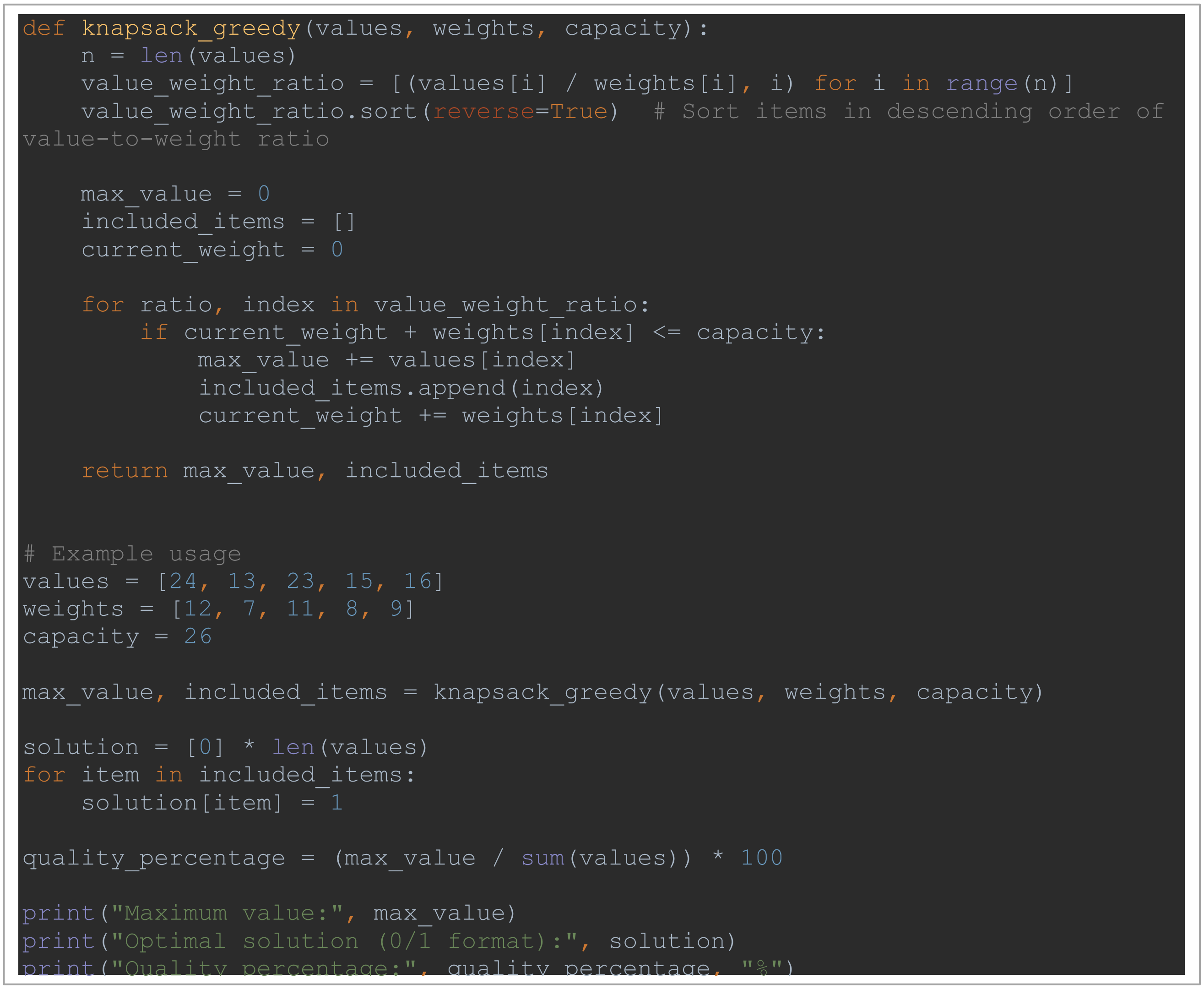
Optimization & Regression CCAI-311

Course Project

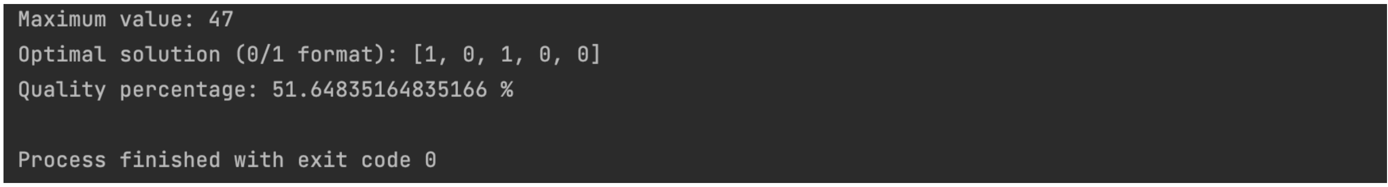
STUDENT NAME	STUDENT ID
KHALID NIMRI	2140145
ASEEL SUHAIL	2140197

1. Approximate method for the KP



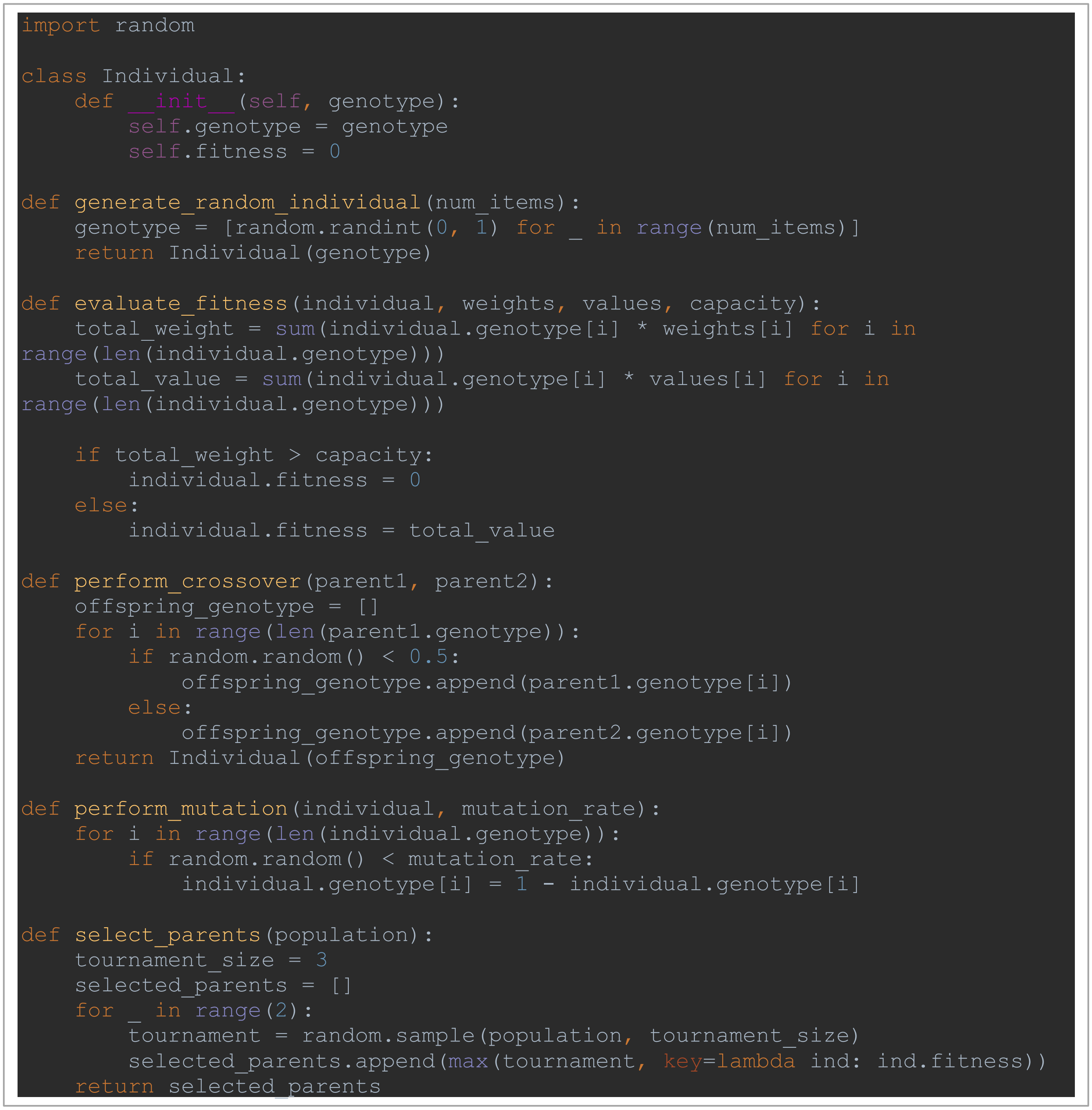
```
def knapsack_greedy(values, weights, capacity):
    n = len(values)
    value_weight_ratio = [(values[i] / weights[i], i) for i in range(n)]
    value_weight_ratio.sort(reverse=True)   # Sort items in descending order of
value-to-weight ratio

    max_value = 0
    included_items = []
    current_weight = 0

    for ratio, index in value_weight_ratio:
        if current_weight + weights[index] <= capacity:
            max_value += values[index]
            included_items.append(index)
            current_weight += weights[index]

    return max_value, included_items


# Example usage
values = [24, 13, 23, 15, 16]
weights = [12, 7, 11, 8, 9]
capacity = 26

max_value, included_items = knapsack_greedy(values, weights, capacity)

solution = [0] * len(values)
for item in included_items:
    solution[item] = 1

quality_percentage = (max_value / sum(values)) * 100

print("Maximum value:", max_value)
print("Optimal solution (0/1 format):", solution)
print("Quality percentage:", quality_percentage, "%")
```

