K230013
BAI-4A
Zunaira Amjad

AI-A2

QUESTION 1

```
def query(x):
    return -1 * (x - 7)**2 + 49

def find_peak(N: int) -> int:
    left = 0
    right = N

    while left < right:
        mid = (left + right) // 2
        if query(mid) < query(mid + 1):
            left = mid + 1
        else:
            right = mid

    return left


N = 14
peak = find_peak(N)
print(f"The peak is at position {peak} with elevation {query(peak)}")
```

OUTPUT

```
The peak is at position 7 with elevation 49
```

QUESTION 2
DRY RUN

1. Problem Representation:

   Chromosome: Each chromosome represents a possible allocation of tasks to facilities. For example, a chromosome could be a list where each element corresponds to a task and the value at each position indicates the facility to which the task is assigned.

   Initial Population: Generate an initial population of chromosomes randomly, ensuring that each task is assigned to one of the facilities

2. Fitness Function:
   - Cost Calculation: Calculate the total cost for each chromosome by summing the costs of assigning each task to its allocated facility.
   - Capacity Constraint Check: Ensure that the total time allocated to each facility does not exceed its daily capacity. If a facility exceeds its capacity, penalize the fitness value to make the solution less favorable.
3. Genetic Operators:
   - Selection: Use Roulette Wheel Selection to choose chromosomes for reproduction based on their fitness.
   - Crossover: Perform one-point crossover on selected chromosomes to produce offspring.
   - Mutation: Apply swap mutation to introduce variability by swapping task allocations between facilities in a chromosome.
4. Termination Condition: The algorithm will run for a predefined number of generations or

   until a satisfactory solution is found.

CODE

```
import random


tasks = ['Task 1', 'Task 2', 'Task 3', 'Task 4', 'Task 5', 'Task 6', 'Task 7']

task_times = [5, 8, 4, 7, 6, 3, 9]

facilities = ['Facility 1', 'Facility 2', 'Facility 3']

facility_capacities = [24, 30, 28]

cost_matrix = [

    [10, 12, 9],

    [15, 14, 16],

    [8, 9, 7],

    [12, 10, 13],

    [14, 13, 12],

    [9, 8, 10],

    [11, 12, 13]

]



population_size = 6

crossover_rate = 0.8

mutation_rate = 0.2

generations = 100
```

```
def initialize_population():
    population = []
    for _ in range(population_size):
        chromosome = [random.randint(0, 2) for _ in range(len(tasks))]
        population.append(chromosome)
    return population


def calculate_fitness(chromosome):
    total_cost = 0
    facility_times = [0, 0, 0]
    for i in range(len(chromosome)):
        facility = chromosome[i]
        total_cost += cost_matrix[i][facility]
        facility_times[facility] += task_times[i]

    penalty = 0
    for i in range(len(facility_times)):
        if facility_times[i] > facility_capacities[i]:
            penalty += (facility_times[i] - facility_capacities[i]) * 100
    fitness = total_cost + penalty
    return fitness
```

```python
def roulette_wheel_selection(population, fitnesses):
    total_fitness = sum(fitnesses)
    if total_fitness == 0:
        return random.choices(population, k=2)
    probabilities = [f / total_fitness for f in fitnesses]
    selected = random.choices(population, weights=probabilities, k=2)
    return selected


def one_point_crossover(parent1, parent2):
    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
    else:
        return parent1, parent2


def swap_mutation(chromosome):
    if random.random() < mutation_rate:
        idx1, idx2 = random.sample(range(len(chromosome)), 2)
        chromosome[idx1], chromosome[idx2] = chromosome[idx2], chromosome[idx1]
```

```python
    return chromosome


def genetic_algorithm():
    population = initialize_population()
    for _ in range(generations):
        fitnesses = [calculate_fitness(chromosome) for chromosome in population]
        new_population = []
        for _ in range(population_size // 2):
            parent1, parent2 = roulette_wheel_selection(population, fitnesses)
            child1, child2 = one_point_crossover(parent1, parent2)
            child1 = swap_mutation(child1)
            child2 = swap_mutation(child2)
            new_population.extend([child1, child2])
        population = new_population
    best_chromosome = min(population, key=calculate_fitness)
    return best_chromosome


best_allocation = genetic_algorithm()
print("Best Allocation:", best_allocation)
print("Total Cost:", calculate_fitness(best_allocation))


facility_times = [0, 0, 0]
```

```
for i in range(len(best_allocation)):

    facility = best_allocation[i]

    facility_times[facility] += task_times[i]

print("Facility Times:", facility_times)

print("Facility Capacities:", facility_capacities)
```

