CS-1181 Project 1: Genetic Algorithm

PURPOSE: For this project, you will be attempting to solve a version of the bin packing problem, which is a famous problem in computer science. Pretend that your town is being attacked by zombies, and you have to abandon your house and go on the run. (It's possible that this isn't exactly how the problem is classically described, but this version is way more interesting.) You are only able to carry 10 pounds of stuff with you in addition to food and other necessities, and you want to bring things that you can sell for the greatest amount of money possible. Below is a list of items you could take, along with their weight and selling price. Which items should you take with you in order to maximize the amount of money you can get?

Item	Weight (lbs)	Value ($)
Cell phone	0.25	600
Gaming laptop	10	2000
Jewelry	0.5	500
Kindle	0.5	300
Video game console	3	500
Small cuckoo clock	5	1500
Silver paperweight	2	400

It turns out that the best you can do in this situation is to take:

- cell phone
- jewelry
- Kindle
- video game console
- small cuckoo clock
- **Weight: 9.25, Value: $3400**

The tricky thing about the bin packing problem is that the only way you can be *certain* that you have the optimal set of items is to try all possible combinations. You might be tempted to try short cuts, like taking items in order of worth until you hit the weight limit:

- gaming laptop
- **Weight: 10, Value: $2000**

Or you could try taking the lightest items until you reach the weight limit:

- cell phone
- jewelry
- Kindle
- silver paperweight
- video game console
- **Weight: 6.25, Value: $2300**

Neither of these strategies nets as much money as the optimal combination. However, trying all possible combinations is a lot of work. Instead, we can use a *genetic algorithm* to arrive at a high-value set of items faster. Genetic algorithms are based on the concept of natural selection. They allow us to explore a search space by "evolving" a set of solutions to a problem that score well against a fitness function. The solution we end up with is not guaranteed to be the optimal one, but it is likely to at least be pretty good. Here's how it works:

PROBLEM: For this project, an individual `Chromosome` is represented as a sequence of seven `Item` objects that correspond to the seven items defined above. The `Item` class contains fields for the item type, its weight, value, and a boolean field called `included`. False indicates leaving the item behind, and true indicates taking it with us. If a chromosome has items with the following included values:

```
T   F   F   F   T   T   F
```

it means we should take the cell phone, video game console, and small cuckoo clock.

Next, we need to develop some way to measure the "fitness" of each chromosome. A set of items is better if it is worth more, as long as it weighs 10 pounds or less. Therefore, the following can be used as the fitness function:

$$\texttt{fitness} = \begin{cases} \texttt{0} & \texttt{if weight > 10} \\ \texttt{value} & \texttt{if weight <= 10} \end{cases}$$

where weight equals the total weight of all of the items **included** in the chromosome, and value equals the total value of all of the items **included** in the chromosome.

Crossover

The first operation you will need to implement is crossover. First, randomly choose two parents (`p1` and `p2`) from the population. Those two parents will create a child whose DNA is related to the parents'. For each of the seven items in the chromosome, randomly pick a number between 1 and 10 and use it to choose which parents' value the child will get. Then, use the function below to set the child's included values:

$$\texttt{included value of child's item} = \begin{cases} \texttt{included value from p1's item} & \texttt{if 1 <= rand <= 5} \\ \texttt{included value from p2's item} & \texttt{if 6 <= rand <= 10} \end{cases}$$

where `rand` is the random number for the current item.

For example, suppose the two random parents have the following included values:

```
Parent 1:  T   F   T   F   T   T   F
Parent 2:  T   T   T   F   F   T   T
```

Then, let's assume our seven random numbers are:

```
           1   4   10  3   6   6   9
```

Therefore, the child's set of item included fields would be:

```
Child:     T   F   T   F   F   T   T
```

Mutation

The next operation is mutation. Start by choosing a single individual from the population and again generate a random number between 1 and 10 for each item. Mutation generally happens more rarely than crossover. Thus, apply the following function to each of the items in the individual:

$$\texttt{item's included value} = \begin{cases} \texttt{keep the item's included value} & \texttt{if 2 <= rand <= 10} \\ \texttt{flip the item's included value} & \texttt{if rand == 1} \end{cases}$$

where `rand` is the random number for the current item.