Output:

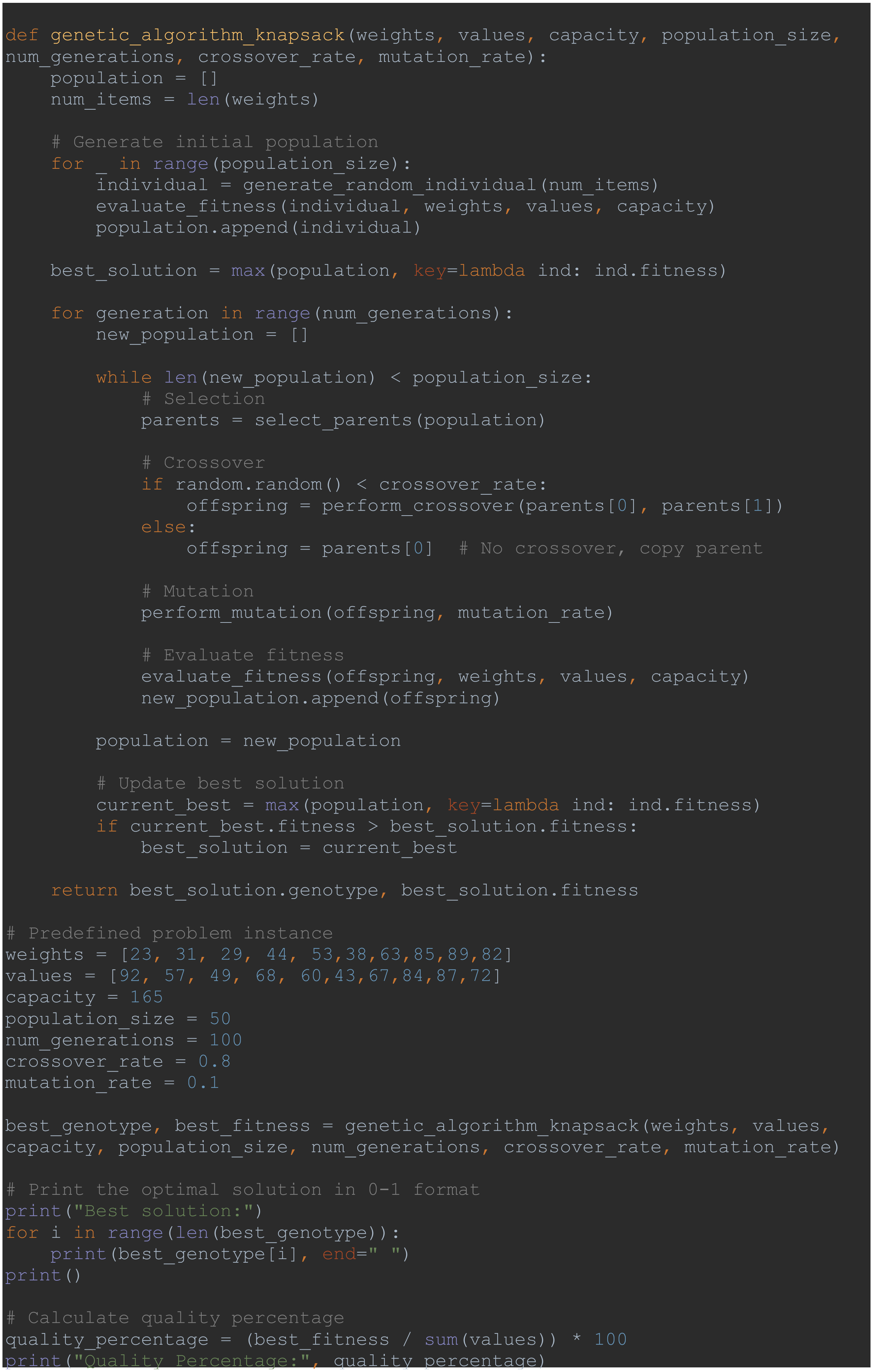


```
Maximum value: 47
Optimal solution (0/1 format): [1, 0, 1, 0, 0]
Quality percentage: 51.64835164835166 %

Process finished with exit code 0
```

