CT421 Project 1: Evolutionary Search

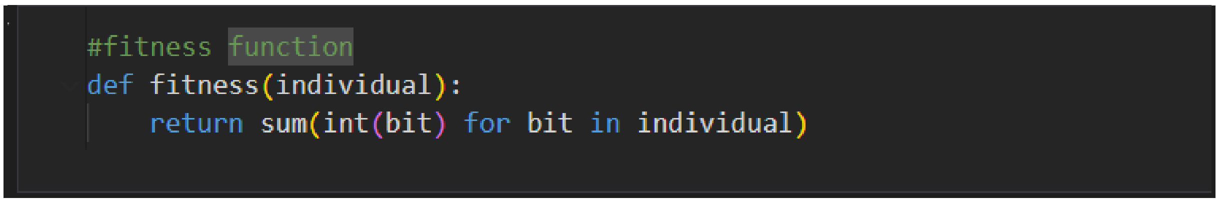
Kamil Klimczak [19382973] && Alessio Musciagna [17105884]

Contributions: Equal split of work, Alessio and Kamil have both worked on separate implementations for part A and eventually decided to go with Alessio's approach that can be seen on the github. Part B was developed through pair programming sessions over several days both in person and screen share sessions where we have both contributed to the final solution. The final report was concatenated by Kamil while Alessio gathered graphs and results for the report.

https://github.com/kkcavan/AIassignment1/tree/master

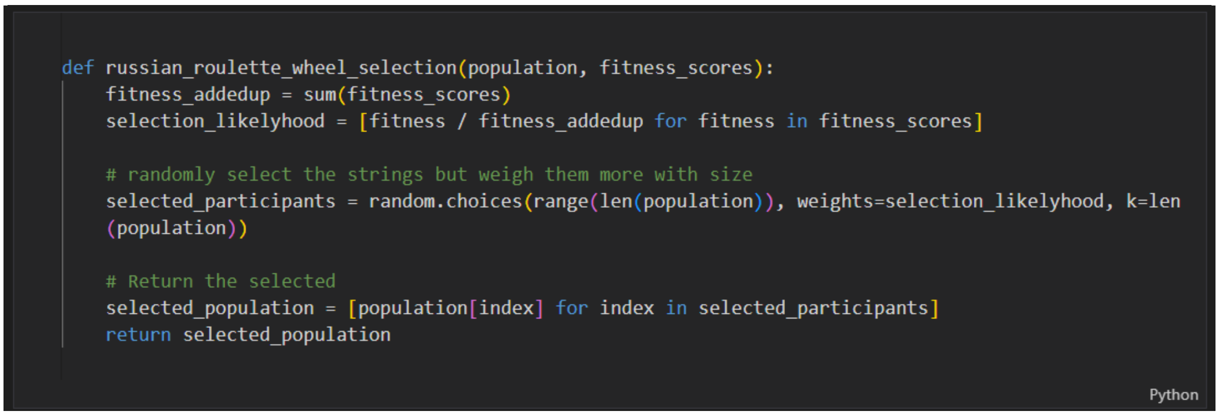
1.1

1. *Description of your representation; fitness function; selection, crossover, mutation, and any other operations. [2]*



```python
#fitness function
def fitness(individual):
    return sum(int(bit) for bit in individual)
```

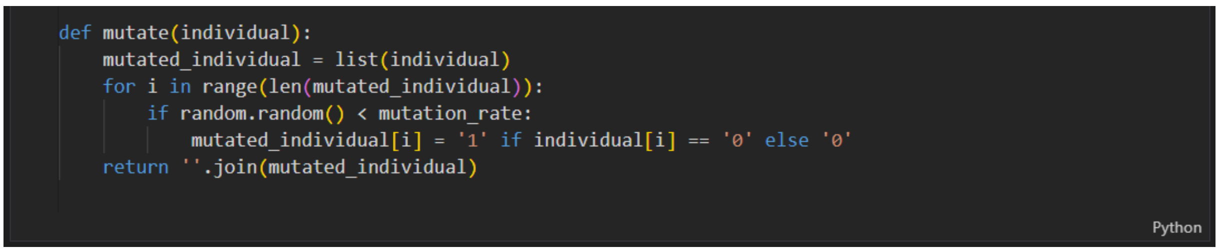
Simply summing each bit in a string to get its fitness out of 0-30.



```python
def russian_roulette_wheel_selection(population, fitness_scores):
    fitness_addedup = sum(fitness_scores)
    selection_likelyhood = [fitness / fitness_addedup for fitness in fitness_scores]

    # randomly select the strings but weigh them more with size
    selected_participants = random.choices(range(len(population)), weights=selection_likelyhood, k=len
    (population))

    # Return the selected
    selected_population = [population[index] for index in selected_participants]
    return selected_population
```

Python

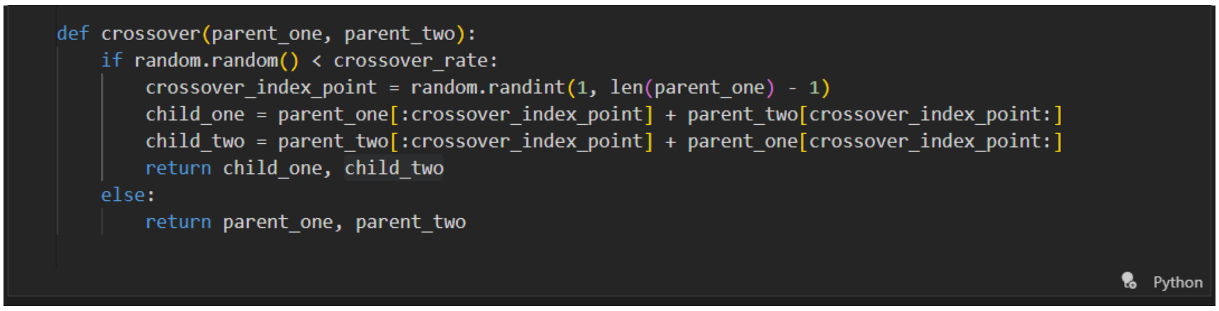
A Roulette Wheel Selection procedure that can be used within a GA algorithm. The way this procedure works can be pictured as a roulette wheel with the size of each slice being represented by the fitness of that string.[3]



```python
def mutate(individual):
    mutated_individual = list(individual)
    for i in range(len(mutated_individual)):
        if random.random() < mutation_rate:
            mutated_individual[i] = '1' if individual[i] == '0' else '0'
    return ''.join(mutated_individual)
```

