**NAME:** Abdullah Aslam Qazi

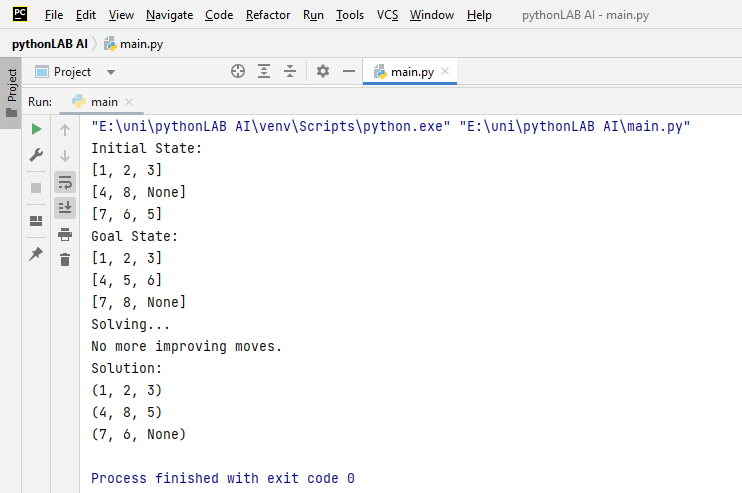
**REGNO:** BCS 203263

**Hill Climbing**

CODE:

import random  
  
*# Function to calculate the heuristic (number of misplaced tiles)*def calculate\_heuristic(state, goal\_state):  
 count = 0  
 for i in range(3):  
 for j in range(3):  
 if state[i][j] != goal\_state[i][j]:  
 count += 1  
 return count  
  
*# Function to generate valid moves*def generate\_moves(state):  
 zero\_pos = find\_zero\_position(state)  
 possible\_moves = []  
  
 if zero\_pos[0] != 0:  
 possible\_moves.append((zero\_pos[0] - 1, zero\_pos[1])) *# Move up* if zero\_pos[0] != 2:  
 possible\_moves.append((zero\_pos[0] + 1, zero\_pos[1])) *# Move down* if zero\_pos[1] != 0:  
 possible\_moves.append((zero\_pos[0], zero\_pos[1] - 1)) *# Move left* if zero\_pos[1] != 2:  
 possible\_moves.append((zero\_pos[0], zero\_pos[1] + 1)) *# Move right* return possible\_moves  
  
*# Function to find the position of the zero tile*def find\_zero\_position(state):  
 for i in range(3):  
 for j in range(3):  
 if state[i][j] is None:  
 return (i, j)  
  
*# Convert the 2D list state to a tuple for hashing purposes*def convert\_state\_to\_tuple(state):  
 return tuple(map(tuple, state))  
  
*# Convert the tuple state to a 2D list for manipulation purposes*def convert\_state\_to\_list(state):  
 return [list(row) for row in state]  
  
*# Hill climbing algorithm to solve the puzzle*def hill\_climbing(initial\_state, goal\_state):  
 current\_state = initial\_state  
 current\_heuristic = calculate\_heuristic(current\_state, goal\_state)  
  
 while current\_state != goal\_state:  
 possible\_moves = generate\_moves(current\_state)  
 best\_state = current\_state  
 best\_heuristic = current\_heuristic  
  
 for move in possible\_moves:  
 new\_state = convert\_state\_to\_list(current\_state)  
 new\_state[move[0]][move[1]], new\_state[find\_zero\_position(current\_state)[0]][find\_zero\_position(current\_state)[1]] = \  
 new\_state[find\_zero\_position(current\_state)[0]][find\_zero\_position(current\_state)[1]], new\_state[move[0]][move[1]]  
 new\_heuristic = calculate\_heuristic(new\_state, goal\_state)  
  
 if new\_heuristic < best\_heuristic:  
 best\_state = new\_state  
 best\_heuristic = new\_heuristic  
  
 if best\_state == current\_state:  
 print("No more improving moves.")  
 break  
  
 current\_state = convert\_state\_to\_tuple(best\_state)  
 current\_heuristic = best\_heuristic  
  
 return current\_state  
  
*# Test the hill climbing algorithm*initial\_state = [  
 [1, 2, 3],  
 [4, 8, None],  
 [7, 6, 5]  
]  
  
goal\_state = [  
 [1, 2, 3],  
 [4, 5, 6],  
 [7, 8, None]  
]  
  
print("Initial State:")  
for row in initial\_state:  
 print(row)  
print("Goal State:")  
for row in goal\_state:  
 print(row)  
  
print("Solving...")  
solution = hill\_climbing(convert\_state\_to\_tuple(initial\_state), convert\_state\_to\_tuple(goal\_state))  
  
print("Solution:")  
for row in solution:  
 print(row)

OUTPUT:



**Genetic Algo**

CODE:

import random  
  
*# Number of cards*NUM\_CARDS = 10  
  
*# Target sums and product*TARGET\_SUM = 36  
TARGET\_PRODUCT = 360  
  
*# Genetic algorithm parameters*POPULATION\_SIZE = 100  
MUTATION\_RATE = 0.01  
NUM\_GENERATIONS = 100  
  
def create\_individual():  
 *"""Create a random individual."""* cards = list(range(1, NUM\_CARDS + 1))  
 random.shuffle(cards)  
 return cards  
  
def evaluate\_fitness(individual):  
 *"""Evaluate the fitness of an individual."""* sum\_diff = abs(sum(individual[:NUM\_CARDS // 2]) - TARGET\_SUM)  
 product\_diff = abs(  
 1 if sum(individual[NUM\_CARDS // 2:]) == 0 else  
 (TARGET\_PRODUCT / (sum(individual[NUM\_CARDS // 2:]) or 1)) - TARGET\_PRODUCT  
 )  
 return sum\_diff + product\_diff  
  
def crossover(parent1, parent2):  
 *"""Perform crossover between two parents to create two children."""* point = random.randint(1, NUM\_CARDS - 1)  
 child1 = parent1[:point] + parent2[point:]  
 child2 = parent2[:point] + parent1[point:]  
 return child1, child2  
  
def mutate(individual):  
 *"""Mutate an individual by randomly swapping two cards."""* index1, index2 = random.sample(range(NUM\_CARDS), 2)  
 individual[index1], individual[index2] = individual[index2], individual[index1]  
 return individual  
  
def genetic\_algorithm():  
 *"""Run the genetic algorithm to find the optimal solution."""* population = [create\_individual() for \_ in range(POPULATION\_SIZE)]  
  
 for \_ in range(NUM\_GENERATIONS):  
 *# Evaluate fitness for each individual* fitness\_scores = [evaluate\_fitness(individual) for individual in population]  
  
 *# Selection* parents = random.choices(population, weights=[1 / (score or 1) for score in fitness\_scores], k=2)  
  
 *# Crossover and mutation* children = crossover(parents[0], parents[1])  
 mutated\_children = [mutate(child) for child in children]  
  
 *# Replace the two worst individuals in the population* worst\_indices = sorted(range(len(fitness\_scores)), key=lambda i: fitness\_scores[i], reverse=True)[:2]  
 population[worst\_indices[0]], population[worst\_indices[1]] = mutated\_children[0], mutated\_children[1]  
  
 *# Find the best individual after the evolution process* best\_individual = min(population, key=evaluate\_fitness)  
 return best\_individual  
  
*# Run the genetic algorithm*best\_solution = genetic\_algorithm()  
  
*# Output the result*print("Optimal solution:")  
print("Pile 1:", best\_solution[:NUM\_CARDS // 2])  
print("Pile 2:", best\_solution[NUM\_CARDS // 2:])

OUTPUT:

