Assignment 3 - Genetic Algorithm

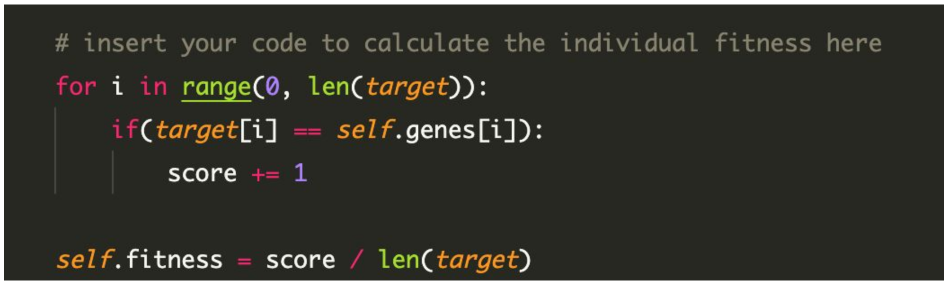
Juan Pablo Lozano Gómez

Requirements
Please install the requirements listed in requirements.txt file using pip (Used to plot).

Part 1. Implementation of GA

1. Fitness function

In my fitness function I decided to follow the approach from the lab and check the number of matching characters between target and the individual. This comparison is done position by position and since both strings have the same size we can iterate them at the same time without any issues. This gives a comparison in time complexity of O(n). I thought of doing something different but based on my research the best string comparison algorithms work on O(n) and focus on cases where the matching is not done position by position, therefore this was enough.

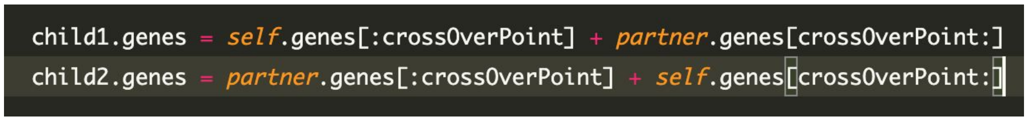


```python
# insert your code to calculate the individual fitness here
for i in range(0, len(target)):
    if(target[i] == self.genes[i]):
        score += 1

self.fitness = score / len(target)
```

2. Crossover method

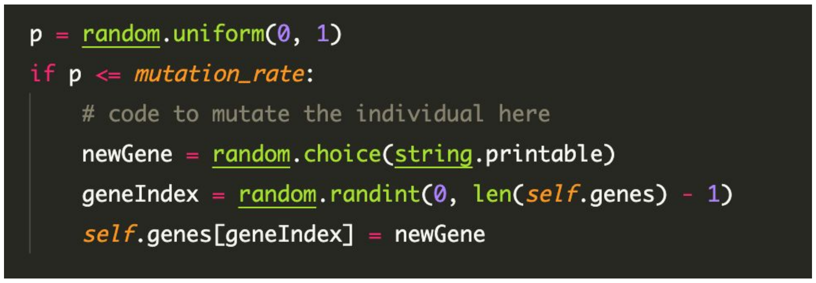
For the crossover I implemented two mechanisms: 1-point crossover and 2-point crossover. This setting can be changed in start.py. These crossovers are done by slicing the lists. No crossover rate was implemented therefore I assumed Pc = 1, which means that crossover is always done. Also the crossover mechanisms that I implemented create two childs instead of one. These are compared with their parents when I generate the new population.



```python
child1.genes = self.genes[:crossOverPoint] + partner.genes[crossOverPoint:]
child2.genes = partner.genes[:crossOverPoint] + self.genes[crossOverPoint:]
```