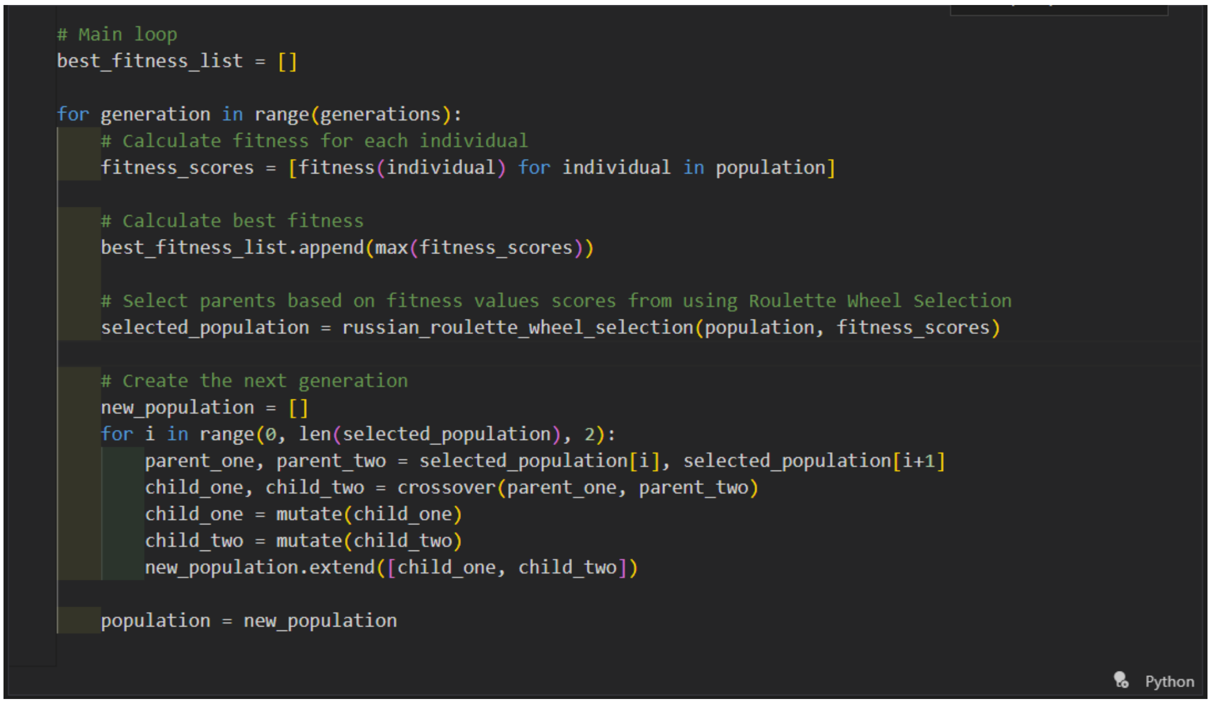
Python

Mutating function that will flip a bit based on a RNG. If the randomly generated number is smaller than the mutation rate the bit will flip causing a mutation.



```python
def crossover(parent_one, parent_two):
    if random.random() < crossover_rate:
        crossover_index_point = random.randint(1, len(parent_one) - 1)
        child_one = parent_one[:crossover_index_point] + parent_two[crossover_index_point:]
        child_two = parent_two[:crossover_index_point] + parent_one[crossover_index_point:]
        return child_one, child_two
    else:
        return parent_one, parent_two
```

A crossover genetic operator. It is used in this GA to combine “genetic” material from two parents. The rate at which this crossover happens depends on the crossover_rate. This is a likelihood of the crossover happening. If it happens that the RNG is smaller than the crossover rate, a point is randomly selected in the string. The offspring will be created and will get the traits of one parent up to the point in the crossover string and the rest will come from the other parent. Otherwise if the rng did not cause a cross over we will return the parents without any modifications.[5]



```python
# Main loop
best_fitness_list = []

for generation in range(generations):
    # Calculate fitness for each individual
    fitness_scores = [fitness(individual) for individual in population]

    # Calculate best fitness
    best_fitness_list.append(max(fitness_scores))

    # Select parents based on fitness values scores from using Roulette Wheel Selection
    selected_population = russian_roulette_wheel_selection(population, fitness_scores)

    # Create the next generation
    new_population = []
    for i in range(0, len(selected_population), 2):
        parent_one, parent_two = selected_population[i], selected_population[i+1]
        child_one, child_two = crossover(parent_one, parent_two)
        child_one = mutate(child_one)
        child_two = mutate(child_two)
        new_population.extend([child_one, child_two])

    population = new_population
```

This is where the core of the GA happens and everything comes together. The GA will run over a predefined number of epochs where a generation will be created. The fitness function is employed to calculate the fitness of each individual string.
The fittest individual strings are found through the max function. This is a representation of the strongest in the herd (almost). We will append these to the most fit list.

Following this we select a number of parents through the roulette wheel, where we hope to mainly land on the fitter individuals and select them as parents.
The selected parents will go through a process of crossover and mutation operations to generate a new and hopefully stronger offspring for the next population.
The offspring will be a combination of genes through crossover followed by a chance of mutation. The offspring then enter the new population.
The old population is replaced with the new population and the cycle can repeat.

In general, high mutation will increase exploration, and high crossover will increase exploitation. So, this means that exploration will come up with new solutions to the problem that have not been seen by the GA before. Whereas exploitation will take the currents solutions and take advantage of solutions.

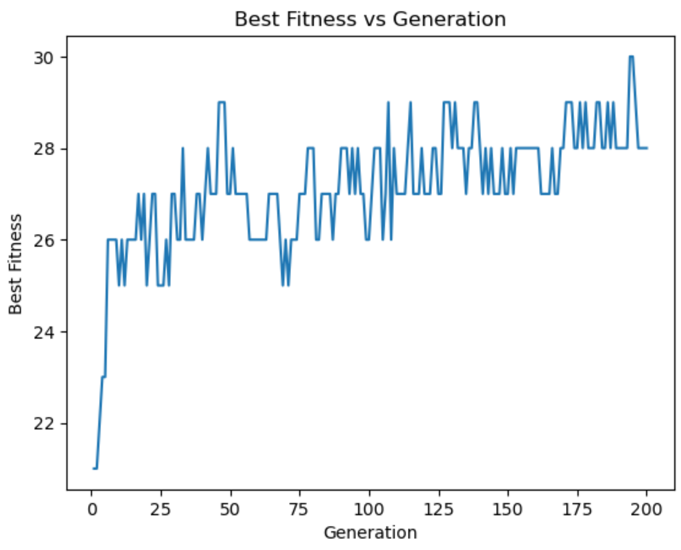


Figure 1: mutation_rate = 0.01 crossover_rate = 0.8. We reach a local optimum around the 50th generation. Due to the mutation rate we introduce novelty into the solution space which caused a decrease in fitness as seen by the low valley at generation 75. Eventually we reach the highest fitness level at generation ~190.

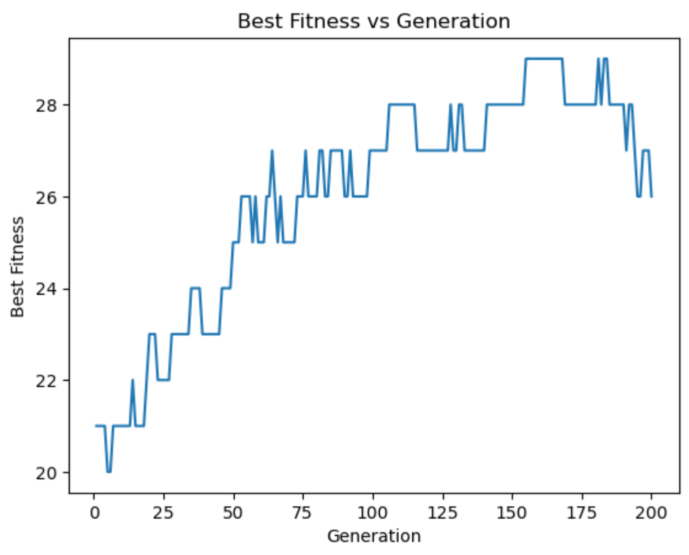


Figure 2: mutation_rate = 0.01 crossover_rate = 0.01.

