Important

There are a few guidelines you must follow in this homework. If you fail to follow any of the following guidelines you will receive a **0** for the entire assignment.

1. All submitted code must compile under **JDK 7**. This includes unused code, don't submit extra files that don't compile. (Java is backwards compatabile so if it compiles under JDK 6 it *should* compile under JDK 7)
2. Don't include any package declarations in your classes.
3. Don't change any *existing* class headers or method signatures. (It is fine to add extra methods and classes)
4. Don't import anything that would trivialize the assignment. (e.g. don't import `java.util.LinkedList` for a Linked List assignment. Ask if you are unsure.)
5. You must submit your source code, the `.java` files, not the compiled `.class` files.

After you submit your files redownload them and run them to make sure they are what you intended to submit. We are not responsible if you submit the wrong files.

Assignment

For this assignment you will be writing general graph search. There are many different ways you could program this, in this assignment you will use functions and some anonymous inner classes to complete this assignment.

Functional Programming

You have been provided a very simple interface called function. This is sort of hack to allow you to pass around functions to methods, rather than just objects. (Although technically this is still an object)

You have seen objects that have similar purposes. A Comparator is a function that always maps two objects to an int. Here we just have one object, that we are mapping to the return type of our choice.

Of course sometimes you need to have multiple parameters in your function, for this assignment we never need more than two, so an easy way to get around that is to introduce a Pair class. This Pair class is just a container for two objects.

Let's look at some examples. I have this array, and I want to give someone a method to change the objects of the array in any way they want. So I write a transform method, that takes in a Function, and the Array of objects.

```
public static <T> void transform(T[] array, Function<T, T> f) {
   for (int i = 0; i < array.length; i++) {
      array[i] = f.apply(array[i]);
   }
}
```

Now let's say I have this array of Strings I want to work with, I could write functions to do all sorts of things, and would have this general framework to apply the function over all of the objects I want.

```
public static void main(String[] args) {
    // This function reverses a String
    Function<String, String> reverse = new Function<String, String>() {
        @Override
        public String apply(String o) {
            char[] c = o.toCharArray();
            for (int i = 0; i < c.length / 2; i++) {
                char tmp = c[i];
                c[i] = c[c.length - 1 - i];
                c[c.length - 1 - i] = tmp;
            }
            return new String(c).
        }
    };

    // Now everything in args is reveresed
    transform(args, reverse);

    // We can even do something as simple as print out the String
    Function<String, String> print = new Function<String, String>() {
        @Override
        public String apply(String o) {
            System.out.print(o + ",␣");
            return o;
        }
    };

    // Now we just printed out each string in our array
    transform(args, print);
}
```

That's a basic overview of Functions, and just one use of them. If you want to learn more about functions you can check them out from Google's guava library: `http://code.google.com/p/guava-libraries/wiki/FunctionalExplained`

This is an ugly little hack to introduce lambdas to Java. Syntax support for lambdas is supposed to be coming in Java 8.

General Graph Search

You will be implementing general graph search in this assignment. You have several static search methods. In the first you are implementing general graph search for an unweighted graph. And in the second you will be implementing Dijkstra's algorithm for a weighted graph.

`search(...)`

You are given a start vertex, an empty structure, a function to find successors, and a goal function.

1. **Start Vertex:** The start vertex is where you should start the search from.
2. **Structure:** The structure is what you should add/remove successors to/from.
3. **Successor Function:** The successor function will return a list of adjacent vertices to whatever vertex it is given.

4. **Goal Function:** The goal function will return true when you have found the goal, at this point you can terminate the search. (This function also serves as a way to perform an operation on each vertex you visit, see the example in Test)

`distance(...)`

This method performs Dijkstra's on a weighted graph. You are given a start vertex, a distance function, a successors function, and a goal function.

1. **Start Vertex:** The start vertex is where you should start the search from.
2. **Distance Function:** The distance function takes in a pair of vertices. When applied to the pair, it will give you the distance from a to b. (these are fields in Pair.java) The distance is not necessarily symmetric, so make sure to order the objects in pair correctly. If the vertices in the pair are not connected null will be returned.

   You are guaranteed that if you apply this function to x, and a vertex in the list `successors.apply(x)`, null will not be returned.
3. **Successor Function:** The successor function will return a list of adjacent vertices to whatever vertex it is given.
4. **Goal Function:** The goal function will return true when you have found the goal, at this point you can terminate the search.

   This is different from the previous goal function in that you must provide the distance the current vertex is from the start vertex when checking it.
5. **Priority Queue:** In this method you need to create a priority queue (`java.util.PriorityQueue`) of `Pair<T, Integer>`, where the second paramater of the Pair is the distance to the corresponding vertex in the Pair.

   You will also need to create your own comparator for the priority queue to order these pairs, smallest distance first, the vertex in the pair should not affect the ordering. (There is no constructor for priority queue that just takes in a comparator, you must give it an initial size, and then a comparator)

Impl

You will also be implementing some practical functions and structures that you can use with this search algorithm.

`adjacecnyList(...)`

This takes in a map of a vertex to a list of the vertices it is adjacent to. You should use this map to compose a function that behaves like an adjacency list.

`euclideanDistanceSquared()`

This returns a function that calculates the euclidean distance squared between two points. Note that this is always an integer when you are using lattice points.

(use Google if you don't know what euclidean distance is)

`manhattanDistance()`

This returns a function that calculates the manhattan distance between two points.

(use Google if you don't know what manhattan distance is)

`queue()`

This returns a structure where the add and remove methods simulate a queue. Note that this can be used to run BFS

`stack()`

This returns a structure where the add and remove methods simulate a stack. Note that this can be used to run DFS

Test

A test class has been provided for you with some examples of functions. Once the assignment is complete running the Test class will run simulations of the Collatz Conjecture. The given start number of 1000 should print out all 111 steps to arrive at a 1.

Deliverables

You must submit all of the following files.

1. `Search.java`
2. `Impl.java`

You may attach them each individually, or submit them in a zip archive.