3. Mutation method

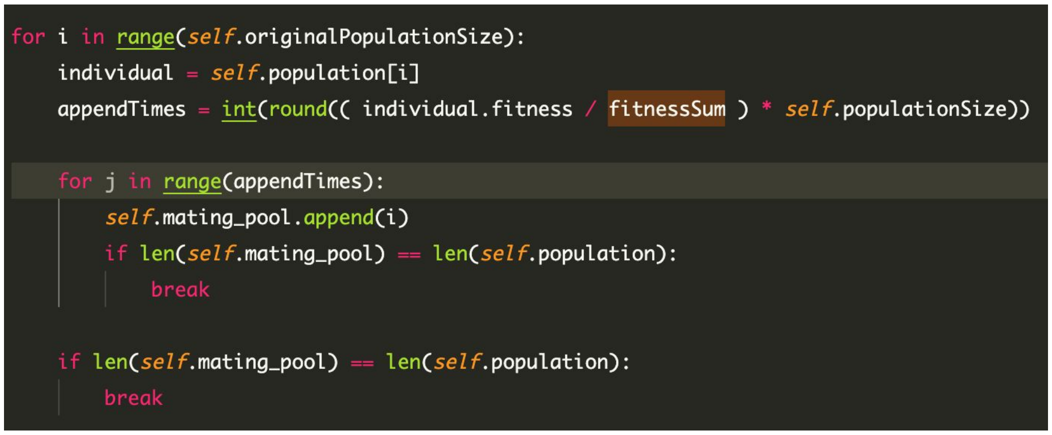
Mutation rate was implemented similar to what was explained in the lab. I grab a random number between 0 and 1 and check if it is below the mutation rate. I randomly select a position in the list of genes and mutate or change the character for a new one which is also selected randomly from the vocabulary available in **strings.printable.**



```
p = random.uniform(0, 1)
if p <= mutation_rate:
    # code to mutate the individual here
    newGene = random.choice(string.printable)
    geneIndex = random.randint(0, len(self.genes) - 1)
    self.genes[geneIndex] = newGene
```

4. Natural selection

For the natural selection process I create a mating pool that adds an individual N times. N is calculated by measuring what percentage of the total fitness of the populations corresponds to this individual. This percentage is then multiplied by the population size to get N.



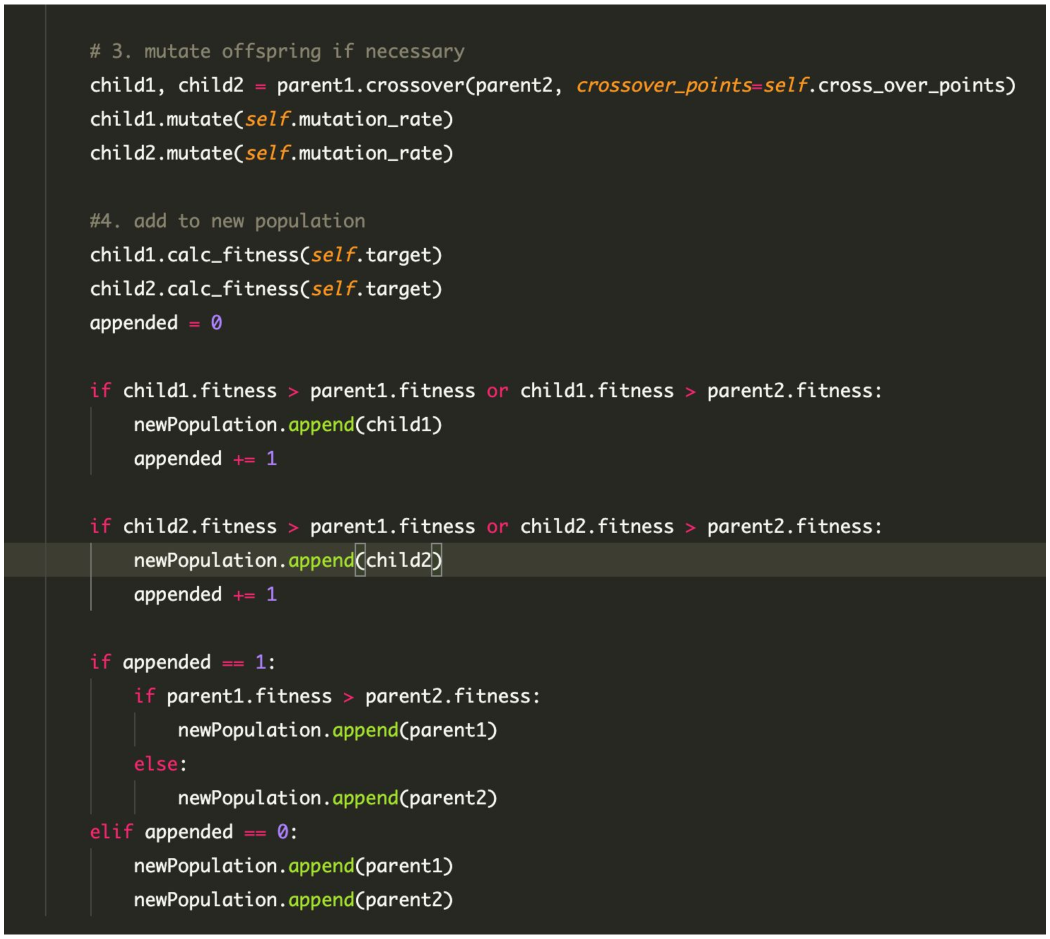
```
for i in range(self.originalPopulationSize):
    individual = self.population[i]
    appendTimes = int(round(( individual.fitness / fitnessSum ) * self.populationSize))

    for j in range(appendTimes):
        self.mating_pool.append(i)
        if len(self.mating_pool) == len(self.population):
            break

    if len(self.mating_pool) == len(self.population):
        break
```

