```python
import random
```

```python
# Helper functions
def create_random_individual(terminals, functions):
    if random.random() < 0.5:
        return random.choice(terminals)
    else:
        func = random.choice(functions)
        left = create_random_individual(terminals, functions)
        right = create_random_individual(terminals, functions)
        return (func, left, right)

def evaluate_individual(individual, x_value):
    if isinstance(individual, str):
        if individual == 'x':
            return x_value
        else:
            return float(individual)
    else:
        func, left, right = individual
        left_val = evaluate_individual(left, x_value)
        right_val = evaluate_individual(right, x_value)
        try:
            return func(left_val, right_val)
        except ZeroDivisionError:
            return float('inf')

def objective_function(individual, x_values, target_values):
    error = 0
    for x, target in zip(x_values, target_values):
        prediction = evaluate_individual(individual, x)
        error += (prediction - target) ** 2
    return error

def perform_crossover(parent1, parent2):
    if isinstance(parent1, str) or isinstance(parent2, str):
        return parent1, parent2
    else:
        func1, left1, right1 = parent1
        func2, left2, right2 = parent2
        return (func1, left1, right2), (func2, left2, right1)

def perform_mutation(individual, terminals, functions):
    if isinstance(individual, str):
        return random.choice(terminals)
    else:
        func = random.choice(functions)
        left = perform_mutation(individual[1], terminals, functions)
        right = perform_mutation(individual[2], terminals, functions)
        return (func, left, right)

def select_individual(population):
    return random.choice(population)

def select_parents(population):
    return [(select_individual(population), select_individual(population)) for _ in range(len(population) // 2)]

def generate_offspring(parents, crossover_rate, mutation_rate, terminals, functions):
    offspring = []
    for parent1, parent2 in parents:
        if random.random() < crossover_rate:
            child1, child2 = perform_crossover(parent1, parent2)
        else:
            child1, child2 = parent1, parent2
        if random.random() < mutation_rate:
            child1 = perform_mutation(child1, terminals, functions)
        if random.random() < mutation_rate:
            child2 = perform_mutation(child2, terminals, functions)
        offspring.append(child1)
        offspring.append(child2)
    return offspring

def replace_population(population, offspring, x_values, target_values, max_population_size):
    combined = population + offspring
    combined.sort(key=lambda x: objective_function(x, x_values, target_values))
    return combined[:max_population_size]
```

```
# Gene Expression Programming main loop
def gene_expression_programming(x_values, target_values, terminals, functions, pop_size=100, max_generations=100, crossover_rate=0.7, mutatio
    population = [create_random_individual(terminals, functions) for _ in range(pop_size)]
    for generation in range(max_generations):
        population.sort(key=lambda ind: objective_function(ind, x_values, target_values))
        best_individual = population[0]
        best_fitness = objective_function(best_individual, x_values, target_values)
        print(f"Generation {generation}, Best Fitness: {best_fitness}")
        if best_fitness < 1e-5:
            print("Found a solution!")
            return best_individual
        parents = select_parents(population)
        offspring = generate_offspring(parents, crossover_rate, mutation_rate, terminals, functions)
        population = replace_population(population, offspring, x_values, target_values, pop_size)
    return population[0]

# Define terminals and functions
terminals = ['x', '1', '2', '3', '4']
functions = [
    lambda a, b: a + b,
    lambda a, b: a - b,
    lambda a, b: a * b,
    lambda a, b: a / (b + 0.001)  # Avoid division by zero
]

# Define the target function and data
x_values = [i for i in range(-10, 11)]
target_values = [x ** 2 for x in x_values]

# Run the Gene Expression Programming algorithm
best_individual = gene_expression_programming(x_values, target_values, terminals, functions)
print("Best solution:", best_individual)
```

```
Best Fitness: 31402.0
Best Fitness: 31402.0
Best Fitness: 31402.0
Best Fitness: 31402.0
Best Fitness: 23674.869072931215
Best Fitness: 23674.869072931215
Best Fitness: 22516.36836482696
Best Fitness: 22516.36836482696
Best Fitness: 22516.36836482696
Best Fitness: 22516.36836482696
 Best Fitness: 22516.36836482696
 Best Fitness: 22516.36836482696
 Best Fitness: 22516.36836482696
 Best Fitness: 22516.36836482696
 Best Fitness: 22516.36836482696
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 2512.3292287776903
 Best Fitness: 336.0
 Best Fitness: 336.0
 Best Fitness: 336.0
 Best Fitness: 336.0
 Best Fitness: 336.0
```

```
 Best Fitness: 336.0
 Best Fitness: 336.0
 Best Fitness: 336.0
 Best Fitness: 336.0
 Best Fitness: 0.0
on!
 (<function <lambda> at 0x7c080aa8dd80>, 'x', (<function <lambda> at 0x7c080aa8de10>, 'x', (<function <lambda> at 0x7c080aa8dd80>, '4',
```

Start coding or generate with AI.