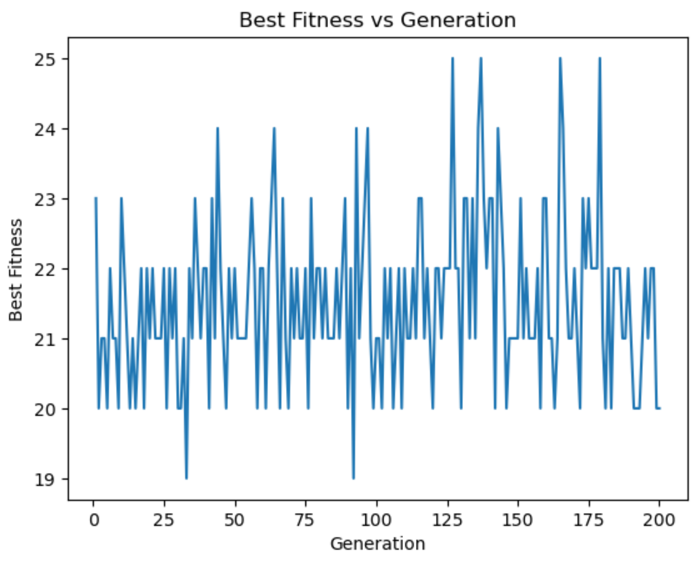


Figure 3: mutation_rate = 0.8 crossover_rate = 0.01

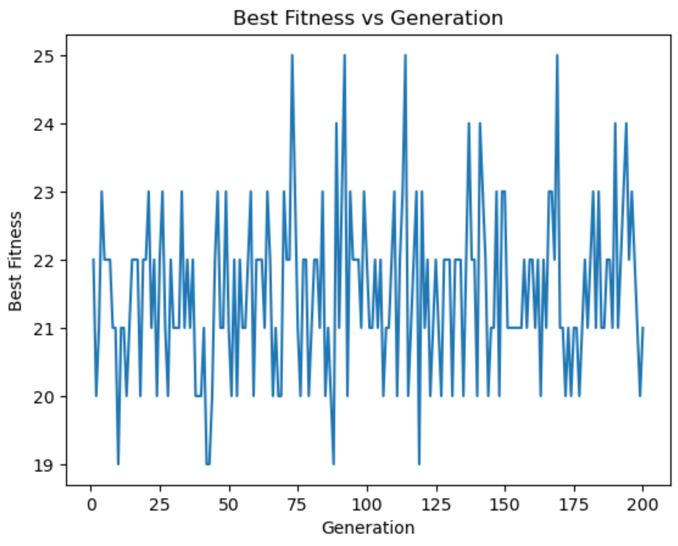


Figure 4: mutation_rate = 0.8 crossover_rate = 0.8

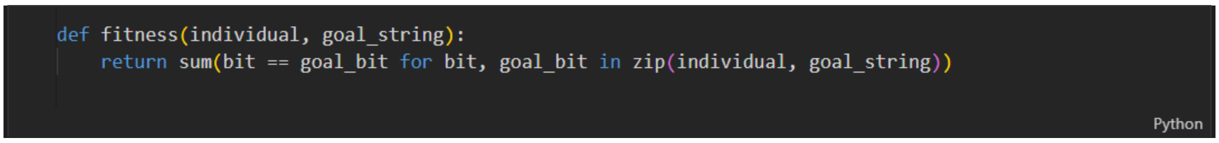
Results: Through trial and testing we have found that the combination of a low mutation rate and a high crossover rate has led to the best performance in this GA problem. This problem solution changes greatly when tweaking the mutation and crossover variables as seen in figure 2,3,4. High mutation rate leads to a very spikey solution with lots of peaks and valleys that indicate that the algorithm has a hard time ending in a plateau.

1.2

2.

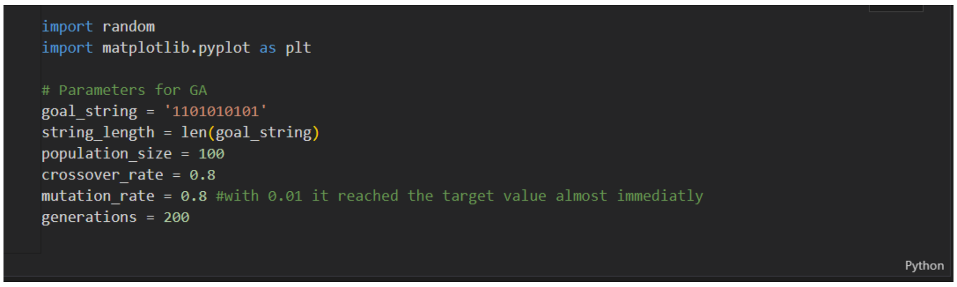
The following snippet have been altered to the new problem; the rest of the problem has been kept the same.

Fitness Function for evolving to a target string,



```python
def fitness(individual, goal_string):
    return sum(bit == goal_bit for bit, goal_bit in zip(individual, goal_string))
```

A modified version of the initial fitness function. This function will gain favourable conditions by matching bit by bit of individuals to our defined ideal string, which is a 30 bit long binary string. In this case the location of the bits matters.



```
import random
import matplotlib.pyplot as plt

# Parameters for GA
goal_string = '1101010101'
string_length = len(goal_string)
population_size = 100
crossover_rate = 0.8
mutation_rate = 0.8 #with 0.01 it reached the target value almost immediatly
generations = 200
```

