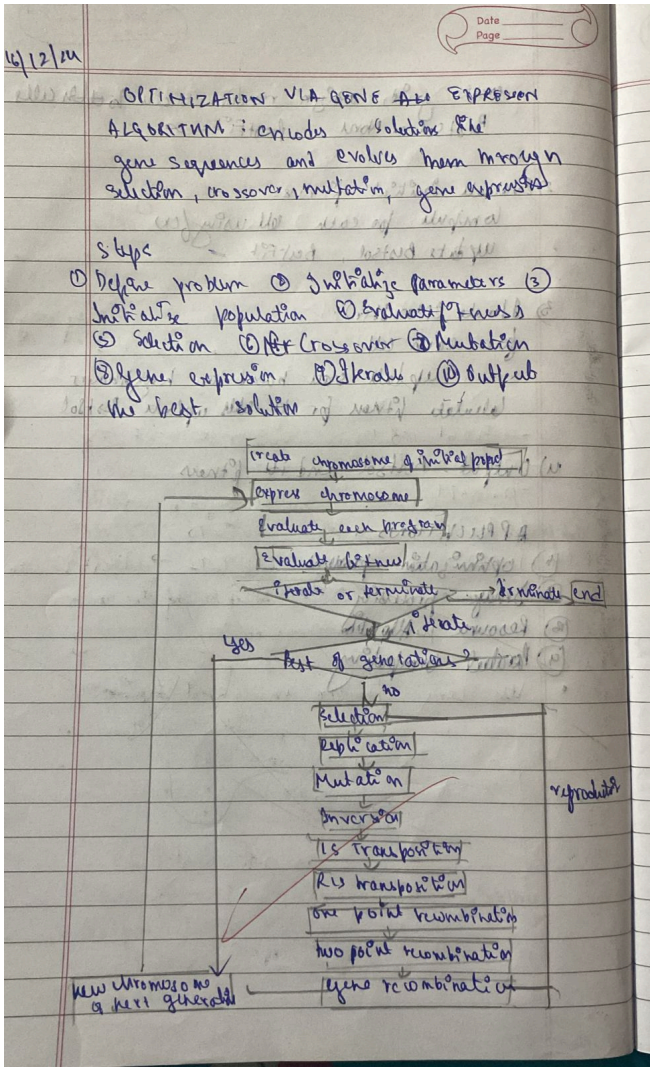
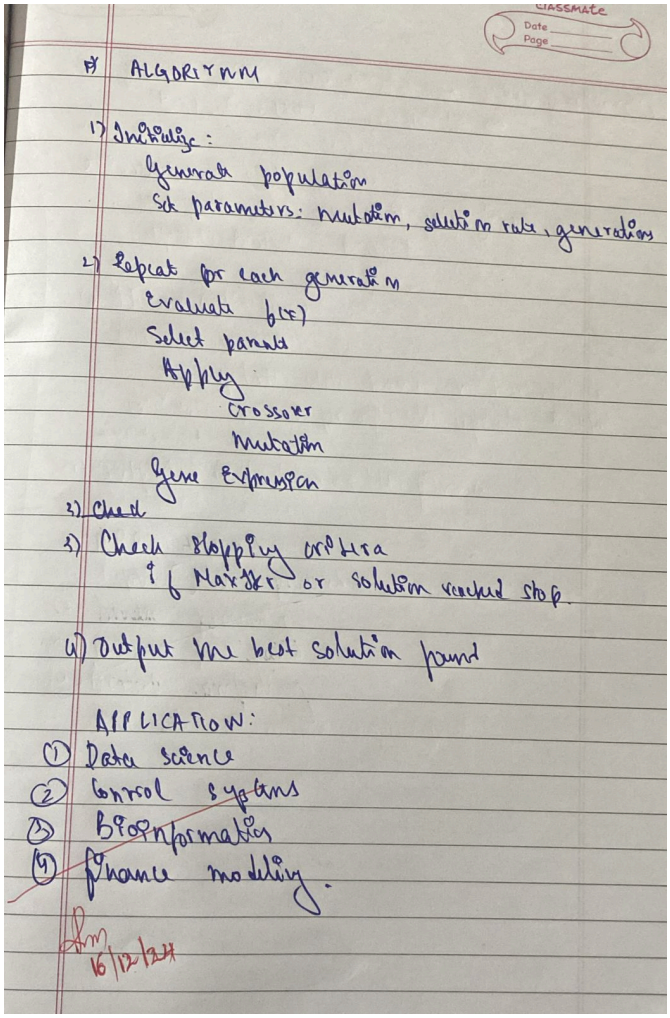
Program 7

Problem statement:Optimization via Gene Expression Algorithms

Algorithm:





Code:

```
import numpy as np
def objective_function(x):
    return x**2 - 4*x + 4
class GeneExpressionAlgorithm:
    def __init__(self, objective_function, population_size, num_genes, max_generations, mutation_rate=0.01, crossover_rate=0.7, lb=-5, ub=5):
        self.objective_function = objective_function
        self.population_size = population_size
        self.num_genes = num_genes
        self.max_generations = max_generations
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.lb = lb
        self.ub = ub
        self.population = np.random.uniform(self.lb, self.ub, (self.population_size, self.num_genes))
        self.fitness = np.array([self.objective_function(individual.sum()) for individual in self.population])
```

```
        self.best_solution = self.population[np.argmin(self.fitness)]
        self.best_fitness = np.min(self.fitness)
    def selection(self):
        total_fitness = np.sum(self.fitness)
        probabilities = (total_fitness - self.fitness) / total_fitness
        probabilities /= np.sum(probabilities)
        selected_indices = np.random.choice(np.arange(self.population_size),
size=self.population_size, p=probabilities)
        selected_population = self.population[selected_indices]
        return selected_population
    def crossover(self, parent1, parent2):
        if np.random.rand() < self.crossover_rate:
            crossover_point = np.random.randint(1, self.num_genes)
            child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
            child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
            return child1, child2
        else:
            return parent1, parent2
    def mutation(self, individual):
        if np.random.rand() < self.mutation_rate:
            mutation_point = np.random.randint(self.num_genes)
            individual[mutation_point] = np.random.uniform(self.lb, self.ub)
        return individual
    def gene_expression(self, individual):
        return individual.sum()
    def run(self):
        for generation in range(self.max_generations):
            selected_population = self.selection()
            new_population = []
            for i in range(0, self.population_size, 2):
                parent1, parent2 = selected_population[i], selected_population[i+1]
                child1, child2 = self.crossover(parent1, parent2)
                new_population.extend([self.mutation(child1), self.mutation(child2)])

            self.population = np.array(new_population)
            self.fitness = np.array([self.objective_function(self.gene_expression(individual)) for
individual in self.population])
            current_best_solution = self.population[np.argmin(self.fitness)]
            current_best_fitness = np.min(self.fitness)
            if current_best_fitness < self.best_fitness:
                self.best_solution = current_best_solution
                self.best_fitness = current_best_fitness
            print(f"Generation {generation+1}/{self.max_generations}: Best Fitness =
{self.best_fitness}")
        return self.best_solution, self.best_fitness
population_size = 30
```

num_genes = 5
max_generations = 10
lb = -5
ub = 5
gea = GeneExpressionAlgorithm(objective_function, population_size, num_genes, max_generations, lb=lb, ub=ub)
best_solution, best_fitness = gea.run()
print(f"\nBest Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")

```python
import numpy as np
def objective_function(x):
    return x**2 - 4*x + 4
class GeneExpressionAlgorithm:
    def __init__(self, objective_function, population_size, num_genes, max_generations, mutation_rate=0.01, crossover_rate=0.7, lb=-5, ub=5):
        self.objective_function = objective_function
        self.population_size = population_size
        self.num_genes = num_genes
        self.max_generations = max_generations
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.lb = lb
        self.ub = ub
        self.population = np.random.uniform(self.lb, self.ub, (self.population_size, self.num_genes))
        self.fitness = np.array([self.objective_function(individual.sum()) for individual in self.population])
        self.best_solution = self.population[np.argmin(self.fitness)]
        self.best_fitness = np.min(self.fitness)
    def selection(self):
        total_fitness = np.sum(self.fitness)
        probabilities = (total_fitness - self.fitness) / total_fitness
        probabilities /= np.sum(probabilities)
        selected_indices = np.random.choice(np.arange(self.population_size), size=self.population_size, p=probabilities)
        selected_population = self.population[selected_indices]
        return selected_population
    def crossover(self, parent1, parent2):
        if np.random.rand() < self.crossover_rate:
            crossover_point = np.random.randint(1, self.num_genes)
            child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
            child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
            return child1, child2
        else:
            return parent1, parent2
    def mutation(self, individual):
        if np.random.rand() < self.mutation_rate:
            mutation_point = np.random.randint(self.num_genes)
            individual[mutation_point] = np.random.uniform(self.lb, self.ub)
        return individual
    def gene_expression(self, individual):
        return individual.sum()
    def run(self):
        for generation in range(self.max_generations):
            selected_population = self.selection()
            new_population = []
            for i in range(0, self.population_size, 2):
                parent1, parent2 = selected_population[i], selected_population[i+1]
                child1, child2 = self.crossover(parent1, parent2)
                new_population.extend([self.mutation(child1), self.mutation(child2)])

            self.population = np.array(new_population)
            self.fitness = np.array([self.objective_function(self.gene_expression(individual)) for individual in self.population])
            current_best_solution = self.population[np.argmin(self.fitness)]
            current_best_fitness = np.min(self.fitness)
            if current_best_fitness < self.best_fitness:
                self.best_solution = current_best_solution
                self.best_fitness = current_best_fitness
```

```
            print(f"Generation {generation+1}/{self.max_generations}: Best Fitness = {self.best_fitness}")
        return self.best_solution, self.best_fitness
population_size = 30
num_genes = 5
max_generations = 10
lb = -5
ub = 5
gea = GeneExpressionAlgorithm(objective_function, population_size, num_genes, max_generations, lb=lb, ub=ub)
best_solution, best_fitness = gea.run()
print(f"\nBest Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")
```

Output:

```
Generation 1/10: Best Fitness = 0.10977190093840417
Generation 2/10: Best Fitness = 0.0022541941958982292
Generation 3/10: Best Fitness = 0.0022541941958982292
Generation 4/10: Best Fitness = 0.0022541941958982292
Generation 5/10: Best Fitness = 0.0022541941958982292
Generation 6/10: Best Fitness = 0.0022541941958982292
Generation 7/10: Best Fitness = 0.0022541941958982292
Generation 8/10: Best Fitness = 0.0022541941958982292
Generation 9/10: Best Fitness = 0.0022541941958982292
Generation 10/10: Best Fitness = 0.0022541941958982292

Best Solution: [ 4.26602667 -3.46384098  2.59225916 -1.68596965  0.33900316]
Best Fitness: 0.0022541941958982292
```