Grant Toepfer

Evolved Names Worst-Case Runtime Analysis

TCSS 342

```
public void day() {
    //kill weak half
    genes = genes.stream()
            .sorted((g1, g2) -> Integer.compare(g1.fitness(), g2.fitness()))
            .limit(numGenomes >> 1)
            .collect(Collectors.toList());

    //create new genes
    new Random().ints(numGenomes - genes.size(), 0, genes.size())
            .mapToObj(genes::get)
            .map(Genome::new)
            .peek(this::crossoverHalf)
            .peek(Genome::mutate)
            .forEach(genes::add);

    //get most fit
    mostFit = genes.stream().reduce((g1, g2) -> g1.fitness() < g2.fitness() ? g1 : g2).get();
}

private void crossoverHalf(Genome g) {
    Random random = new Random();
    if (random.nextBoolean()) {
        g.crossover(genes.get(random.nextInt(genes.size())));
    }
}
```

What it's doing:

The first stream sorts the list based on the fitness of the genomes, then it cuts that list in half, effectively killing half of the genomes, finally it puts the fit genomes into a new list and replaces the previous list.

The "create new genes" step starts by creating a list of integers equal to the size of the genomes that were killed in the previous step. These integers are in the range of 0 (inclusive) to the size of the list we just cut in half (exclusive), these integers bounds are also the bounds of indices to the genes list. The next line converts the stream from those indices to the genomes at those indices. The next line will call the copy constructor on the genomes and replace the stream with a bunch of copies of those genomes. The next two lines will mutate the genes: the first one calls crossoverHalf, which will randomly decide whether or not the genes will crossover with a currently fit gene, the second line will mutate all the genes. Finally, the last line will add these new genes to the genes list, restoring it to full capacity.

The "get most fit" step will call the given function on all the items in the list until there is only one left. The function will return the gene with the lower fitness score. Therefore, the stream itself will return the fittest genome.

Line-by-line analysis

```
genes = genes.stream()
```

creates a stream, can be thought of as the beginning of a for loop. This will be the beginning of S1.

```
.sorted((g1, g2) -> Integer.compare(g1.fitness(), g2.fitness()))
```

Sorts the list based on the fitness of the genomes. This uses a modified merge-sort. Making it O(nlgn). All three of the functions in the comparator are O(1) so the cost of this line is just O(nlgn). Let's call this S2.

```
.limit(numGenomes >> 1)
```

Cuts down the size of the stream. O(1) time. Let's call this C1.

```
.collect(Collectors.toList())
```

Collects all the genomes in the stream into a new list. O(1) time as it's just an add. Let's call this C2.

This ends S1. Sorted and limit are ran once each, while collect is ran for each item in the stream:

$$S_1 = O(nlgn) + C_1 + \sum_{i=0}^{n-1} C_2$$

$$S_1 = O(nlgn) + C_1 + nC_2$$

$$S_1 = O(nlgn) + O(1) + O(n)$$

$$S_1 = O(nlgn)$$

```
new Random().ints(numGenomes - genes.size(), 0, genes.size())
```

This is the beginning of the second stream. It calls random.nextInt at most n >> 1 times and instantiates a new random. We will call the cost of this line C3 + nC4 as well as the start of S2.

```
.mapToObj(genes::get)
```

This calls genes.get once for each item in the stream. Its cost is C5.

```
.map(Genome::new)
```

This calls new Genome(Genome), the copy constructor for each item in the stream. It's cost is mC6 + C7 since the copy constructor copies every character in the string.

```
.peek(this::crossoverHalf)
```

This calls the crossoverHalf method seen above for each item in the stream. Its cost is mC8 + mC9 + C10 since it calls the setFitness method for the genome as well as copies a string equal to at most the size of the larger parent string.

```
.peek(Genome::mutate)
```

Calls the mutate method on each item in the stream. Its cost is mC8 + C11 since it calls the setFitness method for the genome.

```
.forEach(genes::add);
```

Adds the new genomes to the genes list. Its cost is C12.

$$S_2 = C_3 + nC_4 + \sum_{i=0}^{n-1} C_5 + mC_6 + C_7 + mC_8 + mC_9 + C_{10} + mC_8 + C_{11} + C_{12}$$

$$S_2 = C_3 + nC_4 + n(C_5 + mC_6 + C_7 + mC_8 + mC_9 + C_{10} + mC_8 + C_{11} + C_{12})$$

$$S_2 = C_3 + nC_4 + n(m(C_6 + C_8 + C_8 + C_9) + C_5 + C_7 + C_{10} + C_{11} + C_{12})$$

$$S_2 = C + nC + n(m(C) + C)$$

$$S_2 = O(1) + O(n) + O(n * m)$$

$$S_2 = O(n * m)$$

```
mostFit = genes.stream().reduce((g1, g2) -> g1.fitness() < g2.fitness() ? g1 :
g2).get();
```

This one liner is S3. It consists of a stream initialization, a reduction, and a get. The stream initialization is a single n cost, the get is a constant time operation, and reduce is the meat of the line. Reduce will apply the given function sequentially to every genome in the stream. The cost of the function inside of reduce is constant time.

$$S_3 = n + C_{13} + \sum_{i=0}^{n-1} C_{14}$$

$$S_3 = n + C_{13} + nC_{14}$$

$$S_3 = O(n) + O(1) + O(n)$$

$$S_3 = O(n)$$

Overall, the total cost of the method is the sum of all three parts.

$$runtime = S_1 + S_2 + S_3$$

$$runtime = O(nlgn) + O(n * m) + O(n)$$

$$runtime = O(nlgn)$$