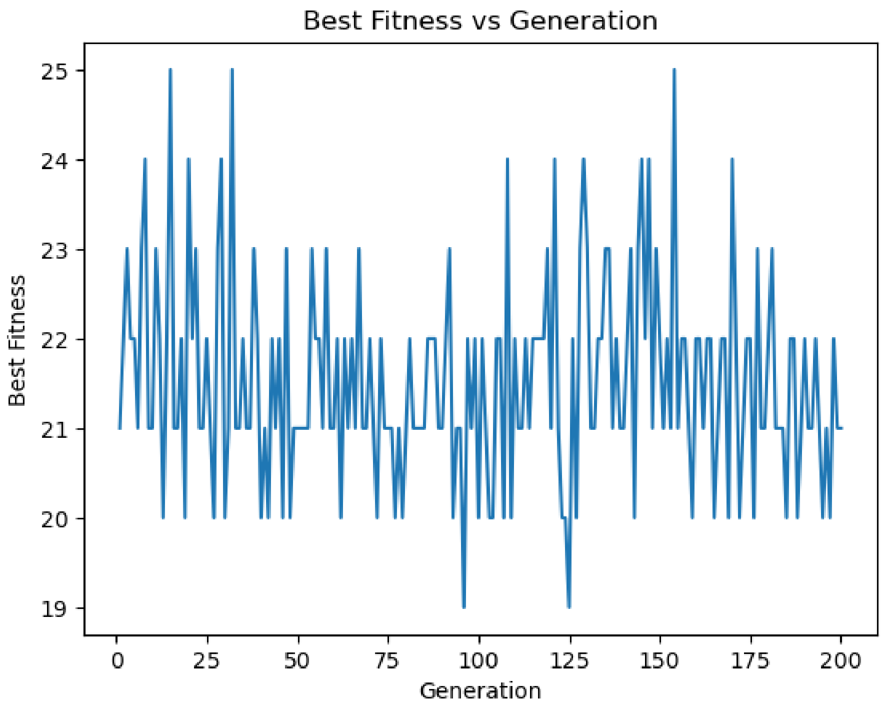


Figure 5: crossover_rate = 0.8 mutation_rate = 0.8 population=100

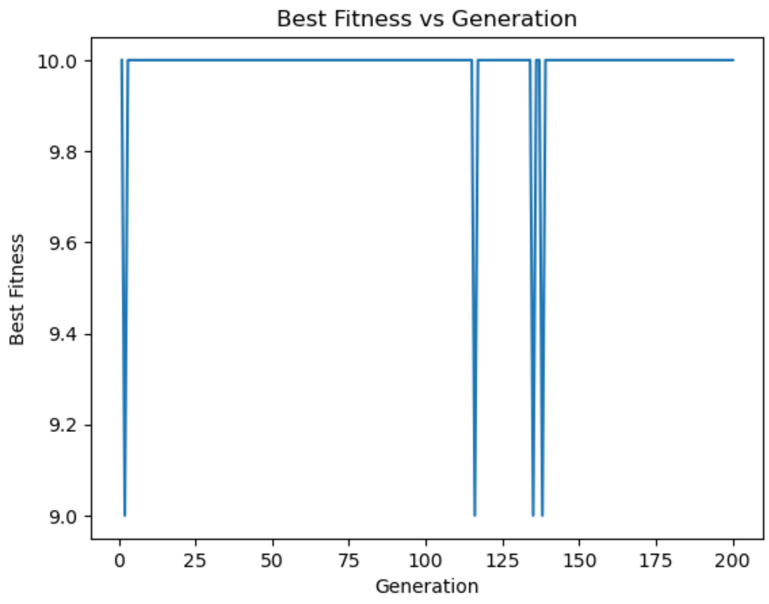


Figure 6: crossover_rate = 0.1 mutation_rate=0.8 Population = 500

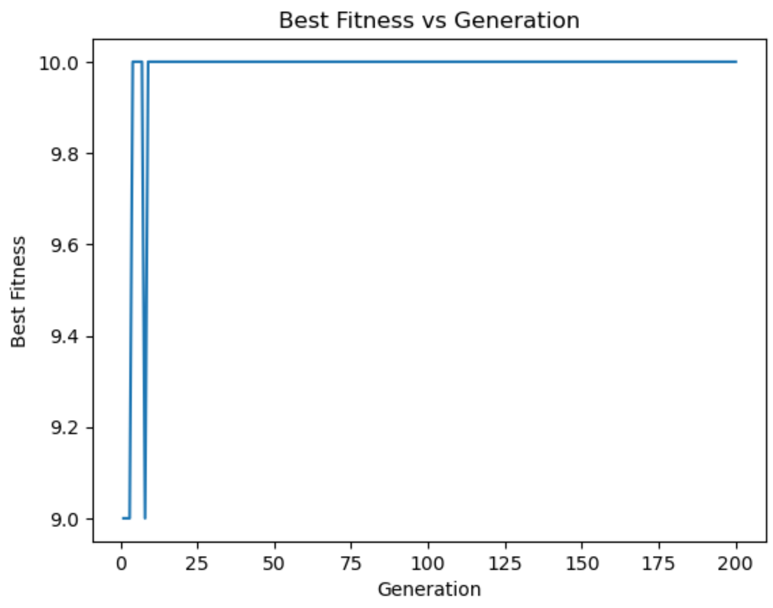


Figure 7:crossover_rate = 0.8 mutation_rate=0.01 Population = 100

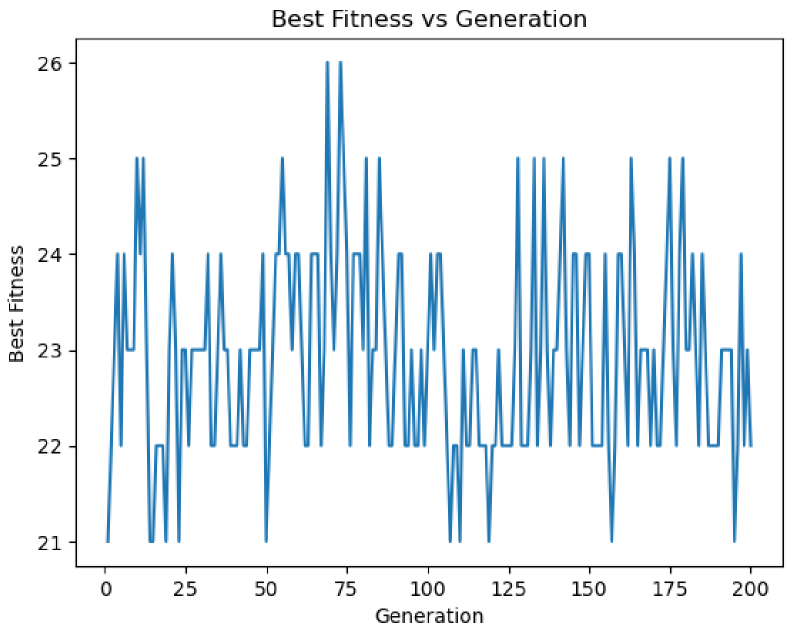


Figure 8: crossover_rate = 0.8 mutation_rate = 0.1 population=100

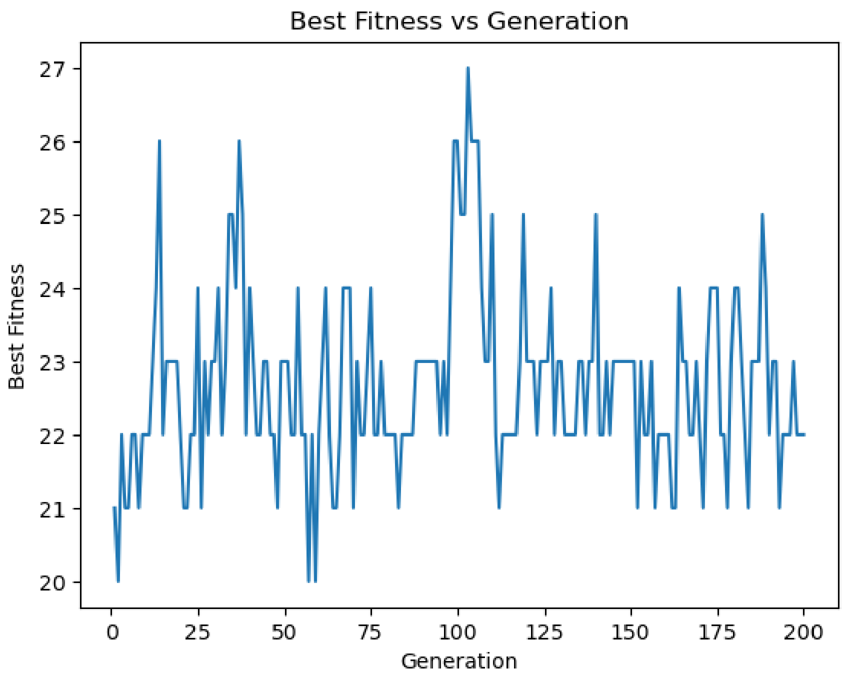
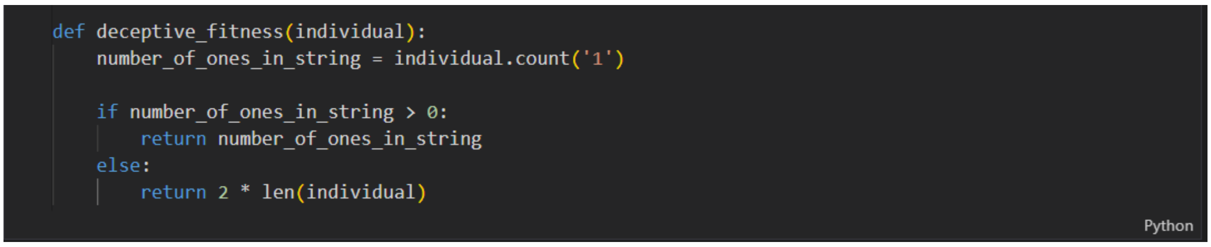


Figure 9: mutation_rate = 0.1 crossover_rate = 0.1 population=100

Results: We can conclude that mutation rate and population has a correlation from figure 6 and figure 7. A higher mutation rate will lead similar results as having a larger population. As we decrease population and increase mutation rate we will end up with a similar scenario. As stated in the previous section 1.1, high mutation rates with low population greatly increase exploration leading to a spikey graph in figure 8 and figure 9.

1.3

3. The following snippet have been altered to the new problem; the rest of the problem has been kept the same.



```python
def deceptive_fitness(individual):
    number_of_ones_in_string = individual.count('1')

    if number_of_ones_in_string > 0:
        return number_of_ones_in_string
    else:
        return 2 * len(individual)
```

Here we have the deceptive landscape problem where the fitness function is equal to the number of 1's in an individual, otherwise it is 2x the length of the string. This deceptive fitness function tricks the algorithm into thinking it is on the right path to a good solution.