OUTPUT

```
Best Allocation: [2, 2, 2, 2, 2, 2, 2]
Total Cost: 1480
Facility Times: [0, 0, 42]
Facility Capacities: [24, 30, 28]
```

QUESTION 3

```
import sys
from collections import deque

def read_sudoku():
    sudoku = []
    for _ in range(9):
        line = sys.stdin.readline().strip()
        row = []
        for c in line:
            if c == '.':
                row.append(0)
            else:
                row.append(int(c))
        sudoku.append(row)
    return sudoku

def print_sudoku(sudoku):
    for row in sudoku:
        print(''.join(map(str, row)))

def get_subgrid(sudoku, row, col):
    subgrid = []
    start_row = (row // 3) * 3
    start_col = (col // 3) * 3
    for i in range(3):
        for j in range(3):
            subgrid.append(sudoku[start_row + i][start_col + j])
    return subgrid

def is_valid(sudoku, row, col, num):
    # Check row
    if num in sudoku[row]:
        return False
    # Check column
    for i in range(9):
        if sudoku[i][col] == num:
            return False
    # Check subgrid
    subgrid = get_subgrid(sudoku, row, col)
```

```
    if num in subgrid:
        return False
    return True

def find_empty_cell(sudoku):
    for i in range(9):
        for j in range(9):
            if sudoku[i][j] == 0:
                return (i, j)
    return None

def solve_sudoku(sudoku):
    empty_cell = find_empty_cell(sudoku)
    if not empty_cell:
        return True
    row, col = empty_cell
    for num in range(1, 10):
        if is_valid(sudoku, row, col, num):
            sudoku[row][col] = num
            if solve_sudoku(sudoku):
                return True
            sudoku[row][col] = 0
    return False

def main():
    sudoku = read_sudoku()
    if solve_sudoku(sudoku):
        print_sudoku(sudoku)
    else:
        print("No solution exists")

if __name__ == "__main__":
    main()
```

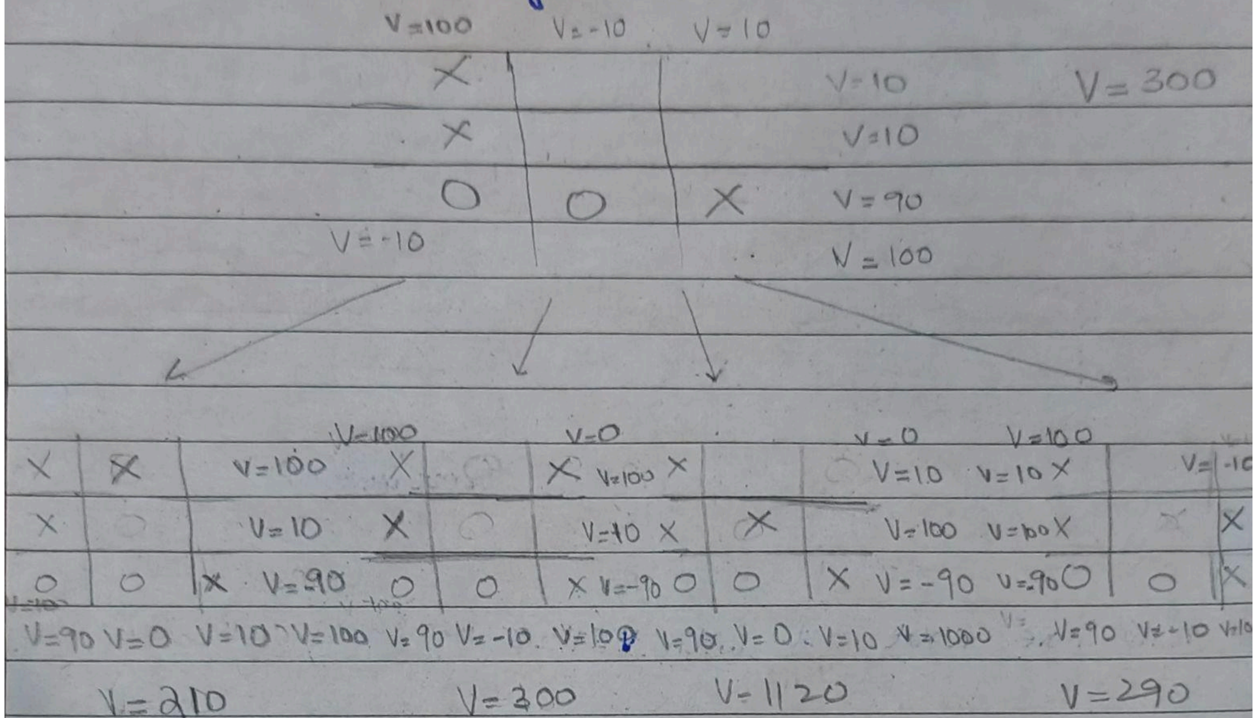
OUTPUT

```
.3.2.6..   | 967 | 345 | 821
9.3.5.1   | 251 | 876 | 493
.18.64..   |     |     |
.81.29..   | 548 | 132 | 976
7.1.8.8   | 729 | 564 | 138
.67.82..   | 136 | 798 | 245
.26.95..   | 372 | 689 | 514
8.2.3.9   | 814 | 253 | 769
.51.13..   | 695 | 417 | 382
```

```
532967481
963581247
418364529
681429753
729156438
356798214
247689315
894253176
175413862
```

