TCSS 342 - Data Structures
Assignment 2 - Evolved Names

Due Date: Friday, April 24th

Guidelines

This assignment consists of programming and written work. Solutions should be a complete working Java program including your original work or **cited contributions** from other sources. These files should be compressed in a .zip file for submission through the Canvas link.

This assignment is to be completed on your own or in a group of two. If you choose to work in a group of two this must be clear in your submission. Please see the course syllabus or the course instructor for clarification on what is acceptable and unacceptable academic behavior regarding collaboration outside of a group of two.

Assignment

Imagine a virtual world in which all that exists are strings of characters from the set

{ A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, _, -, ' }

(The '_' represents the space character.) Strings in this world can reproduce new strings and die if they are not fit enough. You will evolve strings in this world until they spell your name. To do this you will create a Genome class which will contain a list of characters from the above set representing a string in your world, and you will create a Population class which will contain a list of Genomes representing all the strings in your world.

Your Genome class must:

- have some internal representation of the string of characters.
- initialize a new string to the default value 'A'.
- be able to **mutate** by:
    - possibly **adding a new character** somewhere in the string.
    - possibly **deleting a randomly selected character** from the string.
    - possibly **changing a character** in the string to a different value.
- be able to **crossover** with another genome:
    - given two Genomes create a third that is a combination of the two.
- be able to measure Genome **fitness**:
    - using one of the two **zero-based fitness** methods listed here:
        - calculate how close the string in the Genome is from your name using the simple method detailed below.
        - **(Optional)** calculate how close the string in the Genome is from your name using the Levenshtein edit-distance.
- display:
    - output the string and its fitness in an easy to read format.

Your Population class must:

- maintain a list of Genomes representing the current population.
- initialize the population with a fixed number of default Genomes.
- update the list of Genomes every breeding cycle by:
    - removing the least-fit members of the population.
    - mutating or breeding the most-fit members of the population.
    - **Note:** Because we are using a zero-based fitness the “most fit” member of the population has the lowest fitness score not the highest.
- display the entire population.
- display the most-fit individual in the population.

The Main class is a controller and will:

- instantiate the Population class.
    - I use 100 genomes and a mutation rate of 0.05.
- call day() from the Population class until the fitness of the most fit genome is zero.
- output simulation progress.
- output runtime statistics.

Formal Specifications

Your simulation will implement the Genome class according to this interface:

- Genome(double *mutationRate*) - a constructor that initializes a Genome with value ‘A’ and assigns the internal mutation rate a double between 0 and 1.
- Genome(Genome *gene*) - a copy constructor that initializes a Genome with the same values as the input *gene*.
- void mutate() - this function mutates the string in this Genome using the following rules:
    - with *mutationRate* chance add a randomly selected character to a randomly selected position in the string.
    - with *mutationRate* chance delete a single character from a randomly selected position of the string but do this only if the string has length at least 2.
    - for each character in the string:
        - with *mutationRate* chance the character is replaced by a randomly selected character.
- void crossover(Genome *other*) - this function will update the current Genome by crossing it over with *other*.
    - Create the new list by following these steps for each index in the string starting at the first index:
        - Randomly choose one of the two parent strings.
        - If the parent string has a character at this index (i.e. it is long enough) copy that character into the new list. Otherwise end the new list here.
- Integer fitness() - returns the fitness of the Genome calculated using the following algorithm:

    - Let *n* be the length of the current string. Let *m* be the length of the target string.
    - Let *l* be the max(*n, m*).
    - Let *f* be initialized to |*m* - *n*|.
    - For each character position $1 \leq i \leq l$ add one to *f* if the character in the current string is different from the character in the target string (or if one of the two characters does not exist). Otherwise add nothing to *f*.
    - Return *f*.
- **(Optional)** Integer fitness() - instead of the algorithm above use the Wagner-Fischer algorithm for calculating Levenshtein edit distance:
    - Let *n* be the length of the current string. Let *m* be the length of the target string.
    - Create an *(n + 1) x (m + 1)* matrix *D* initialized with 0s.
    - Fill the first row of the matrix with the column indices and fill the first column of the matrix with the row indices.
    - Implement this nested loop to fill in the rest of the matrix.

```
for row from 1 to n
   for column from 1 to m
      if(current[row-1] == target[column-1]) D[i,j] = D[i-1,j-1]
      else D[i,j] = min(D[i-1,j]+1,D[i,j-1]+1,D[i-1,j-1]+1)
```

    - Return the value stored in *D[n,m] + (abs(n - m) + 1) / 2*. (Use integer arithmetic.)
- String toString() - this function will display the Genome's character string and fitness in an easy to read format.