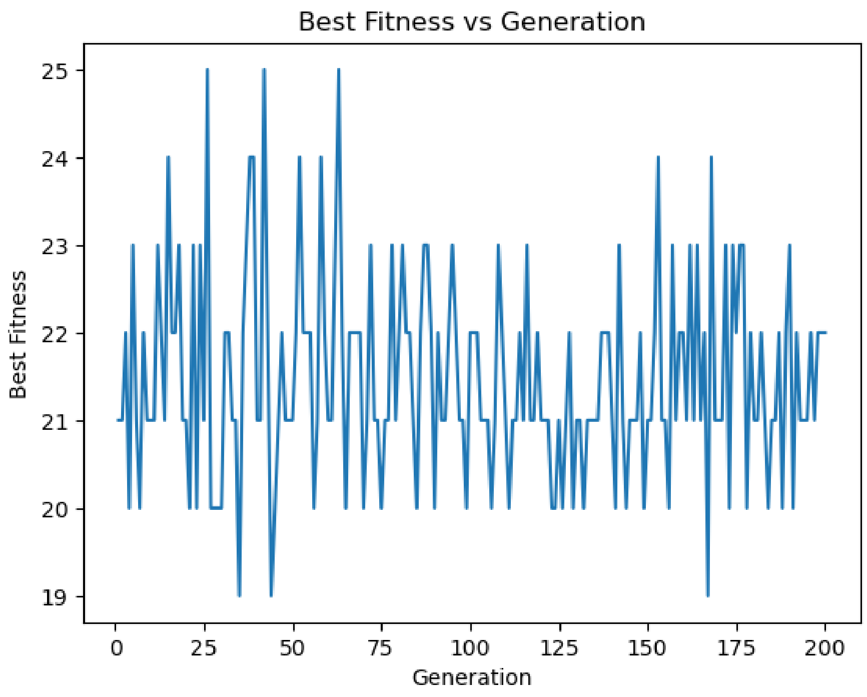


Figure 10: mutation_rate = 0.8 corssover_rate = 0.8

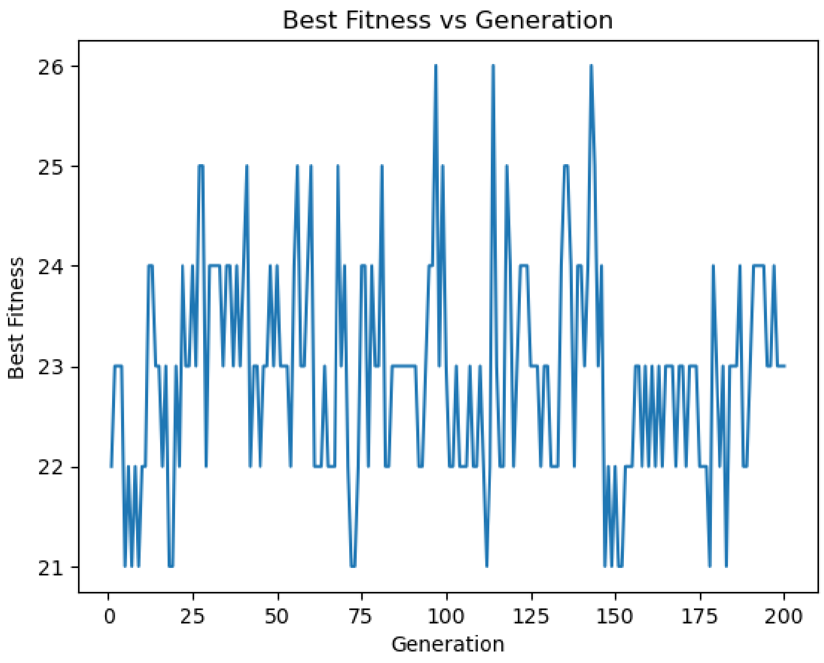


Figure 11: mutation_rate = 0.1 corssover_rate = 0.8

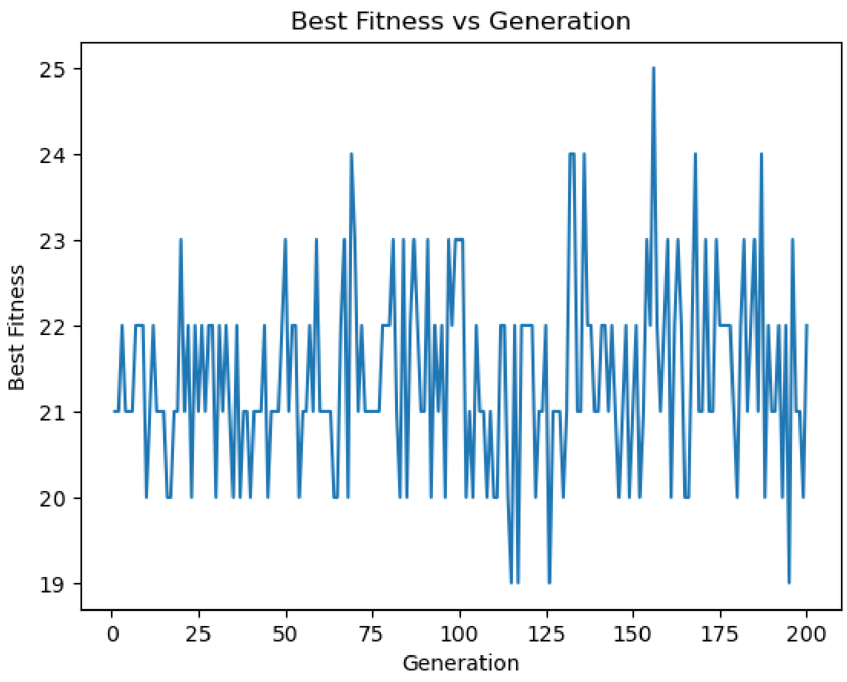


Figure 12: mutation_rate = 0.8 corssover_rate = 0.1

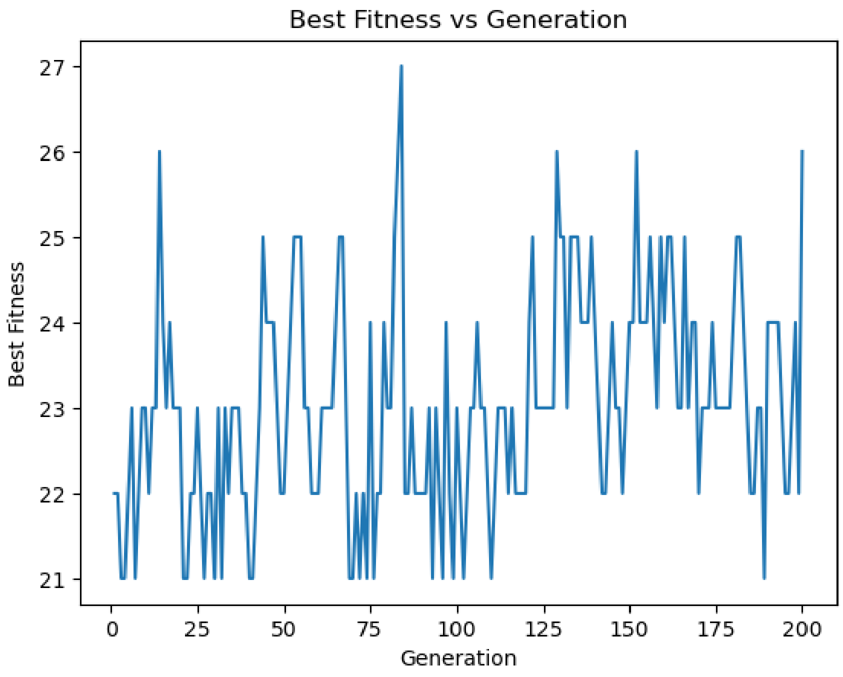


Figure 13: mutation_rate = 0.1 corssover_rate = 0.1

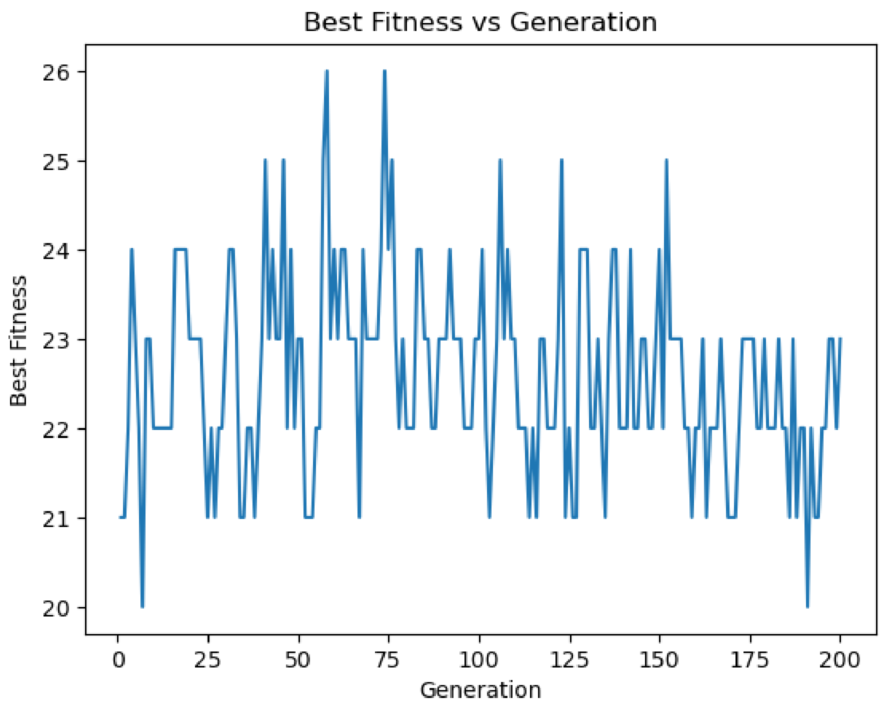


Figure 14: crossover_rate = 0.01 mutation_rate=0.8 Population = 100

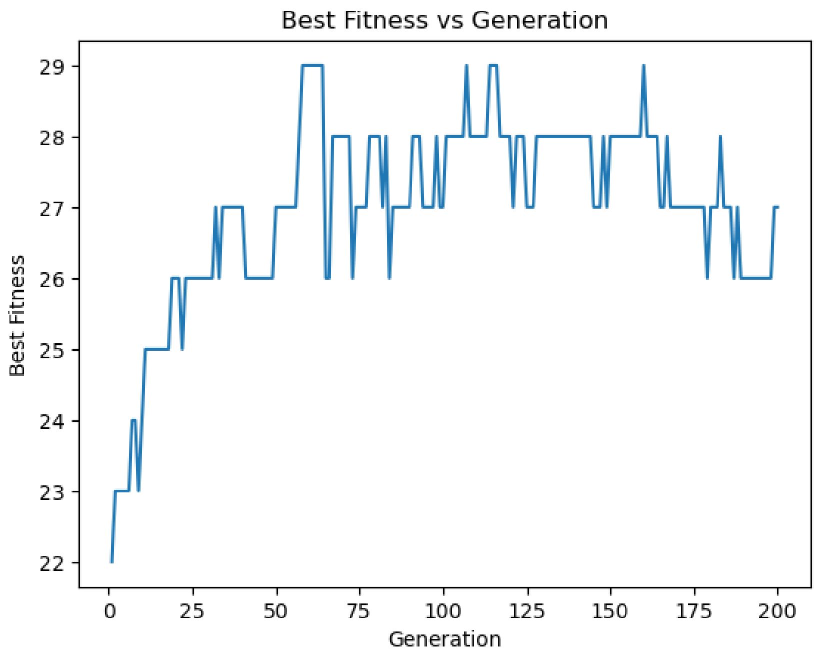


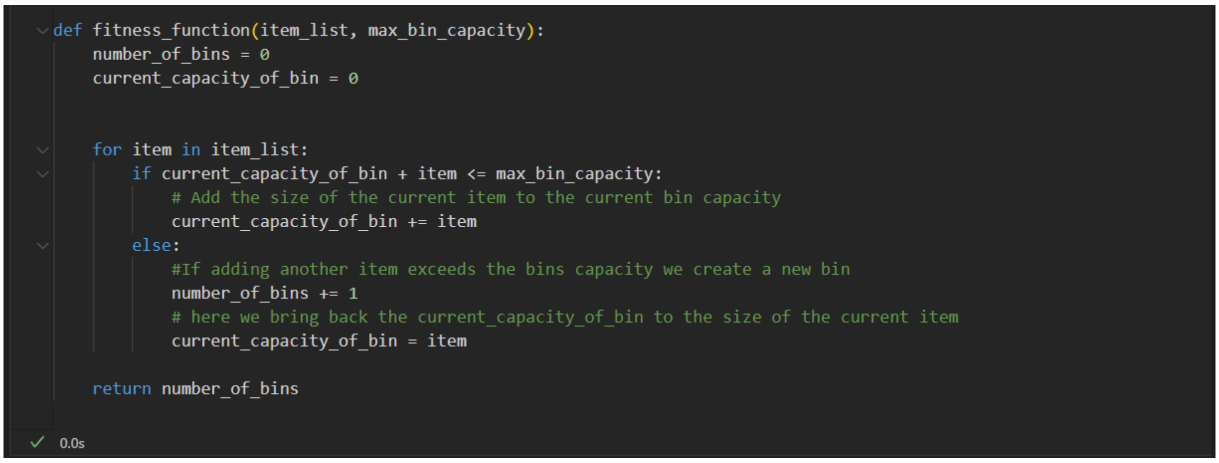
Figure 15: crossover_rate = 0.5 mutation_rate=0.01 Population = 100

Results: As we can see from this deceptive problem it is very hard for the GA to find a plateau scenario. The GA is tricked into believing that it is finding high fitness individuals. From the graphs we see that even though we find peaks , we do not stay in a high fitness solution for very long.

Part B:

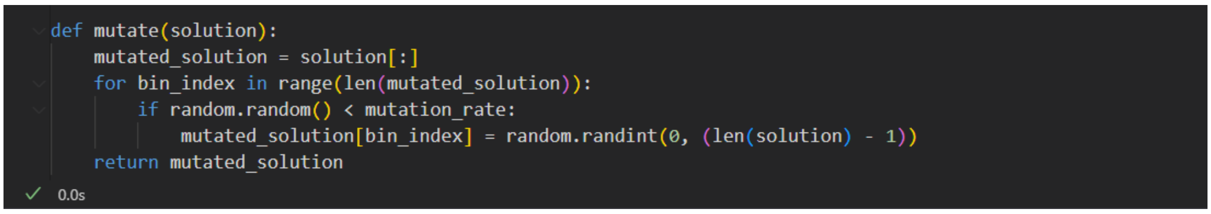
Insight into problem landscape: computationally the bin packing problem is known to be np-hard so by this we know that is in a class of very computationally challenging problems in which there is no efficient algorithm that can solve all instances optimally. This is due to the sheer number of combinations that need to be considered. Exact solutions are impractical for such a problem. Thus a way of finding a solution for a problem in which capacities of bins and number of items change, we tend to use genetic algorithms to find viable solutions.[1]