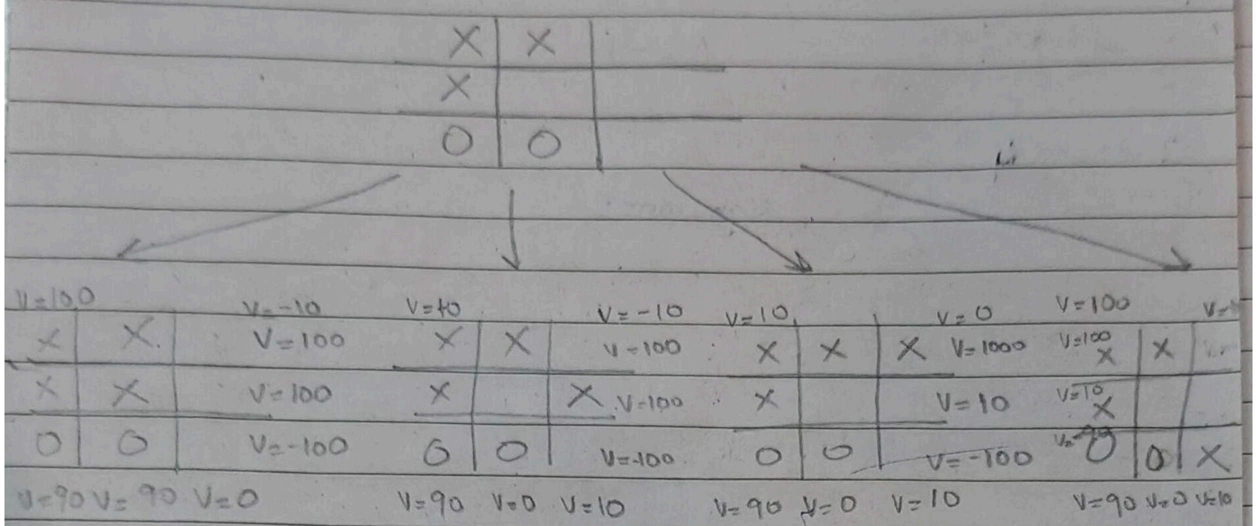
Question 4

Right most



best move

leftmost



V= 370 V=200 V=1020 V=210

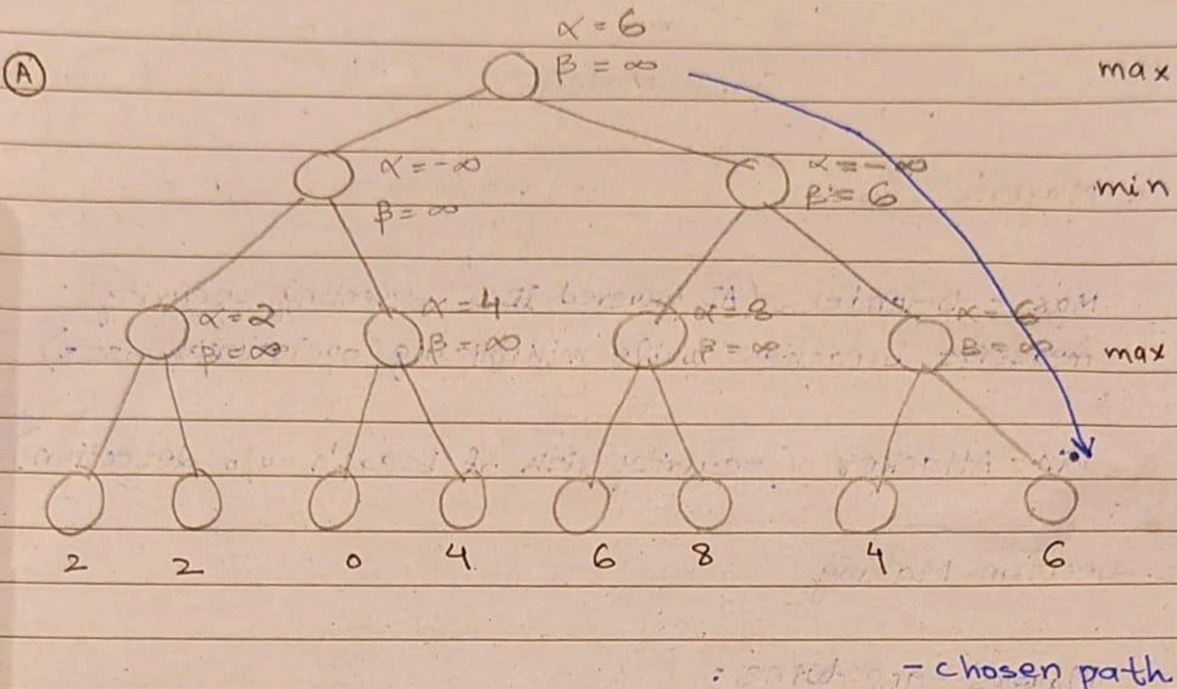
(best move)

Will move rightmost with V = 1120