For example, assume the chosen individual's included values are:

```
T    T    F    T    T    T    F
```

and the seven random numbers are:

```
5    1    7    8    9    2    1
```

Then after mutation, the second and last items' included values are flipped and the individual now looks like this:

```
T    F    F    T    T    T    T
```

Running the Genetic Algorithm

Now with the above pieces, the algorithm itself can be implemented as follows (keep in mind that an individual refers to a single chromosome):

1. Create a set of ten random individuals to serve as the **currentPopulation**
2. Add each of the individuals in the **currentPopulation** to the **nextGeneration**
3. Randomly pair off individuals, perform crossover to create a child, and add it to the **nextGeneration**
4. Randomly choose ten percent of the individuals in the **nextGeneration** and mutate them
5. Sort the individuals in the **nextGeneration** according to their fitness
6. Clear out the **currentPopulation**
7. Add the top ten of the **nextGeneration** back into the **currentPopulation**
8. Repeat steps 2 through 7 twenty times
9. Sort the **currentPopulation**
10. Display the fittest individual to the console

REQUIREMENTS: For this project, you will need to implement the following three classes. Your code must conform to this specification exactly (i.e. your classes and methods must be named as shown below and you must have the same method signatures).

```
public class Item {
    // A label for the item, such as "Jewelry" or "Kindle"
    private final String name;

    // The weight of the item in pounds
    private final double weight;

    // The value of the item rounded to the nearest dollar
    private final int value;

    // Indicates whether the item should be taken or not
    private boolean included;

    // Initializes the Item's fields to the values that are passed in
    // The included field is initialized to false
    public Item(String name, double weight, int value) {}

    // Initializes this item's fields to the be the same as the other item's
    public Item(Item other) {}
    // Getter for the item's fields
    // You don't need a getter for the name
    public double getWeight() {}
    public int getValue() {}
    public boolean isIncluded() {}

    // Setter for the item's included field
    // You don't need setters for the other fields
    public void setIncluded(boolean included) {}

    // Displays the item in the form <name> (<weight> lbs, $<value>)
    public String toString() {}
}

public class Chromosome extends ArrayList<Item> implements Comparable<Chromosome> {
    // Used for random number generation
    private static Random rng;

    // This no-arg constructor can be empty
    public Chromosome() {}

    // Adds a copy of each of the items passed in to this Chromosome
    // Uses a random number to decide whether each item's included field
    // is set to true or false
    public Chromosome(ArrayList<Item> items) {}

    // Creates and returns a new child chromosome by performing the crossover
    // operation on this chromosome and the other one that is passed in
    // (i.e. for each item, use a random number to decide which parent's item
    // should be copied and added to the child)
    public Chromosome crossover(Chromosome other) {}

    // Performs the mutation operation on this chromosome (i.e. for each item in this
    // chromosome, use a random number to decide whether or not to flip it's included
    // field from true to false or vice versa)
    public void mutate() {}

    // Returns the fitness of this chromosome. If the sum of all of the included
    // items' weights are greater than 10, the fitness is zero. Otherwise, the
    // fitness is equal to the sum of all of the included items' values.
    public int getFitness() {}

    // Returns -1 if this chromosome's fitness is greater than the other's fitness,
    // +1 if this chromosome's fitness is less than the other one's,
    // and 0 if their fitness is the same
    public int compareTo(Chromosome other) {}

    // Displays the name, weight and value of all items in this chromosome whose
    // included value is true, followed by the fitness of this chromosome
    public String toString() {}
}

public class GeneticAlgorithm {
    // Reads in a data file with the format shown below and creates and returns an
    // ArrayList of Item objects
    //    item1_label, item1_weight, item1_value
    //    item2_label, item2_weight, item2_value
    //    ...
    public static ArrayList<Item> readData(String filename)
            throws FileNotFoundException {}
    // Creates and returns an ArrayList of populationSize Chromosome objects that
    // each contain the items, with their included field randomly set to true / false
    public static ArrayList<Chromosome> initializePopulation(ArrayList<Item> items,
            int populationSize) {}

    // Reads the data about the items in from a file called items.txt and performs
    // the steps described in the Running the Genetic Algorithm section above
    public static void main(String[] args) throws FileNotFoundException {}
}
```

RUBRIC:

(80 pts) **Base Functionality**

- [10] `Item` class is implemented correctly
- [35] `Chromosome` class is implemented correctly
- [10] `readFile` method is implemented correctly
- [5] `initializePopulation` method is implemented correctly
- [20] `main` method is implemented correctly

(20 pts) **Style**

- [10] Code is clearly written and follows standard conventions (variable names, indentation, etc.)
- [10] The code is meaningfully commented

IMPORTANT NOTES:

- **Submissions that do not compile will receive a zero**
- **Submissions with improperly cited AI will receive a zero and an academic integrity violation**
- **Submissions that are partially or fully copied from another submission will receive a zero and an academic integrity violation**