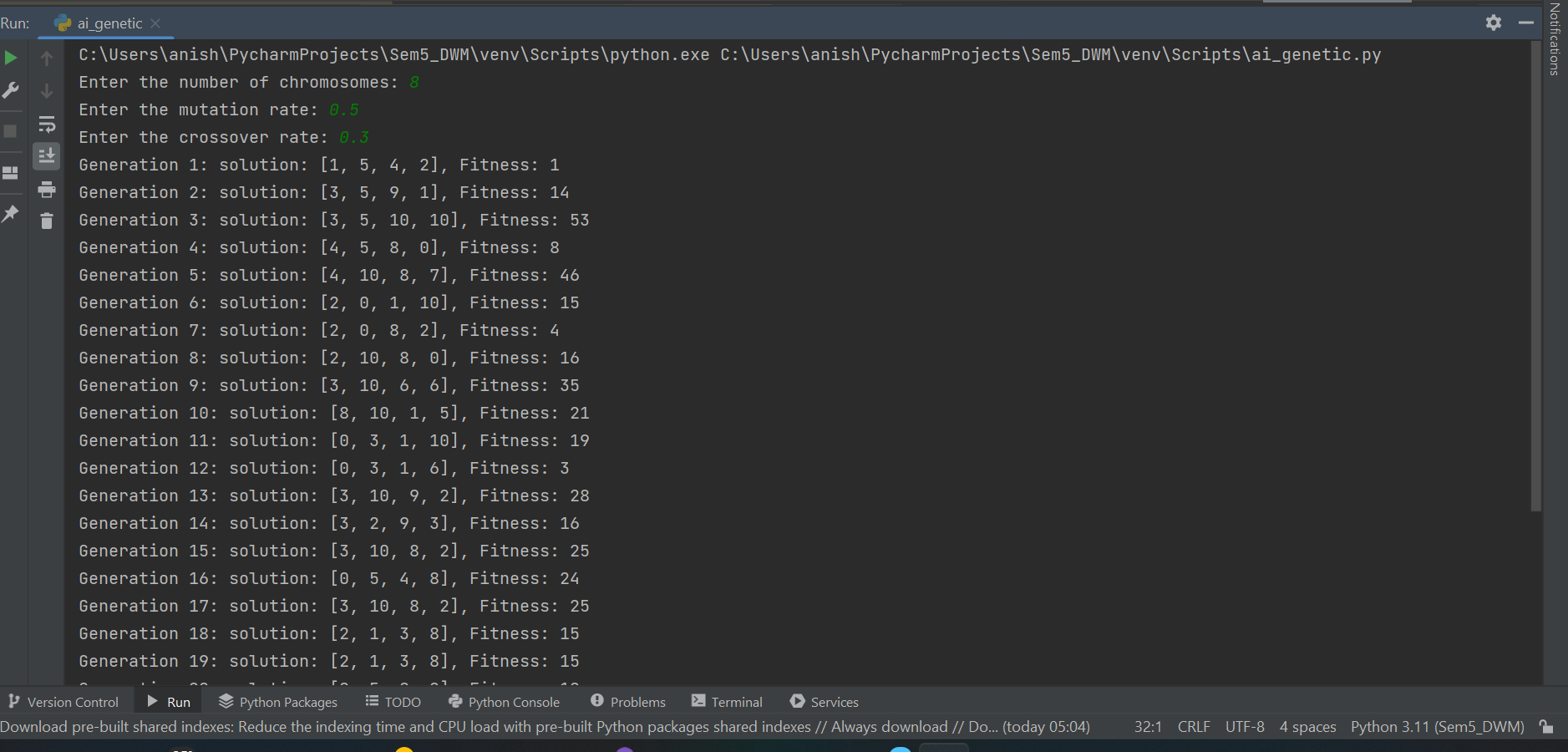
**Anisha Jain**

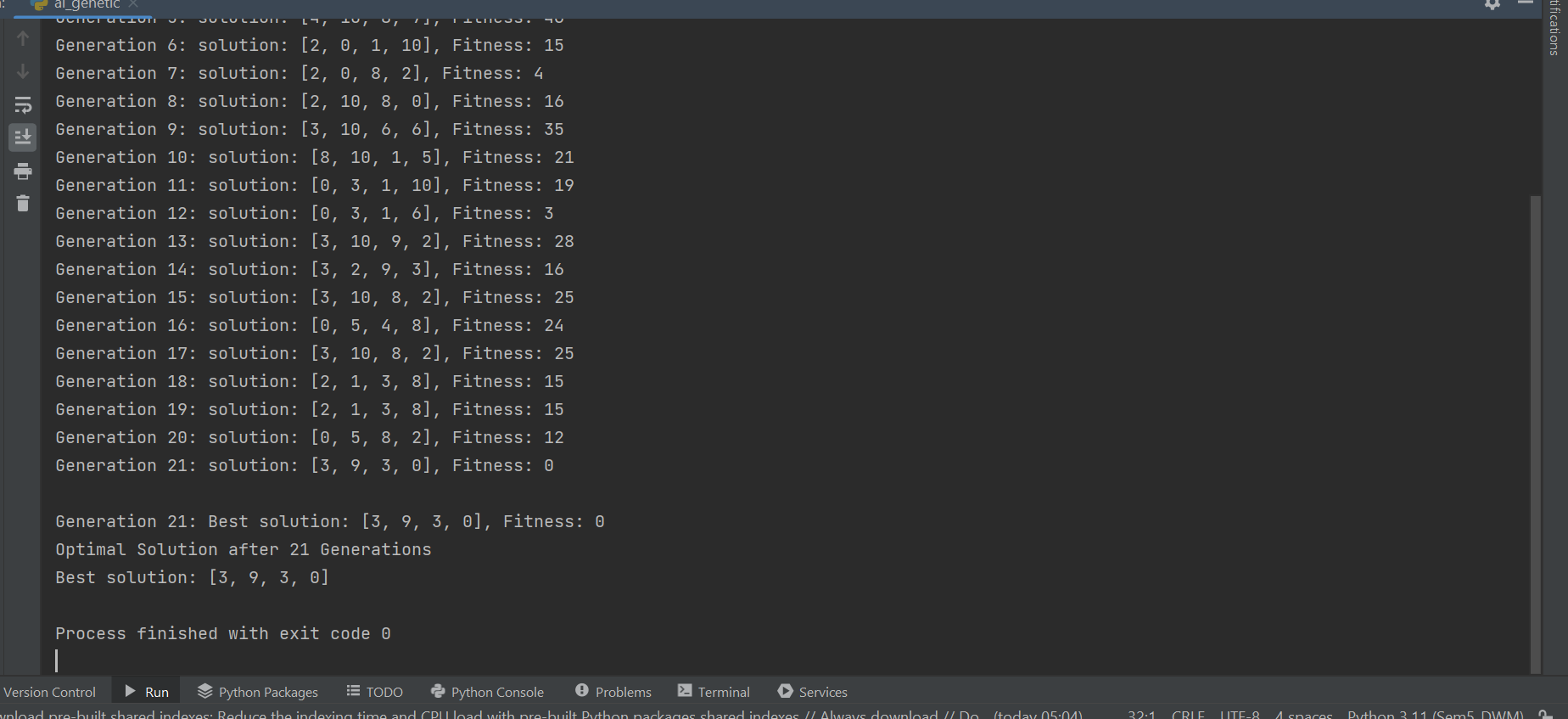
**C14-65**

**EXPERIMENT 5**

**Genetics Algorithm :**

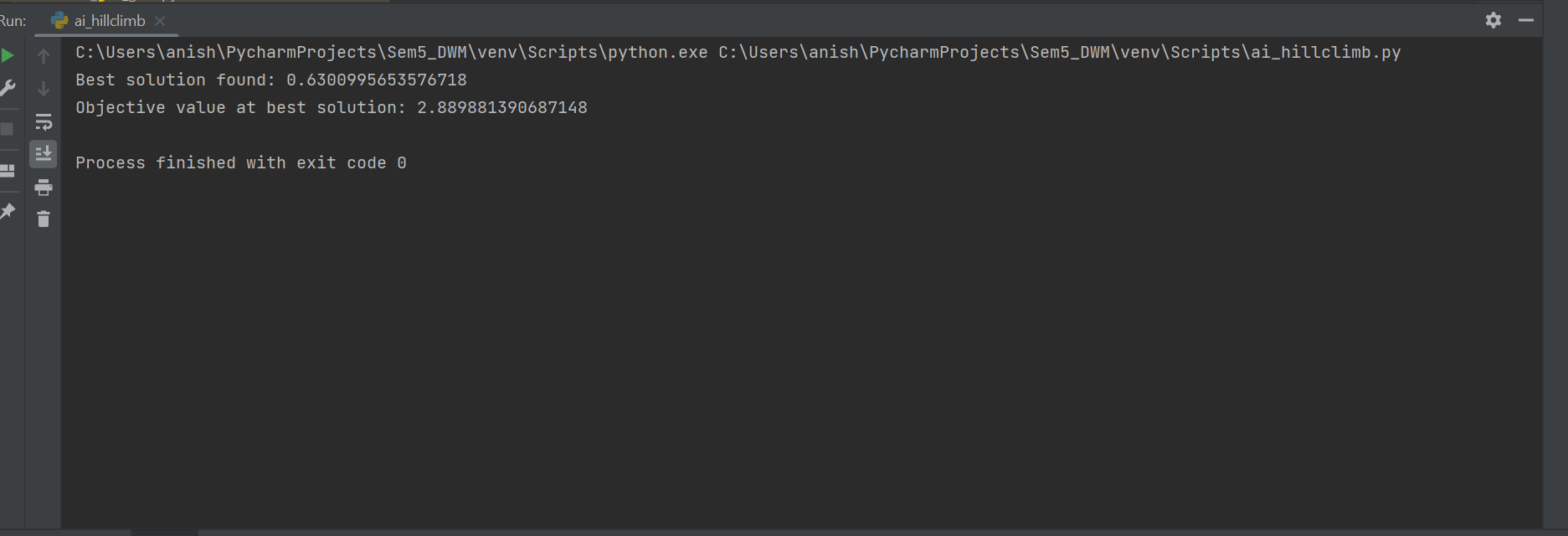
import random# Step 1: Determine the number of chromosomes, generation, mutation rate, and crossover rate valuedef genetic\_algorithm(num\_chromosomes, mutation\_rate, crossover\_rate, num\_variables): # Step 2: Generate chromosome population with random initialization population = generate\_population(num\_chromosomes, num\_variables) i = int(1) while True: # Step 3: Evaluation of fitness value of chromosomes fitness\_values = [fitness\_function(chromosome) for chromosome in population] # Step 8: Solution (Best Chromosomes) best\_chromosome = population[fitness\_values.index(min(fitness\_values))] print(f"Generation {i}: solution: {best\_chromosome}, Fitness: {min(fitness\_values)}") if min(fitness\_values) == 0: print(f"\nGeneration {i}: Best solution: {best\_chromosome}, Fitness: {min(fitness\_values)}") print(f"Optimal Solution after {i} Generations") break i = i + 1 next\_generation = [] # Step 5: Chromosomes selection selected\_population = selection(population, fitness\_values) for \_ in range(num\_chromosomes // 2): parent1, parent2 = selected\_population[random.randint(0, len(selected\_population) - 1)], selected\_population[random.randint(0, len(selected\_population) - 1)] # Step 6: Crossover child1, child2 = crossover(parent1, parent2, crossover\_rate) # Step 7: Mutation child1 = mutation(child1, mutation\_rate) child2 = mutation(child2, mutation\_rate) next\_generation.extend([child1, child2]) population = next\_generation return best\_chromosome# Step 2: Generate initial populationdef generate\_population(size, num\_variables): population = [] for \_ in range(size): individual = [random.randint(0, 10) for \_ in range(num\_variables)] population.append(individual) return population# Step 4: Fitness function evaluationdef fitness\_function(variables): x, y, z, w = variables return abs((x + 2\*y + 3\*z + 4\*w) - 30)# Step 5: Chromosome selection using roulette wheel selectiondef selection(population, fitness\_values): selected\_population = [] total\_fitness = sum(fitness\_values) probabilities = [fitness / total\_fitness for fitness in fitness\_values] for \_ in range(len(population)): selected = random.choices(population, probabilities)[0] selected\_population.append(selected) return selected\_population# Step 6: Crossoverdef crossover(parent1, parent2, crossover\_rate): if random.random() < crossover\_rate: crossover\_point = random.randint(1, len(parent1) - 1) child1 = parent1[:crossover\_point] + parent2[crossover\_point:] child2 = parent2[:crossover\_point] + parent1[crossover\_point:] return child1, child2 return parent1, parent2# Step 7: Mutationdef mutation(individual, mutation\_rate): if random.random() < mutation\_rate: index = random.randint(0, len(individual) - 1) new\_value = random.randint(0, 10) individual[index] = new\_value return individual# Get input from the userdef get\_input(): num\_chromosomes = int(input("Enter the number of chromosomes: ")) mutation\_rate = float(input("Enter the mutation rate: ")) crossover\_rate = float(input("Enter the crossover rate: ")) num\_variables = 4 # Assuming 4 variables (x, y, z, w) for the given equation return num\_chromosomes, mutation\_rate, crossover\_rate, num\_variables# Main programdef main(): num\_chromosomes, mutation\_rate, crossover\_rate, num\_variables = get\_input() best\_chromosome = genetic\_algorithm(num\_chromosomes, mutation\_rate, crossover\_rate, num\_variables) print("Best solution:", best\_chromosome)if \_\_name\_\_ == "\_\_main\_\_": main()





**Hill Climbing Algorithm :**

import randomdef obj\_function(x): return -(4\*x \*\* 4)+(4\*x)+1def random\_neighbor(current): return current + random.uniform(-0.5, 0.5)def hill\_climbing(max\_iterations): current\_solution = random.uniform(-10, 10) for \_ in range(max\_iterations): neighbor = random\_neighbor(current\_solution) if obj\_function(neighbor) > obj\_function(current\_solution): current\_solution = neighbor return current\_solutionbest\_solution = hill\_climbing(1000)print("Best solution found:", best\_solution)print("Objective value at best solution:", obj\_function(best\_solution))

****