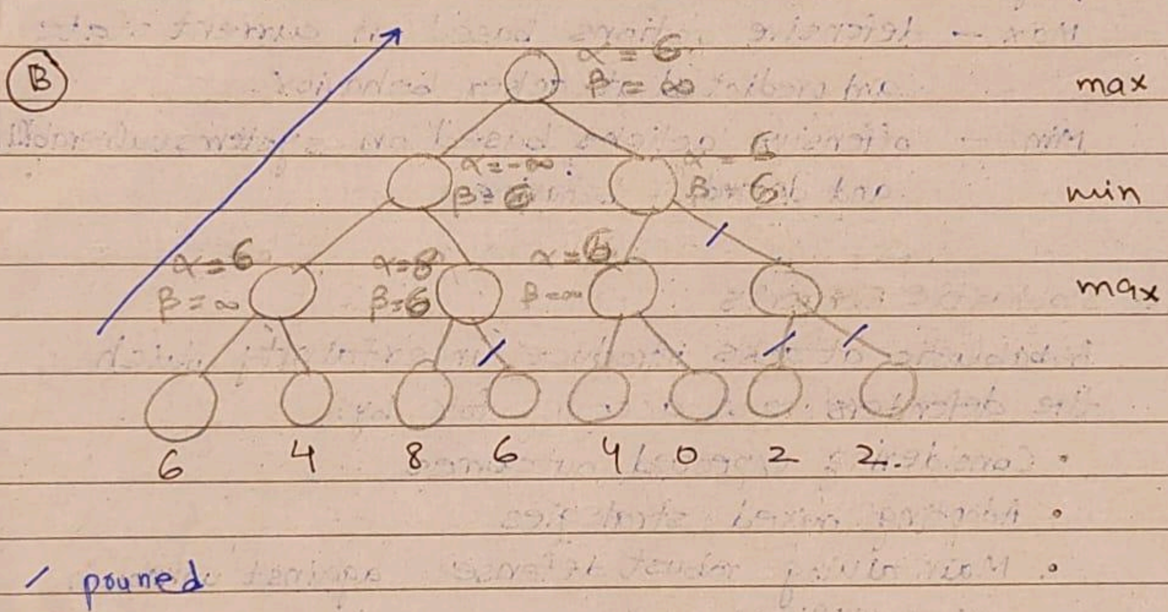
Date:

Question 5

(A)



(B)



Question 6

(a) Game Model

1. Players

Max - Defender (AI powered IDS preventing security ~~measures~~ breaches while minimizing operational costs)

