children so that the placement of letters in the two children ends up being a combination of the parents. `mutate()` swaps two random locations in the `letters` list.

We can plug `SendMoreMoney2` into `GeneticAlgorithm` just as easily as we plugged in `SimpleEquation`. But be forewarned: This is a fairly tough problem, and it will take a long time to execute if the parameters are not well tweaked. And there's still some randomness even if one gets them right! The problem may be solved in a few seconds or a few minutes. Unfortunately, that is the nature of genetic algorithms.

Listing 5.12 send_more_money2.py continued

```
if __name__ == "__main__":
    initial_population: List[SendMoreMoney2] = [SendMoreMoney2.random_
     instance() for _ in range(1000)]
    ga: GeneticAlgorithm[SendMoreMoney2] = GeneticAlgorithm(initial_
     population=initial_population, threshold=1.0, max_generations = 1000,
     mutation_chance = 0.2, crossover_chance = 0.7, selection_
     type=GeneticAlgorithm.SelectionType.ROULETTE)
    result: SendMoreMoney2 = ga.run()
    print(result)
```

The following output is from a run that solved the problem in 3 generations using 1,000 individuals in each generation (as created above). See if you can mess around with the configurable parameters of `GeneticAlgorithm` and get a similar result with fewer individuals. Does it seem to work better with roulette selection than it does with tournament selection?

```
Generation 0 Best 0.0040650406504065045 Avg 8.854014252391551e-05
Generation 1 Best 0.16666666666666666 Avg 0.001277329479413134
Generation 2 Best 0.5 Avg 0.014920889170684687
8324 + 913 = 9237 Difference: 0
```

This solution indicates that SEND = 8324, MORE = 913, and MONEY = 9237. How is that possible? It looks like letters are missing from the solution. In fact, if M = 0, there are several solutions to the problem not possible in the version from chapter 3. MORE is actually 0913 here, and MONEY is 09237. The 0 is just ignored.

5.5 Optimizing list compression

Suppose that we have some information we want to compress. Suppose that it is a list of items, and we do not care about the order of the items, as long as all of them are intact. What order of the items will maximize the compression ratio? Did you even know that the order of the items will affect the compression ratio for most compression algorithms?

The answer will depend on the compression algorithm used. For this example, we will use the `compress()` function from the `zlib` module with its standard settings. The solution is shown here in its entirety for a list of 12 first names. If we do not run the genetic algorithm and we just run `compress()` on the 12 names in the order they were originally presented, the resulting compressed data will be 165 bytes.

Listing 5.13 list_compression.py

```
from __future__ import annotations
from typing import Tuple, List, Any
from chromosome import Chromosome
from genetic_algorithm import GeneticAlgorithm
from random import shuffle, sample
from copy import deepcopy
from zlib import compress
from sys import getsizeof
from pickle import dumps

# 165 bytes compressed
PEOPLE: List[str] = ["Michael", "Sarah", "Joshua", "Narine", "David",
     "Sajid", "Melanie", "Daniel", "Wei", "Dean", "Brian", "Murat", "Lisa"]


class ListCompression(Chromosome):
    def __init__(self, lst: List[Any]) -> None:
        self.lst: List[Any] = lst

    @property
    def bytes_compressed(self) -> int:
        return getsizeof(compress(dumps(self.lst)))

    def fitness(self) -> float:
        return 1 / self.bytes_compressed

    @classmethod
    def random_instance(cls) -> ListCompression:
        mylst: List[str] = deepcopy(PEOPLE)
        shuffle(mylst)
        return ListCompression(mylst)

    def crossover(self, other: ListCompression) -> Tuple[ListCompression,
     ListCompression]:
        child1: ListCompression = deepcopy(self)
        child2: ListCompression = deepcopy(other)
        idx1, idx2 = sample(range(len(self.lst)), k=2)
        l1, l2 = child1.lst[idx1], child2.lst[idx2]
        child1.lst[child1.lst.index(l2)], child1.lst[idx2] =
     child1.lst[idx2], l2
        child2.lst[child2.lst.index(l1)], child2.lst[idx1] =
     child2.lst[idx1], l1
        return child1, child2

    def mutate(self) -> None: # swap two locations
        idx1, idx2 = sample(range(len(self.lst)), k=2)
        self.lst[idx1], self.lst[idx2] = self.lst[idx2], self.lst[idx1]

    def __str__(self) -> str:
        return f"Order: {self.lst} Bytes: {self.bytes_compressed}"

if __name__ == "__main__":
    initial_population: List[ListCompression] = [ListCompression.random_
     instance() for _ in range(1000)]
    ga: GeneticAlgorithm[ListCompression] = GeneticAlgorithm(initial_
     population=initial_population, threshold=1.0, max_generations = 1000,
```

```
    mutation_chance = 0.2, crossover_chance = 0.7, selection_
    type=GeneticAlgorithm.SelectionType.TOURNAMENT)
result: ListCompression = ga.run()
print(result)
```

Note how similar this implementation is to the implementation from SEND+MORE=MONEY in section 5.4. The `crossover()` and `mutate()` functions are essentially the same. In both problems' solutions, we are taking a list of items and continually rearranging them and testing those rearrangements. One could write a generic superclass for both problems' solutions that would work with a wide variety of problems. Any problem that can be represented as a list of items that needs to find its optimal order could be solved the same way. The only real point of customization for the subclasses would be their respective fitness functions.

If we run list_compression.py, it may take a very long time to complete. This is because we don't know what constitutes the "right" answer ahead of time, unlike the prior two problems, so we have no real threshold that we are working toward. Instead, we set the number of generations and the number of individuals in each generation to an arbitrarily high number and hope for the best. What is the minimum number of bytes that rearranging the 12 names will yield in compression? Frankly, we don't know the answer to that. In my best run, using the configuration in the preceding solution, after 546 generations, the genetic algorithm found an order of the 12 names that yielded 159 bytes compressed.

That's only a savings of 6 bytes over the original order—a ~4% savings. One might say that 4% is irrelevant, but if this were a far larger list that would be transferred many times over the network, that could add up. Imagine if this were a 1 MB list that would eventually be transferred across the internet 10,000,000 times. If the genetic algorithm could optimize the order of the list for compression to save 4%, it would save ~40 kilobytes per transfer and ultimately 400 GB in bandwidth across all transfers. That's not a huge amount, but perhaps it could be significant enough that it's worth running the algorithm once to find a near optimal order for compression.

Consider this, though—we don't really know if we found the optimal order for the 12 names, let alone for the hypothetical 1 MB list. How would we know if we did? Unless we have a deep understanding of the compression algorithm, we would have to try compressing every possible order of the list. Just for a list of 12 items, that's a fairly unfeasible 479,001,600 possible orders (12!, where ! means factorial). Using a genetic algorithm that attempts to find optimality is perhaps more feasible, even if we don't know whether its ultimate solution is truly optimal.

5.6 Challenges for genetic algorithms

Genetic algorithms are not a panacea. In fact, they are not suitable for most problems. For any problem in which a fast deterministic algorithm exists, a genetic algorithm approach does not make sense. Their inherently stochastic nature makes their runtimes unpredictable. To solve this problem, they can be cut off after a certain number of generations. But then it is not clear if a truly optimal solution has been found.