2. Exact method for the KP :



```
import random

class Individual:
    def __init__(self, genotype):
        self.genotype = genotype
        self.fitness = 0

def generate_random_individual(num_items):
    genotype = [random.randint(0, 1) for _ in range(num_items)]
    return Individual(genotype)

def evaluate_fitness(individual, weights, values, capacity):
    total_weight = sum(individual.genotype[i] * weights[i] for i in
range(len(individual.genotype)))
    total_value = sum(individual.genotype[i] * values[i] for i in
range(len(individual.genotype)))

    if total_weight > capacity:
        individual.fitness = 0
    else:
        individual.fitness = total_value

def perform_crossover(parent1, parent2):
    offspring_genotype = []
    for i in range(len(parent1.genotype)):
        if random.random() < 0.5:
            offspring_genotype.append(parent1.genotype[i])
        else:
            offspring_genotype.append(parent2.genotype[i])
    return Individual(offspring_genotype)

def perform_mutation(individual, mutation_rate):
    for i in range(len(individual.genotype)):
        if random.random() < mutation_rate:
            individual.genotype[i] = 1 - individual.genotype[i]

def select_parents(population):
    tournament_size = 3
    selected_parents = []
    for _ in range(2):
        tournament = random.sample(population, tournament_size)
        selected_parents.append(max(tournament, key=lambda ind: ind.fitness))
    return selected_parents
```



```python
def genetic_algorithm_knapsack(weights, values, capacity, population_size,
num_generations, crossover_rate, mutation_rate):
    population = []
    num_items = len(weights)

    # Generate initial population
    for _ in range(population_size):
        individual = generate_random_individual(num_items)
        evaluate_fitness(individual, weights, values, capacity)
        population.append(individual)

    best_solution = max(population, key=lambda ind: ind.fitness)

    for generation in range(num_generations):
        new_population = []

        while len(new_population) < population_size:
            # Selection
            parents = select_parents(population)

            # Crossover
            if random.random() < crossover_rate:
                offspring = perform_crossover(parents[0], parents[1])
            else:
                offspring = parents[0]  # No crossover, copy parent

            # Mutation
            perform_mutation(offspring, mutation_rate)

            # Evaluate fitness
            evaluate_fitness(offspring, weights, values, capacity)
            new_population.append(offspring)

        population = new_population

        # Update best solution
        current_best = max(population, key=lambda ind: ind.fitness)
        if current_best.fitness > best_solution.fitness:
            best_solution = current_best

    return best_solution.genotype, best_solution.fitness

# Predefined problem instance
weights = [23, 31, 29, 44, 53,38,63,85,89,82]
values = [92, 57, 49, 68, 60,43,67,84,87,72]
capacity = 165
population_size = 50
num_generations = 100
crossover_rate = 0.8
mutation_rate = 0.1

best_genotype, best_fitness = genetic_algorithm_knapsack(weights, values,
capacity, population_size, num_generations, crossover_rate, mutation_rate)

# Print the optimal solution in 0-1 format
print("Best solution:")
for i in range(len(best_genotype)):
    print(best_genotype[i], end=" ")
print()

# Calculate quality percentage
quality_percentage = (best_fitness / sum(values)) * 100
print("Quality Percentage:", quality_percentage)
```

3. Comparison of the obtained results
&
4. Instances (8 instances) and details.

Instance	Approximate algorithm	Exact algorithm
1	**45.51%**	**45.51%**
2	**51.65%**	**56.04%**
3	**55.09%**	**56.60%**
4	**54.25%**	**56.91%**
5	**65.59%**	**68.80%**
6	**38.91%**	**45.68%**
7	**52.29%**	**52.90%**
8	**51.77%**	**53.47%**