Initially to load the dataset into a jupyter notebook we have split the dataset text file into separate files for each binning problem and converted them to csv files. Using the pandas library we read in each file and create a dataframe out of them, renaming the columns to Weight and num_items respectively. We then flatten the dataframe into one column by multiplying the weight by the num of items.



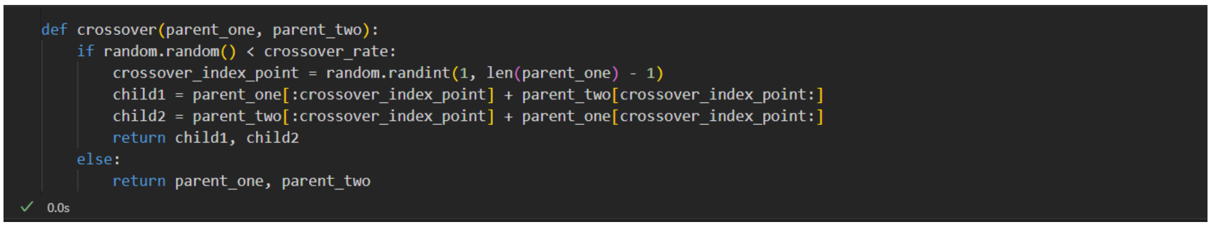
```python
def fitness_function(item_list, max_bin_capacity):
    number_of_bins = 0
    current_capacity_of_bin = 0


    for item in item_list:
        if current_capacity_of_bin + item <= max_bin_capacity:
            # Add the size of the current item to the current bin capacity
            current_capacity_of_bin += item
        else:
            #If adding another item exceeds the bins capacity we create a new bin
            number_of_bins += 1
            # here we bring back the current_capacity_of_bin to the size of the current item
            current_capacity_of_bin = item

    return number_of_bins
```
✓ 0.0s

The fitness function: we have defined num_bin and current_capacity. We iterate though each item (weight) in a list of all items. If we add an items weight to the current capacity of a bin and it fits, we add that item to the capacity. Otherwise, we will create a new bin and set the capacity of the bin to the new item.



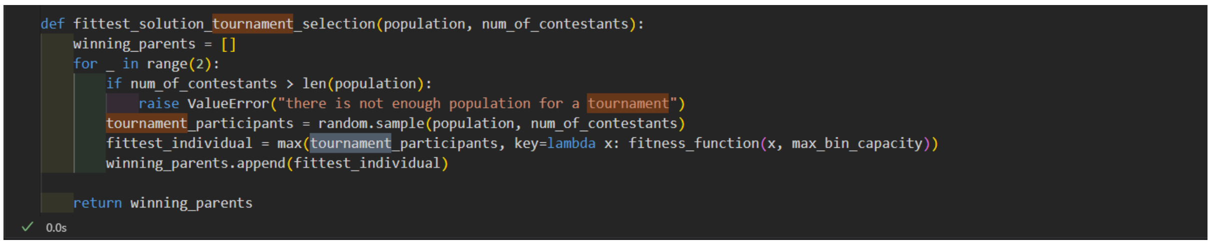
```python
def mutate(solution):
    mutated_solution = solution[:]
    for bin_index in range(len(mutated_solution)):
        if random.random() < mutation_rate:
            mutated_solution[bin_index] = random.randint(0, (len(solution) - 1))
    return mutated_solution
```
✓ 0.0s

The mutate function works similarly to the previous problems. If the mutation rate criterion is met, the gene index is mutated to a random index, which in this context means placing an item into a different bin.



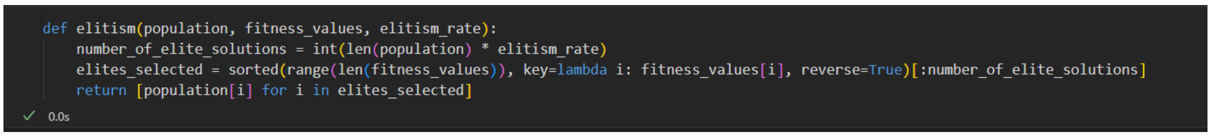
```python
def crossover(parent_one, parent_two):
    if random.random() < crossover_rate:
        crossover_index_point = random.randint(1, len(parent_one) - 1)
        child1 = parent_one[:crossover_index_point] + parent_two[crossover_index_point:]
        child2 = parent_two[:crossover_index_point] + parent_one[crossover_index_point:]
        return child1, child2
    else:
        return parent_one, parent_two
```
✓ 0.0s

A function that is responsible for creating new offspring solutions through combining genetic material from two parents. It works on the same principle as the above problems, we will find a random index in a list and replace it to that point with one parent and the rest with the other parents. So, we are basically combining two solutions by splitting them at some point and giving each child a random proportion of its parent's solution.[4]



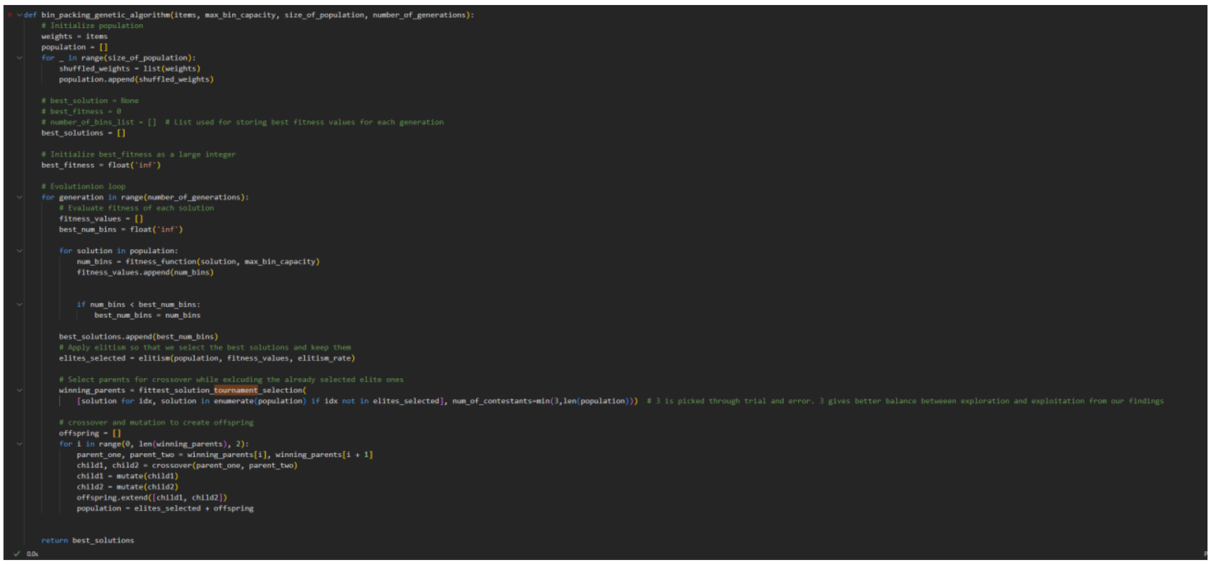
```
def fittest_solution_tournament_selection(population, num_of_contestants):
    winning_parents = []
    for _ in range(2):
        if num_of_contestants > len(population):
            raise ValueError("there is not enough population for a tournament")
        tournament_participants = random.sample(population, num_of_contestants)
        fittest_individual = max(tournament_participants, key=lambda x: fitness_function(x, max_bin_capacity))
        winning_parents.append(fittest_individual)

    return winning_parents
✓ 0.0s
```

We perform a selection process two times to find 2 parents for a crossover. We then randomly sample 2 individual solutions by rng(twice). These 2 solutions will "fight". Leading to the strong individual being selected, This means that most full bins will win and then since we run this twice, we will have 2 parents selected for a crossover.



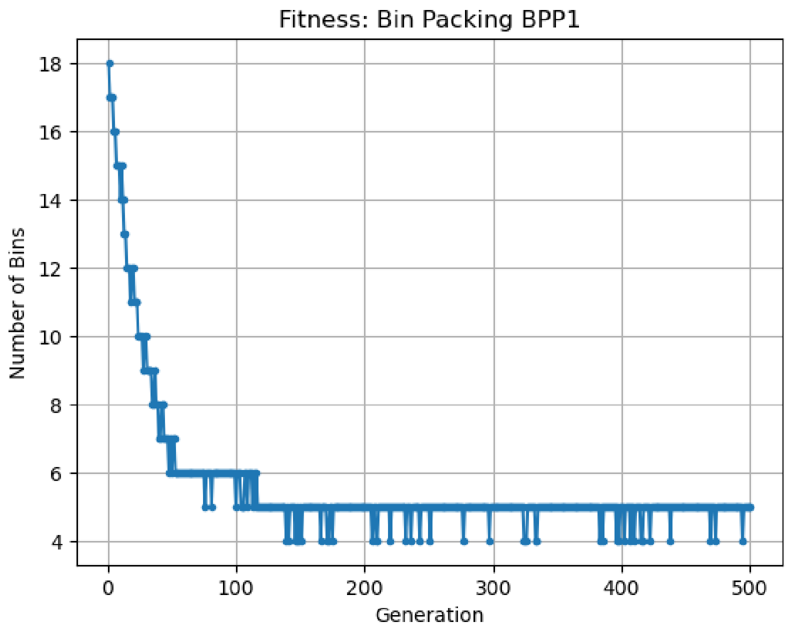
```
def elitism(population, fitness_values, elitism_rate):
    number_of_elite_solutions = int(len(population) * elitism_rate)
    elites_selected = sorted(range(len(fitness_values)), key=lambda i: fitness_values[i], reverse=True)[:number_of_elite_solutions]
    return [population[i] for i in elites_selected]
✓ 0.0s
```

Our solution implementation contains elitism too. This is to carry good generations forward with no change. This will assist us in carrying over good solutions. The function ranks the fitness values of solutions and preserves a list of elite solutions.



```
def bin_packing_genetic_algorithm(items, max_bin_capacity, size_of_population, number_of_generations):
    # Initialize population
    weights = items
    population = []
    for _ in range(size_of_population):
        shuffled_weights = list(weights)
        population.append(shuffled_weights)

    # best_solution = None
    # best_fitness = 0
    # number_of_bins_list = []  # List used for storing best fitness values for each generation
    best_solutions = []

    # Initialize best_fitness as a large integer
    best_fitness = float('inf')

    # Evolutionion loop
    for generation in range(number_of_generations):
        # Evaluate fitness of each solution
        fitness_values = []
        best_num_bins = float('inf')

        for solution in population:
            num_bins = fitness_function(solution, max_bin_capacity)
            fitness_values.append(num_bins)


            if num_bins < best_num_bins:
                best_num_bins = num_bins

        best_solutions.append(best_num_bins)
        # Apply elitism so that we select the best solutions and keep them
        elites_selected = elitism(population, fitness_values, elitism_rate)

        # Select parents for crossover while exlcuding the already selected elite ones
        winning_parents = fittest_solution_tournament_selection(
            [solution for idx, solution in enumerate(population) if idx not in elites_selected], num_of_contestants=min(3,len(population)))  # 3 is picked through trial and error. 3 gives better balance betweeen exploration and exploitation from our findings

        # crossover and mutation to create offspring
        offspring = []
        for i in range(0, len(winning_parents), 2):
            parent_one, parent_two = winning_parents[i], winning_parents[i + 1]
            child1, child2 = crossover(parent_one, parent_two)
            child1 = mutate(child1)
            child2 = mutate(child2)
            offspring.extend([child1, child2])
            population = elites_selected + offspring


    return best_solutions
✓ 0.0s
```

