```python
#simple_genetic_algorithm.py

from typing import List
import random
from functions import general_decoder
import matplotlib.pyplot as plt

LENGTH_OF_DECIMAL=4

def initialize(pop_size: int, alnum_set: List[str], var_string_length) -> List[str]:
    """
    Generates pop_size of random strings with characters of their alphanumeric character set.

    Args:
        pop_size (int): Number of strings to be generated
        alnum_set (List[str]): Valid characters to genrate random string from
        var_string_length (int): Number of characters expected to be in each string
    Returns:
        (List[str]): List of pop_size random strings made with characters of their alphanumeric
                        character set
    """

    return ["".join(random.choices(alnum_set, k=var_string_length)) for _ in range(pop_size)]


def perform_reproduction(population, inverse_fitness_func) -> List:
    """
    Takes in a population of strings and probabalistically candidates for mating  based on the
    relative fitness of each string (i.e. the more fit members of the population will be picked more often
    and will therefore make up a bigger portion of the mating pool).

    Args:
        population (List[str]): List of strings that make up the mating pool
        inverse_fitness_func (<function>): Benchmark function with known global minima; returned values
                                                closer to zero mean the given arguments are closer to optimum
                                  Args:
                                      (str): Alphanumeric string representing number system value(s)
                                  Returns:
                                      (float): floating point value between 0 to 1; results closer to zero
                                                are closer to optimization
    Returns:
        (List[str]): List of strings that make up the new mating pool generation
    """

    #compile a list of floating point values, numbers closer to 0 have a higher fitness
    inverse_fitnesses = [inverse_fitness_func(s) for s in population]
    min_inv_fit = min(inverse_fitnesses)
    max_inv_fit = max(inverse_fitnesses)

    #compile a list of inverse fitness values that are normalized to [0,1] with regard to
    #the range of the population's fitness measures, numbers closer to 0 have a higher fitness
    inverse_fitnesses = [(X-min_inv_fit)/(max_inv_fit-min_inv_fit+1) for X in inverse_fitnesses]    #adding 1 to avoid division by zero

    #invert list of normalized inverse fitness values so that numbers closer to 1 have a higher fitness
```

```python
    fitnesses = [1 / ((s)+1) for s in inverse_fitnesses]

    #sum up all found fitness measures
    total_fitness = sum(fitnesses)

    #compile a list of each fitness measure's proportion to the sum of all found fitness measures
    #of the mating pool; to be used as a probability that the condidate will mate
    probabilities_of_reproduction = [fit / total_fitness for fit in fitnesses]

    #create list of randomly chosen strings that are weight biased
    return random.choices(population=population, weights=probabilities_of_reproduction, k=len(population))


def perform_mating(population):
    """
    Takes a list of strings, returns a list of strings after potential mating operations.

    Args:
        population (List[str]): List of strings that make up the mating pool
    Returns:
        (List[str]): List of strings that make up the now potentially modified mating pool
    """

    new_pop = []

    while (len(population) > 0):
        s1 = population.pop(random.randint(0, len(population) - 1))
        s2 = population.pop(random.randint(0, len(population) - 1))

        s1p, s2p = crossover_pair(s1, s2)
        new_pop.append(s1p)
        new_pop.append(s2p)

    return new_pop


def crossover_pair(s1, s2):
    """
    Takes two strings for crossover, randomly determines a crossover point
    between 1 and len(s1)-1 then performs crossover of character data at crossover point.
    Assumes len(s1) = len(s2)

    Args:
        s1 (str): string for mating crossover
        s2 (str): string for mating crossover
    Returns:
        s1p (str): string from mating crossover
        s2p (str): string from mating crossover
    """
    crossover_point = random.randint(1, len(s1) - 2)
    s1p = s1[:crossover_point] + s2[crossover_point:]
    s2p = s2[:crossover_point] + s1[crossover_point:]

    return (s1p, s2p)


def perform_mutations(population, probability_of_mutation, alnum_set):
    """
    Takes a list of strings, returns a list of strings after potential mutation operation.

    Args:
        population (List[str]): List of strings that make up the mating pool
        probability_of_mutation (float): decimal number between 0 and 1 that represents
```

```python
        the liklihood
                                            that a character will mutate into another character from
                                            its alphanumeric character set
        alnum_set (List[str]): Valid alphanumeric characters to genrate number-system value-strings from
    Returns:
        (List[str]): List of strings that make up a potentially mutated mating pool
    """

    return [attempt_mutation(s, probability_of_mutation, alnum_set) for s in population]


def attempt_mutation(s, probability_of_mutation, alnum_set):
    """
    Takes a string and probabilisticly performs character mutation

    Args:
        s (str): string to mutate
        probability_of_mutation (float): decimal number between 0 and 1 that represents the liklihood
                                            that a character will mutate into another character from
                                            its alphanumeric character set
        alnum_set (List[str]): Valid alphanumeric characters to genrate number-system value-strings from
    Returns:
        None
    """

    #bit flipping
    #on average, method 2 seems to outperform method 1

    # #method 1
    # #choose one bit randomly if random chance falls into the probability  that the string should mutate
    # if random.random() < probability_of_mutation:
    #     bit_to_flip = random.randint(0, len(s) - 1)
    #     s = list(s)
    #     s[bit_to_flip] = random.choice(alnum_set)
    #     return "".join(s)
    # else:
    #     return s

    #method 2
    #bit by bit, perform mutation if random chance falls into the probability  that the bit should mutate
    for i in range(len(s)):
        if random.random() < probability_of_mutation:
            s = s[:i] + str(alnum_set[random.randint(0, len(alnum_set)-1)]) + s[i + 1:]
    return s

# program starts here:
# -------------------------------------------------------------------------------------------------------------------------------------------

def SGA(test_function, pop_size, alnum_set, var_string_length, variable_length, domain_min, domain_max,
        number_of_generations, probability_of_mutation):
    """
    Simple Genetic Algorithm that finds global minima of test functions through generational mating,
    reproduction and bit mutations while printing out each generation's performance results.