5. New population method

This method generates a new population from the mating pool. To do this it randomly selects two members to mate and from those it creates two childs using crossover and mutation. It then compares childs and parents and selects the best two and appends them to the new population. This elitism makes sure that the best individuals move on and are only replaced by their childs if these are better (higher fitness). Also by generating two childs per mating with 2-point crossover I accomplish more diversity in half the number of iterations since for every population the number of mating encounters will be population / 2.



```python
# 3. mutate offspring if necessary
child1, child2 = parent1.crossover(parent2, crossover_points=self.cross_over_points)
child1.mutate(self.mutation_rate)
child2.mutate(self.mutation_rate)

#4. add to new population
child1.calc_fitness(self.target)
child2.calc_fitness(self.target)
appended = 0

if child1.fitness > parent1.fitness or child1.fitness > parent2.fitness:
    newPopulation.append(child1)
    appended += 1

if child2.fitness > parent1.fitness or child2.fitness > parent2.fitness:
    newPopulation.append(child2)
    appended += 1

if appended == 1:
    if parent1.fitness > parent2.fitness:
        newPopulation.append(parent1)
    else:
        newPopulation.append(parent2)
elif appended == 0:
    newPopulation.append(parent1)
    newPopulation.append(parent2)
```

6. Evaluation method

Finally, to evaluate the algorithm, the population, it's best individual and how close it is getting to the solution, I check every individual in the population to see if there is an individual whose fitness is 1. This method keeps track of the best individual and also the average fitness of the population. Both things are done here to avoid iterating twice over the population.

7. Main method:

I added some extra lines to print a summary and to track the execution time. I also change the main method to run 20 trials of a setting and to run multiple configurations in a single run and plot the outcome (see last question in point 3).

Part 2. Running the GA implemented

Results

Settings	Population size	Mutation rate	Average # of generations	Population fitness average (calculated based on the last generation)	Avg execution time (seconds)
1	200	0.01	4208.75	0.947631578947367	25.936353838443758
2	200	0.02	2106.8	0.947631578947367	13.790352916717529
3	100	0.01	7570.0	0.9478947368421039	23.035631048679353
4	10000	0.01	47.2	0.9245586842105361	11.82737169265747

Part 3. Analysis

o **What happens if you increase the mutation rate (Setting 2)? Does the algorithm converge faster or slower than Setting 1? Explain why.**

The algorithm converges in half the time and therefore faster. This prevents it from getting stuck in a local maxima. By duplicating the mutation rate we have now twice the probability of mutating a character in the genes of an individual. In other words when selecting a random number between 1 and 0 we now have twice the amount of space in which our random number can effectively trigger a mutation.

o What happens when you *decrease* the size of the population (Setting 3)?