Your simulation will implement the Population class. It must function according to this interface :

- String *target* - a public data element that is initialized to your name.
- Genome *mostFit* - a public data element that is equal to the most-fit Genome in the population.
- void Population(Integer *numGenomes*, Double *mutationRate*) - a constructor that initializes a Population with a number of default genomes (see above).
- void day() - this function is called every breeding cycle and carries out the following steps:
    - update *mostFit* variable to the most-fit Genome in the population. (Remember this is the genome with the **lowest fitness**!)
    - delete the least-fit half of the population.
    - create new genomes from the remaining population until the number of genomes is restored by doing either of the following with equal chance:
        - pick a remaining genome at random and clone it (with the copy constructor) and mutate the clone.
        - pick a remaining genome at random and clone it and then crossover the clone with another remaining genome selected at random and then mutate the result.

You will also provide a Main class for control and testing of your evolutionary algorithm.

- void main(String[] args) - this method should instantiate a population and call day() until the target string is part of the population.
    - The target string has fitness zero so the loop should repeat until the most fit genome has fitness zero.
    - After each execution of day() output the most fit genome.
    - To measure performance output the number of generations (i.e times day() is called) and the execution time.
- void testGenome() - this method tests the Genome class.
- void testPopulation() - this method tests the Population class.

Include any other methods used to test components of your Genome and Population classes.

Analysis

In addition to this programming assignment you will also complete a detailed worst-case runtime analysis of your day() method in your Population class implementation. You will assume for your analysis that the *n* is the number of genomes in the population and that *m* is the length of the longest Genome in the population. For the day method provide:

- A line-by-line analysis of operation costs.
- An expression of each loop in the code as a summation of terms.
- An expression of the entire method's cost as a sum of individual line terms and summation terms.
- A simplified expression of the cost of the method.
- A big-oh expression of the cost of the method.

To carry out this analysis you will have to analyse any other methods in Population or Genome that are called by the day method as well. You do not need to present formal analyses of these methods but you must at least include big-oh runtime bounds for the methods called by day().

Submission

The following files are provided for you:

- trace.txt - an example trace of my solution.

You will submit a .zip file containing:

- Genome.java - an implementation of the Genome class satisfying the above criteria.
- Population.java - an implementation of the Population class satisfying the above criteria.
- Main.java - a controller for your evolutionary system that serves to run the evolutionary simulation and test the components of your code.
- Analysis.pdf - a pdf containing the runtime analysis of the day method from the Population class.

Grading Rubric

Each of the following will be awarded **one or more** points toward your assignment grade. Not all points need to be achieved to receive a perfect grade. Excess points contribute to the total points gathered for the quarter. Excess points at the end of the quarter will convert into a bonus assignment grade.

Genome

- Constructor correct and efficient.
- Copy constructor correct and efficient.
- Mutate is correct and efficient.
- Crossover is correct and efficient.
- Fitness is correct and efficient.
- **(Optional)** Precise implementation of the Wagner-Fischer algorithm.
- Proper selection and use of internal data structures.

Population

- Constructor correct and efficient.
- Day is correct and efficient.
- Proper selection and use of data structures.

Main

- Simulation runs properly to completion.
- Simulation outputs the most fit in each generation.
- Simulation statistics are displayed.
- Tests of Genome class included.
- Tests of Population class included.
- Proper selection and use of data structures.

Analysis

- Line-by-line analysis is correct.
- Summations used properly.
- All terms are gathered into the sum properly.
- Simplification is correct.
- Correct big-oh bound.

Extra Points

- Compiles first try.
- All and only the files requested are submitted.
- First graded submission. (That is, not a resubmission after a grade awarded.)
- On time.
- Work alone.

Tips for maximizing your grade:

- Make sure your classes match the interface structure exactly. I will use my own controller (Main.java) and test files on your code and it should work without changes. Do not change the method name (even capitalization), return type, argument number, type, and order. Make sure all parts of the interface are implemented.
- Only zip up the .java files. If you use eclipse these are found in the “src” directory, not the “bin” directory. I can do nothing with the .class files.
- All program components should be in the default package. If you use a different package it increases the difficulty of grading and thus affects your grade.
- Place your name in the comments at the top of every file. If you are a group of two make sure both names appear clearly in these comments.