```

```python
    Args:
        test_function (<function>): Benchmark function that has known optimum values (global min)
        pop_size (int): the number of strings to create for population
        alnum_set (List[str]): Valid alphanumeric characters to genrate number system value-strings from
        var_string_length (int): character length of string that contains one or more number system value-string string variables
        variable_length (int): character length of one number system value-string variable
        domain_min (Union[float,int]): min value of operational domain
        domain_max (Union[float,int]): max value of operational domain
        number_of_generations (int): number of times to mate / mutate the poulation in the attempt to hone in
                                        on the optimal input values for the given test_function
        probability_of_mutation (float): decimal number between 0 and 1 that represents the liklihood
                                            that a character will mutate into another character from
                                            its alphanumeric character set
    Prints:
        table: generational performances, avoiding repeat max performance levels between contiguous generations
    """

    #initialize a random population of pop_size values to be the starting point for optimization attempt
    #returns string with (var_string_length * pop_size) number of characters
    population = initialize(pop_size, alnum_set, var_string_length)

    #small anonymous function used to find fitness measures of a member of a mating pool population,
    #determined by the given benchmark test function
    inverse_fitness = lambda string: test_function(*general_decoder(string, variable_length, domain_min, domain_max, len(alnum_set)))


    #print perfromance of poulation
    #header
    print("\nTested population size: ", pop_size, " Number of generations: ", number_of_generations)
    pad = str((4 + LENGTH_OF_DECIMAL) * int(var_string_length / variable_length))
    print(("\n{:<16s}{:<" + pad + "s}\t {:}").format("Generation", "Strongest Candidate", "Fitness"))
    print("="*80)

    #print off generational performances, avoiding repeat performance levels between contiguous generations
    last_fit_individual = fittest_individual = [] #coordinate values
    last_max_fit = 0 #fitness value
    new_fit = True #is the found fitness value different than the last
    max_fitness_list = [] #list of all found fitness values to be used for a graph
    global_max_found = (0, [], 0) #to record the overall best found fitness measure form all generations
    first_gen_repeat = last_gen_repeat = 0 #to keep track of how many generations have had repeated max_fitness measures

    for i in range(number_of_generations):

        #create new generation of population
        population = perform_reproduction(population, inverse_fitness)
        population = perform_mating(population)
        population = perform_mutations(population, probability_of_mutation, alnum_set)

        #determine the max fitness measure of this generation
        max_fitness = max(1/(inverse_fitness(m)+1) for m in population)
```

```
        max_fitness_list.append(max_fitness)

        #determine the string variable values that have max_fitness of this generation
        for m in population:
            if 1/(inverse_fitness(m)+1) == max_fitness:
                fittest_individual = general_decoder(m, variable_length, domain_min, domain_max, len(alnum_set))

    #case: if this generation is the last, or it is more fit than its predecessor
        if (i == number_of_generations - 1) or not (fittest_individual == last_fit_individual):
            #case: if is the new fittest member after repeated max peformance
            if (first_gen_repeat != last_gen_repeat) and (last_gen_repeat - first_gen_repeat > 1) :
                print(("\n\tFor generation {} to {}, the max fitness level was {:."+str(LENGTH_OF_DECIMAL)+"f}.\n").format(first_gen_repeat, last_gen_repeat, last_max_fit))

            #print generation's performance results
            print(("{:<16d}[{:<" + pad + "s}]\t {:>}").format(i, " ".join([("{:." + str(LENGTH_OF_DECIMAL) + "f},").format(x) for x in fittest_individual]), max_fitness))

            last_fit_individual = fittest_individual
            last_max_fit = max_fitness

            #record the overall best found fitness measure form all generations
            if max_fitness > global_max_found[2]:
                global_max_found = ("Gen: " + str(i), fittest_individual, max_fitness)

            first_gen_repeat = last_gen_repeat = i
            new_fit = True

        #case: first repeat of same fittest member between generations
        elif last_fit_individual == fittest_individual:
            new_fit = False
            last_gen_repeat = i

    print("="*80)
    print("Highest fitness acheived by:\n", global_max_found)
    print("="*80)
    print("")
    plt.plot(max_fitness_list)
    plt.show()
```