Min - Attacker (security risk of breach w/o detection)

2. Decision-Making

Players take turns:

Max — defensive actions based on current state and predicted attacker behavior

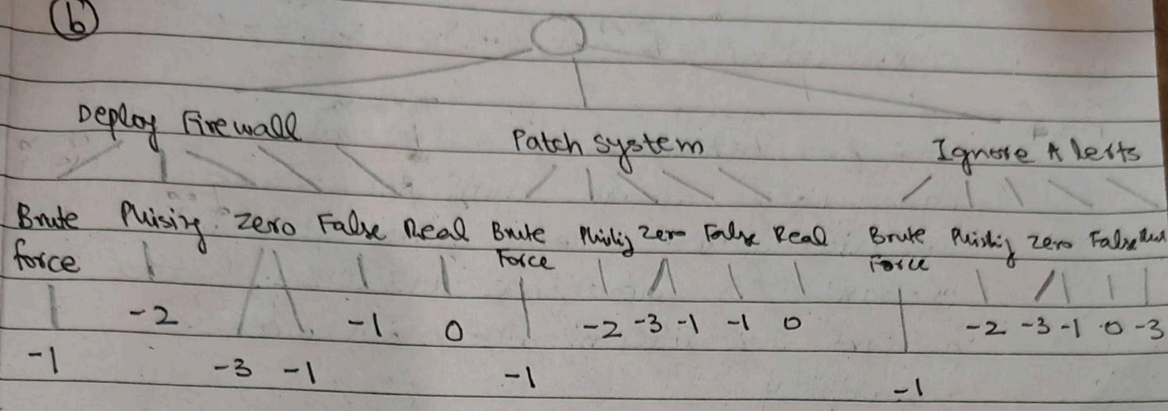
Min — offensive actions based on systems vulnerabilities and defender behavior

3. Stochastic Elements

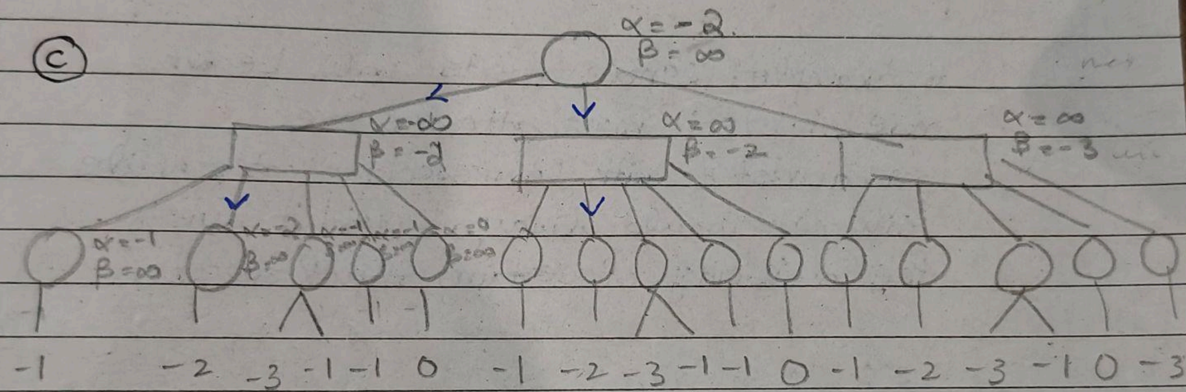
Probablistic attacks introduce uncertainity which the defenders must account for by:

- Considering expected outcomes
- Adopting mixed strategies
- Maintaining robust defenses against unknown vulnerabilities
- Increasing monitoring

(b)



(c)



Defender $\longrightarrow$ Deploy firewall $\longrightarrow$ Phishing
Defender $\longrightarrow$ Patch system $\longrightarrow$ Phishing

$$\text{expected value} = 0.5\underbrace{(-1)}_{\text{Attacker wins}} + 0.5\underbrace{(1)}_{\text{Defender wins}} = 0.$$

(d)

1. expected value = $0.5(-1) + 0.5(1) = 0$.

2. Defender considers expected values at chance nodes rather than worst-case scenarios which accepts some risk in exchange for operational efficiency, better account for different attack outcomes probability

For example: Ignore alerts might become more viable if most attacks are probabilistic with low success rates while Minimax would always avoid it due to the possibility of certain attack success.