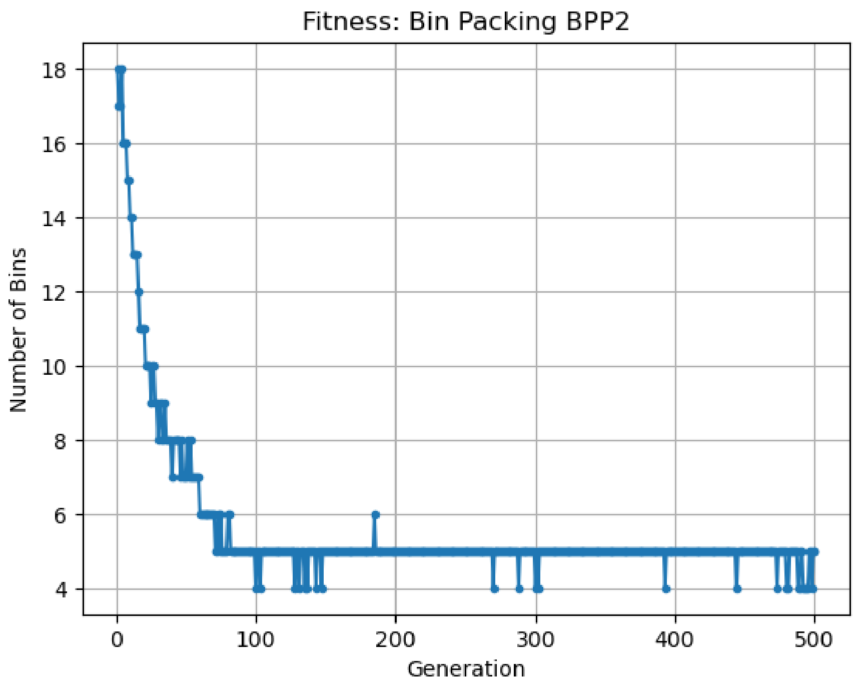
The main function of our genetic algorithm as a function. We can see in what order the GA runs. We start packing one item per bin so we'll have the max number of bins. That's why the high spike at the beginning of the graph. Then we enforce elitism mutation and crossover in order to have a better packing. That's why the decreasing curve. Eventually it will find the local optima then it will go down. This happens from generation 100 onwards.



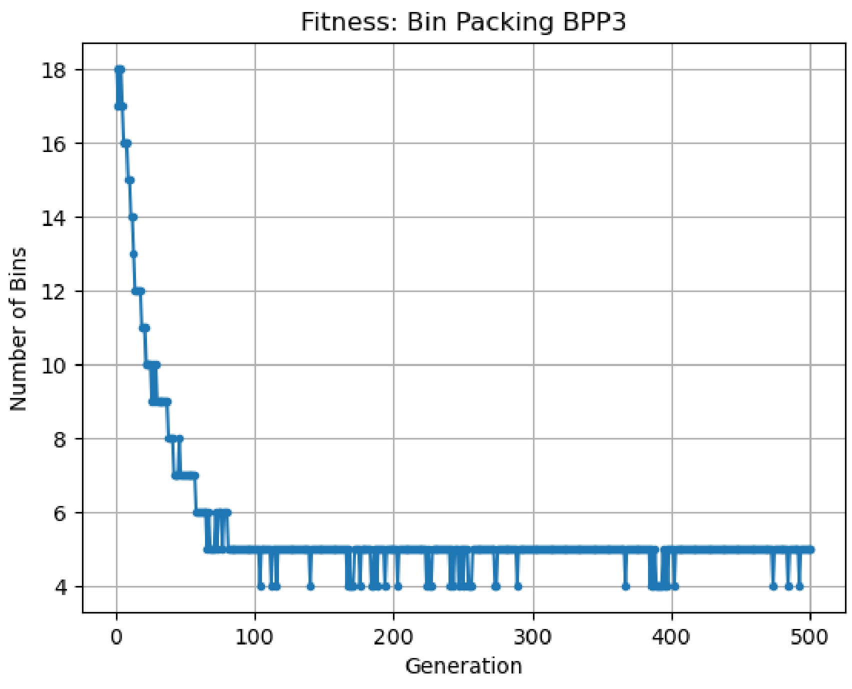
bpp1, 500, [18, 17, 17, 17, 16, 16, 15, 15, 15, 14, 15, 14, 13, 13, 12, 12, 12, 11, 12, 12, 11, 11, 11, 10, 10, 10, 10, 9, 10, 10, 9, 9, 9, 9, 8, 8, 9, 8, 8, 7, 7, 8, 8, 7, 7, 7, 7, 6, 7, 6, 7, 7, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6, 5, 5, 5, 6, 6, 5, 6, 6, 6, 5, 6, 5, 6, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 4, 5, 5, 5, 5, 4, 5, 4, 4, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 4, 4, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 5, 5, 5, 5, 5, 5, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 4, 5, 5, 5, 4, 5, 5, 5, 5, 4, 5, 5, 4, 5, 5, 5, 5, 4, 4, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5]

4: min bins is the best solution we can achieve

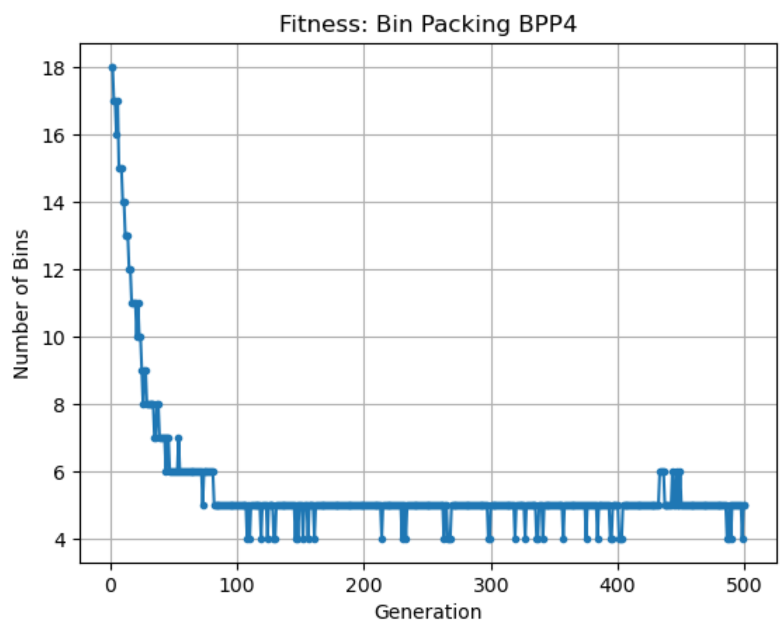
5.624 : average number of bins

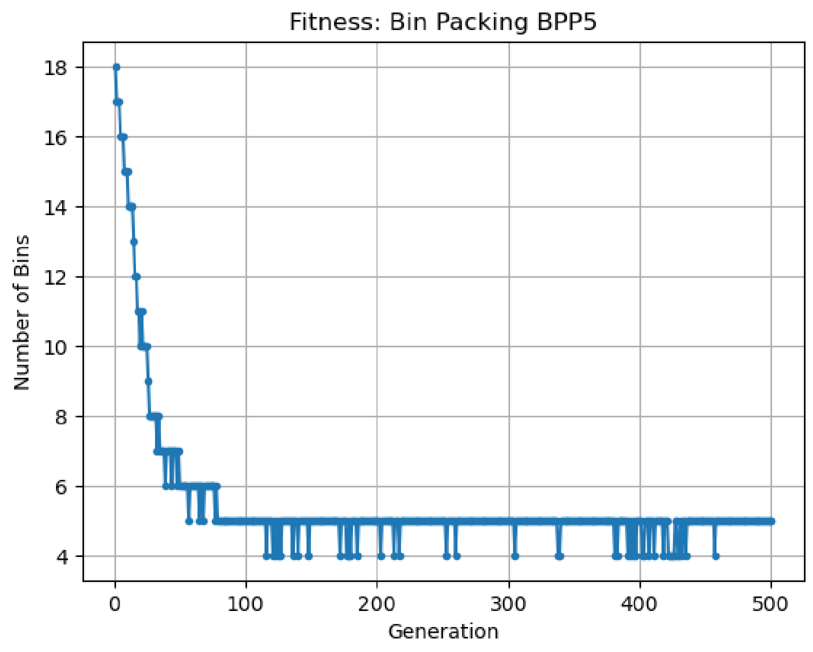


Fitness: Bin Packing BPP2
Number of Bins
Generation



Fitness: Bin Packing BPP3
Number of Bins
Generation





References:

1. https://en.wikipedia.org/wiki/Genetic_algorithm
2. https://www.geeksforgeeks.org/genetic-algorithms/
3. https://dev.to/rpalo/python-s-random-choices-is-awesome-46ii
4. https://www.geeksforgeeks.org/python-single-point-crossover-in-genetic-algorithm/
5. https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/