**Project 5**

For this project we're going back to where we started the semester. Your task is to improve on your genetic algorithm-based solution to the bin packing problem from Project 1. In particular you need to do the following:

0. Fix any issues you had with Project 1 originally
1. Add the capability to find the optimal solution for small instances of the problem
2. Multithread your genetic algorithm

Fixing issues with Project 1

Before beginning on the meat of this assignment, you need to make sure you're working from a solid foundation. A good way to get started here is to take a look at the grading feedback for your Project 1 submission. Another thing to consider is running your program on the more_items.txt file available on Pilot. You should get an optimal value of around $7600. Keep in mind that due to the random nature of genetic algorithms, you may need to run your program multiple times to fully analyze its results. You might also need to tweak your population size, number of epochs, and mutation rate to handle larger datasets.

Finding the optimal solution for small problem instances

The bin packing problem is an example of a special class of problems called NP-complete. This means that the only way to be sure that you have the optimal set of items (the one worth the most while still under the weight limit), is to try all possible combinations. For example, if you have three items: A, B, and C, you will need to check the following possible sets: A, B, C, A and B, A and C, B and C, and A, B and C.

To meet this requirement, add a class to your project called BruteForce. Your class should contain the following method:

```
public static ArrayList<Item> getOptimalSet(ArrayList<Item> items)
```

This method can call any helper methods (also written by you) that you would like. Note that the number of possible combinations of items is 2 raised to the power of n, where n is the number of items. This number gets very big very fast. Because of this, if your method is called with an ArrayList that contains more than 10 items, it should throw an InvalidArgumentException rather than try to compute the answer. Your BruteForce class should have its own main method, that reads in the Items from the data file and outputs the optimal set.

Note that the idea of finding all possible subsets of a set of items is somewhat similar to the practice problem you did on recursion, where you found all possible permutations of the letters in a string. That would be a great starting point for this!

Multithreading

This part of the project is separate from the first part and will have its own main method. The task here is to modify your genetic algorithm code from Project 1 so that the work is divided amongst multiple threads. There are several ways to do this, but here is the approach you should take for this assignment:

If your code does not already have these, add constants to control the values of key parameters like population size, the number of epochs (iterations) the genetic algorithm should run for, and the number of threads you will have:

```
public static final int POP_SIZE = 100;
public static final int NUM_EPOCHS = 1000;
public static final int NUM_THREADS = 1;
```

After your code initializes the population based on the data file containing information about the available items, create the desired number of threads. Each thread should be passed a copy of the current population of Chromosome objects. The thread's run methods should run for NUM_EPOCHS / NUM_THREADS iterations, doing the normal genetic algorithm operations of crossover, mutation, and killing off the weakest individuals. When the desired number of iterations is met, the main method should join back up with all of the worker threads and retrieve their best POP_SIZE / NUM_THREADS Chromosomes (for example via a getter method in your thread class). It should then sort the results across all of the threads and display the best set of items, along with its fitness.

Run your program several times for different numbers of threads and record the average fitness and time each run takes. Provide this information as a comment at the top of your driver program.

Note that there is some overhead associated with creating and managing threads. This means that threading a program will only be a big time-saver in certain situations. For genetic algorithms, this generally happens when computing a chromosome's fitness is an expensive operation. To simulate this, please add the following code to your Chromosome class:

```
public static long dummy = 0; // this is a class field
```

And in your getFitness method:

```
dummy = 0;
for (int i=0; i<this.size()*1000; i++) {
    dummy += i;
```

```
        }

       // the rest of the method is the same as before
```

Your program will be graded according to this rubric:

[20 points] Project 1 requirements are met – any problems from before have been resolved.

[20 points] Brute-force solution produces the correct result for all valid inputs and throws an exception when the input is too large.

[40 points] Multithreading is implemented correctly, and the requested performance data is included.

[10 points] Good object-oriented development principles are followed. The program is logically organized into classes with appropriate fields and methods.

[10 points] The code is clearly written, including following standard coding conventions and containing meaningful comments.