When we decrease the size of the population the number of generations increase by 80% but the execution time remains almost the same. The increase in generations makes sense since now the algorithm has less individuals to work with, to mate , to combine, mutate and so on. Therefore less diversity is generated in each generation. The execution time doesn't increase much since we now do loops or iterations and comparisons for half the amount of individuals.

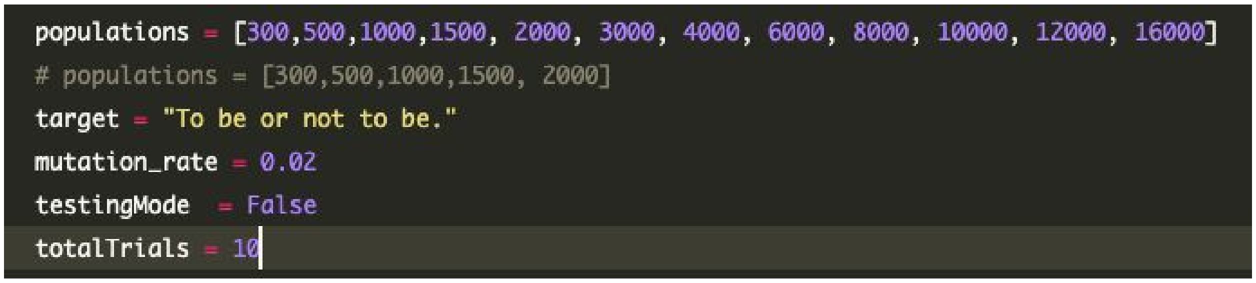
- **What happens when you increase the size of the population (Setting 4)?**

When we increase the size of the population the number of generations needed to find our target string decreases drastically. Actually by increasing the population by two digits the algorithm reduces the number of generations also by two digits. This makes sense since the diversity in the population increased and the number of new fitter individuals generated through mating, crossover, elitism and mutation increases in each generation.

- **How does the size of the population impact the computational cost to process (generate and evaluate) the population?**

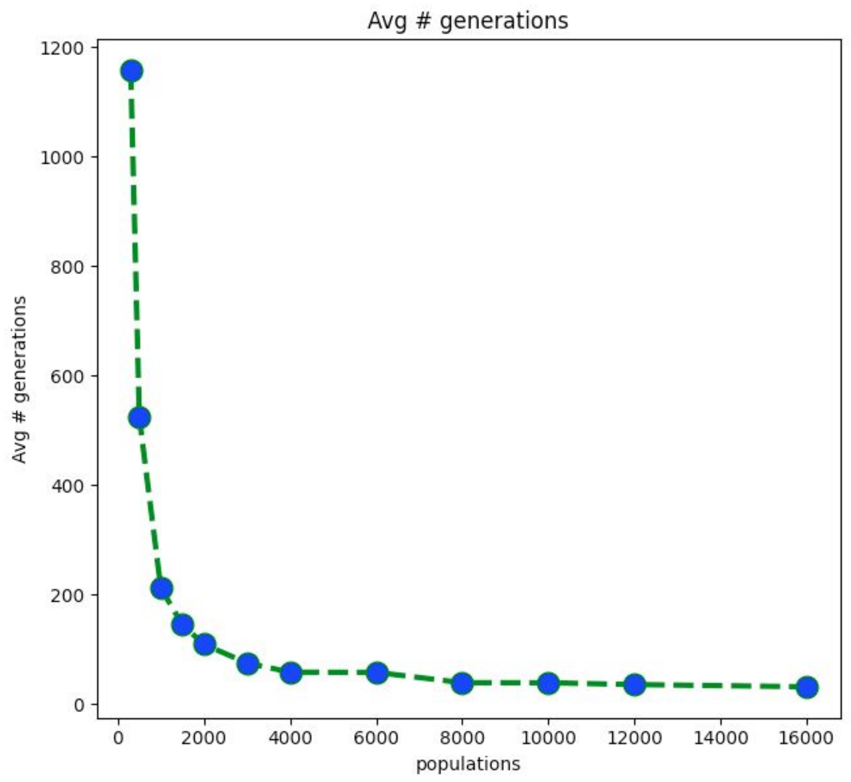
It could say that the computational cost (timewise) increases linearly due to the operations present in each method of my implementation. Most of these methods have loops that iterate once over the list of individuals and once over the genes to calculate the fitness of each individual (O(n*m) n being population size and m number of genes). However the real computational cost decreases because the increase in population size gives the algorithm more individuals to work with and more chances of finding fitter individuals and genes that match our target string. **However there is a point where everything changes and where the size of the population the computations matter and the execution time increases linearly.**

