement the class HuffmanEncoder, with the following API:

```
public class HuffmanEncoder {
    public static Map<Character, Integer> buildFrequencyTable(char[] inputSymbols)
    public static void main(String[] args)
}
```

buildFrequencyTable. The `buildFrequencyTable` method should map characters to their counts. For example, suppose we have the character array ['a', 'b', 'b', 'c', 'c' , 'c', 'c', d', 'd', 'd', 'd', 'd', e', 'e', 'e', 'e', 'e', 'e'], then this method should return a map from 'a' to 1, 'b' to 2, and so forth.

The main method. The main method should open the file given as the 0th command line argument (`args[0]`), and write a new file with the name `args[0] + ".huf"` that contains a huffman encoded version of the original file. For example `java HuffmanEncoder watermelonsugar.txt` should generate a new Huffman encoded version of `watermelonsugar.txt` that contains `watermelonsugar.txt.huf`.

Pseudocode for the Huffman encoding process is given below:

```
1: Read the file as 8 bit symbols.
2: Build frequency table.
3: Use frequency table to construct a binary decoding trie.
4: Write the binary decoding trie to the .huf file.
5: (optional: write the number of symbols to the .huf file)
6: Use binary trie to create lookup table for encoding.
7: Create a list of bitsequences.
8: For each 8 bit symbol:
    Lookup that symbol in the lookup table.
    Add the appropriate bit sequence to the list of bitsequences.
9: Assemble all bit sequences into one huge bit sequence.
10: Write the huge bit sequence to the .huf file.
```

Some of these tasks are tricky and require knowledge of special libraries. To save time, we have provided a number of utility methods to make this process easier for you. Using these methods is optional.

```
1: char[] FileUtils.readFile(String filename)
4/5/10: ObjectWriter's writeObject method.
9: BitSequence BitSequence.assemble(List<BitSequence>)
```

See `ObjectWritingAndReadingDemo.java` for a demo of how to use the `ObjectWriter` and `ObjectReader` classes to write Java objects to files for later loading.

Important: Do not call writeObject once for each symbol! This will result in huge files, very slow performance, and a very complex decoder! For your sanity, use `BitSequence.assemble`!

Try running your file on the provided text files: `watermelonsugar.txt` and `signalalarm.txt`. You should see a modest decrease in file size for both. Your code should take no more than seconds to execute. There are no tests for `HuffmanEncoder` because the precise behavior is not specified.

The implementation of HuffmanEncoder is:

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

import java.util.List;

public class HuffmanEncoder {
    public static Map<Character, Integer> buildFrequencyTable(char[] inputSymbols) {
        Map<Character, Integer> frequencyTable = new HashMap<>();
        for (char c : inputSymbols) {
            if (frequencyTable.containsKey(c)) {
                frequencyTable.put(c, frequencyTable.get(c) + 1);
            } else {
                frequencyTable.put(c, 1);
            }
        }
        return frequencyTable;
```

```
    }

    public static void main(String[] args) {
        char[] someFile = FileUtils.readFile(args[0]);
        Map<Character, Integer> frequencyTable = buildFrequencyTable(someFile);
        BinaryTrie decodingTrie = new BinaryTrie(frequencyTable);
        ObjectWriter ow = new ObjectWriter(args[0] + ".huf");
        ow.writeObject(decodingTrie);
        Map<Character, BitSequence> lookUpTable = decodingTrie.buildLookupTable();
        List<BitSequence> bitSequenceList = new ArrayList<>();
        for (char c : someFile) {
            BitSequence sequence = lookUpTable.get(c);
            bitSequenceList.add(sequence);
        }
        ow.writeObject(BitSequence.assemble(bitSequenceList));
    }
}
```

Nothing tricky, just following the official guide.

C. HuffmanDecoder

Once you've written HuffmanEncoder and verified that it is able to generate files that are smaller than the ones passed in, write a class `HuffmanDecoder` that reverses the process, with the following API:

```
public class HuffmanDecoder {
    public static void main(String[] args)
}
```

The main method. The main method should open the file given as the 0th command line argument (`args[0]`), decode it, and and write a new file with the name given as `args[1]`. For example `java HuffmanDecoder watermelonsugar.txt.huf originalwatermelon.txt` should decode the contents of `watermelonsugar.txt.huf` and write them into `originalwatermelon.txt`.

Pseudocode for the Huffman decoding process is given below:

```
1: Read the Huffman coding trie.
2: If applicable, read the number of symbols.
3: Read the massive bit sequence corresponding to the original txt.
4: Repeat until there are no more symbols:
    4a: Perform a longest prefix match on the massive sequence.
    4b: Record the symbol in some data structure.
    4c: Create a new bit sequence containing the remaining unmatched bits.
5: Write the symbols in some data structure to the specified file.
```

As above, we have provided utility methods to make your life easier:

```
1/2/3: ObjectReader's readObject method.
4c: BitSequence has methods that may be useful to you.
5: FileUtils.writeCharArray(String filename, char[] chars)
```

Your `HuffmanDecoder` should **perfectly** decode the output of your `HuffmanEncoder`. For example, if we run the following commands:

```
java HuffmanEncoder watermelonsugar.txt
java HuffmanDecoder watermelonsugar.txt.huf originalwatermelon.txt
diff watermelonsugar.txt originalwatermelon.txt
```

Then the output of the `diff` command should be nothing. This is because `diff` is a command line tool that compares two files, and prints out any differences. If the files have no differences, nothing is output.

Your `HuffmanEncoder` and `HuffmanDecoder` should work for ANY file, not just English text, and even if the input isn't a text file at all!

The implementation of HuffmanDecoder is:

```
import java.util.ArrayList;
import java.util.List;

public class HuffmanDecoder {

    public static void main(String[] args) {
        ObjectReader or = new ObjectReader(args[0]);
        Object trie = or.readObject();
        Object sequence = or.readObject();
        BinaryTrie decodingTrie = (BinaryTrie) trie;
        BitSequence decodingSequence = (BitSequence) sequence;
        List<Character> charList = new ArrayList<>();

        while (decodingSequence.length() >= 1) {
            Match match = decodingTrie.longestPrefixMatch(decodingSequence);
            charList.add(match.getSymbol());
            decodingSequence = decodingSequence.lastNBits(decodingSequence.length()
                    - match.getSequence().length());
        }
        char[] chars = new char[charList.size()];
        for (int i = 0; i < charList.size(); i++) {
            chars[i] = charList.get(i);
        }
        FileUtils.writeCharArray(args[1], chars);
    }
}
```

Nothing tricky, just following the official guide.