```python
import random

# Define the Knapsack Problem (Objective Function)
def knapsack_fitness(items, capacity, solution):
    total_weight = sum([items[i][0] for i in range(len(solution)) if solution[i] == 1])
    total_value = sum([items[i][1] for i in range(len(solution)) if solution[i] == 1])

    # If total weight exceeds the capacity, return 0 (invalid solution)
    if total_weight > capacity:
        return 0
    return total_value

# Gene Expression Algorithm (GEA)
class GeneExpressionAlgorithm:
    def __init__(self, population_size, num_items, mutation_rate, crossover_rate, generations, capacity, items):
        self.population_size = population_size
        self.num_items = num_items
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.generations = generations
        self.capacity = capacity
        self.items = items
        self.population = []

    # Initialize population with random solutions (binary representation)
    def initialize_population(self):
        self.population = [[random.randint(0, 1) for _ in range(self.num_items)] for _ in range(self.population_size)]

    # Evaluate fitness of the population
    def evaluate_fitness(self):
        return [knapsack_fitness(self.items, self.capacity, individual) for individual in self.population]

    # Select individuals based on fitness (roulette wheel selection)
    def selection(self):
        fitness_values = self.evaluate_fitness()
        total_fitness = sum(fitness_values)
        if total_fitness == 0:  # Avoid division by zero
            return random.choices(self.population, k=self.population_size)
        return random.choices(self.population, weights=[f / total_fitness for f in fitness_values], k=self.population_size)

    # Crossover (single-point) between two individuals
    def crossover(self, parent1, parent2):
        if random.random() < self.crossover_rate:
            crossover_point = random.randint(1, self.num_items - 1)
            return parent1[:crossover_point] + parent2[crossover_point:]
        return parent1
```

```python
                mutation_point = random.randint(
                individual[mutation_point] = 1 - individual[mutation_point]
        return individual

    # Evolve population over generations
    def evolve(self):
        self.initialize_population()
        best_solution = None
        best_fitness = 0

        for gen in range(self.generations):
            # Selection
            selected = self.selection()

            # Crossover and Mutation
            new_population = []
            for i in range(0, self.population_size, 2):
                parent1 = selected[i]
                parent2 = selected[i + 1] if i + 1 < self.population_size else selected[i]

                offspring1 = self.crossover(parent1, parent2)
                offspring2 = self.crossover(parent2, parent1)

                new_population.append(self.mutation(offspring1))
                new_population.append(self.mutation(offspring2))

            self.population = new_population

            # Evaluate fitness and track the best solution
            fitness_values = self.evaluate_fitness()
            max_fitness = max(fitness_values)
            if max_fitness > best_fitness:
                best_fitness = max_fitness
                best_solution = self.population[fitness_values.index(max_fitness)]

            print(f"Generation {gen + 1}: Best Fitness = {best_fitness}")

        return best_solution, best_fitness
```

```python
# Get user input for the knapsack problem
def get_user_input():
    print("Enter the number of items:")
    num_items = int(input())
    items = []
    print("Enter the weight and value of each item (space-separated):")
    for i in range(num_items):
        weight, value = map(int, input(f"Item {i + 1}: ").split())
        items.append((weight, value))

    print("Enter the knapsack capacity:")
    capacity = int(input())

    return items, capacity, num_items

# Get user input for GEA parameters
def get_algorithm_parameters():
    print("Enter the population size:")
    population_size = int(input())
    print("Enter the mutation rate (e.g., 0.1 for 10%):")
    mutation_rate = float(input())
    print("Enter the crossover rate (e.g., 0.8 for 80%):")
    crossover_rate = float(input())
    print("Enter the number of generations:")
    generations = int(input())

    return population_size, mutation_rate, crossover_rate, generations

# Main function
if __name__ == "__main__":
    # Get user input
    items, capacity, num_items = get_user_input()
    population_size, mutation_rate, crossover_rate, generations = get_algorithm_parameters()

    # Run GEA
    gea = GeneExpressionAlgorithm(population_size, num_items, mutation_rate, crossover_rate, generations, capacity, items)
    best_solution, best_fitness = gea.evolve()

    print("\nBest Solution:", best_solution)
```

```
Enter the number of items:
5
Enter the weight and value of each item (space-separated):
Item 1: 10 20
Item 2: 30 40
Item 3: 50 60
Item 4: 70 80
Item 5: 90 100
Enter the knapsack capacity:
90
Enter the population size:
100
Enter the mutation rate (e.g., 0.1 for 10%):
0.2
Enter the crossover rate (e.g., 0.8 for 80%):
0.9
Enter the number of generations:
10
Generation 1: Best Fitness = 120
Generation 2: Best Fitness = 120
Generation 3: Best Fitness = 120
Generation 4: Best Fitness = 120
Generation 5: Best Fitness = 120
Generation 6: Best Fitness = 120
Generation 7: Best Fitness = 120
Generation 8: Best Fitness = 120
Generation 9: Best Fitness = 120
Generation 10: Best Fitness = 120

Best Solution: [1, 1, 1, 0, 0]
Best Fitness (Total Value): 120
```