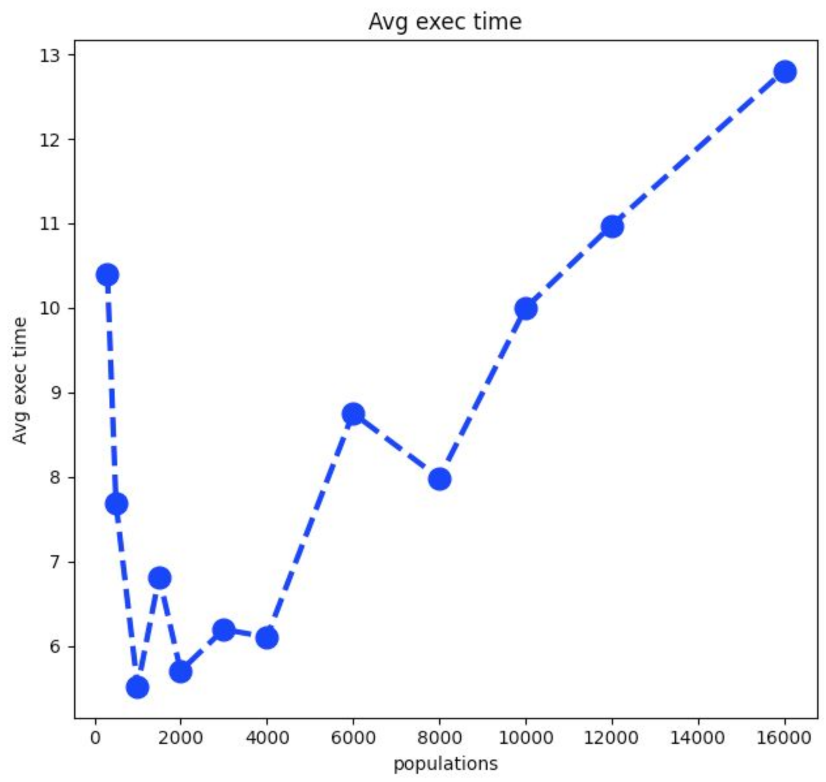
To test this I made the following experiment : 12 populations sizes, 10 repetitions (more would be too much for my current machine) and mutation rate of 0.02.



```
populations = [300,500,1000,1500, 2000, 3000, 4000, 6000, 8000, 10000, 12000, 16000]
# populations = [300,500,1000,1500, 2000]
target = "To be or not to be."
mutation_rate = 0.02
testingMode  = False
totalTrials = 10
```

These are the results of such an experiment:


Avg # generations
Avg # generations
1200
1000
800
600
400
200
0
0
2000
4000
6000
8000
10000
12000
14000
16000
populations


Avg exec time
Avg exec time
13
12
11
10
9
8
7
6
0
2000
4000
6000
8000
10000
12000
14000
16000
populations