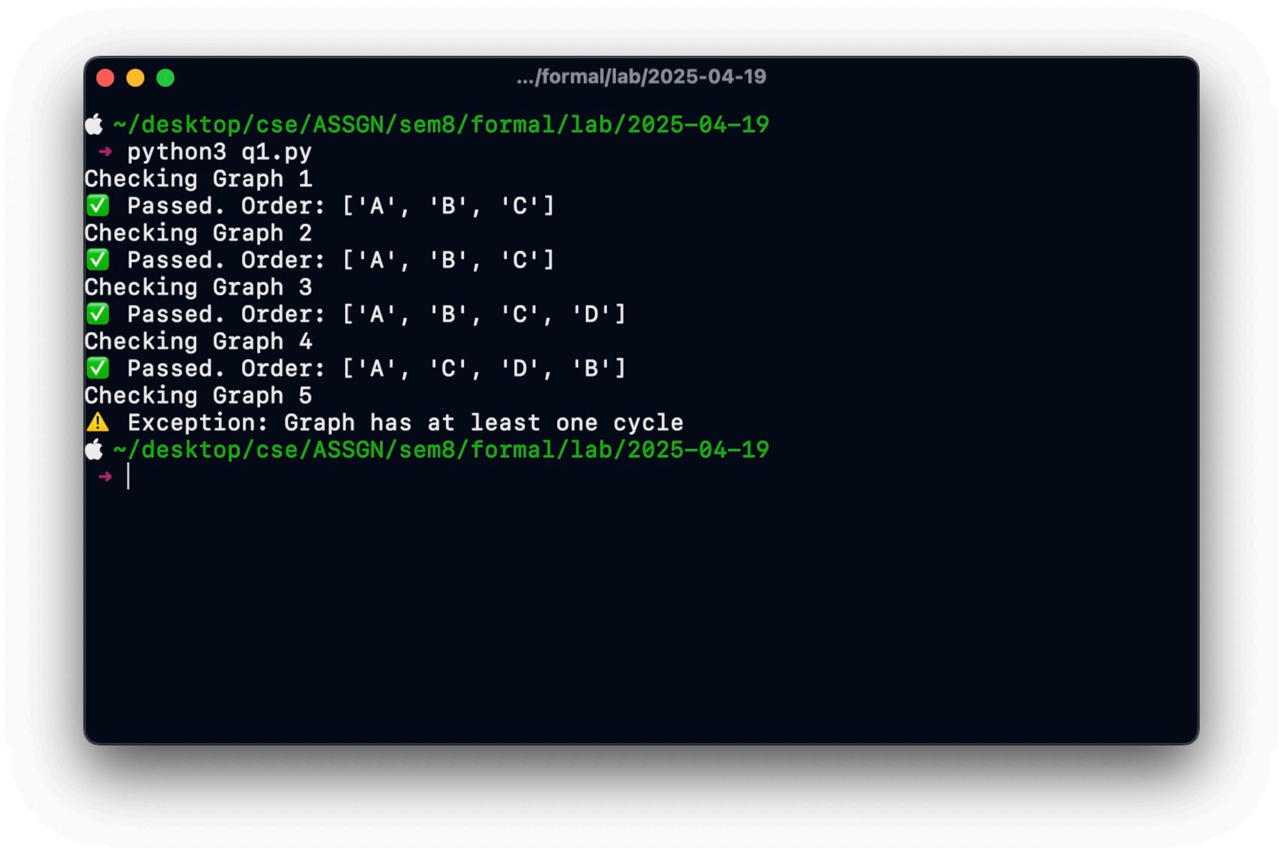
ASSIGNMENT 9
19/04/25

NAME : SHRESTH SONKAR
REGNO : 20214272
GROUP : CS8D
TOPIC : FORMAL METHOD LAB
CODE : CS-18201

```python
# Q1. Use model checking to verify the correctness of a
topological sorting algorithm.

from collections import deque

def topological_sort(graph):
    """
    Performs topological sorting using Kahn's
algorithm.
    Raises ValueError if the graph contains a cycle.
    """
    in_degree = {node: 0 for node in graph}
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    queue = deque([u for u in graph if in_degree[u] ==
0])
    result = []

    while queue:
        u = queue.popleft()
        result.append(u)
        for v in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)

    if len(result) != len(graph):
        raise ValueError("Graph has at least one
cycle")

    return result


def verify_topological_sort(graph, order):
    """
    Verifies that the topological order is valid for
the given graph.
    Raises AssertionError if the order is incorrect.
    """
    pos = {node: i for i, node in enumerate(order)}
    for u in graph:
        for v in graph[u]:
```

```python
            assert pos[u] < pos[v], f"Order invalid: 
{u} appears after {v}"


def run_model_check():
    test_graphs = [
        # Graph 1: Linear DAG
        {"A": ["B"], "B": ["C"], "C": []},

        # Graph 2: A splits to B and C
        {"A": ["B", "C"], "B": [], "C": []},

        # Graph 3: Diamond shape
        {"A": ["B", "C"], "B": ["D"], "C": ["D"], "D": 
[]},

        # Graph 4: Disconnected nodes
        {"A": ["B"], "B": [], "C": [], "D": []},

        # Graph 5: Cycle (should fail)
        {"A": ["B"], "B": ["C"], "C": ["A"]}
    ]

    for i, graph in enumerate(test_graphs):
        print(f"Checking Graph {i + 1}")
        try:
            order = topological_sort(graph)
            verify_topological_sort(graph, order)
            print(f"✅ Passed. Order: {order}")
        except AssertionError as e:
            print(f"❌ Assertion failed: {e}")
        except Exception as e:
            print(f"⚠️ Exception: {e}")


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




```
# Q2. Develop a proof of correctness for binary search
algorithm using Hoare Logic.

"""
Proof of Correctness for Binary Search using Hoare
Logic

Problem:
Given a sorted list 'arr' and a target value 'x', find
the index of 'x' in 'arr' using binary search.
Return -1 if 'x' is not present.

Precondition (P):
    arr is a list of elements sorted in non-decreasing
order.
    x is the target element.

Postcondition (Q):
    If x is in arr, return index i such that arr[i] ==
x.
    If x is not in arr, return -1.

Binary Search Loop Invariant (I):
    x is in arr[l..r] if it is in arr at all.

Termination:
```

```
    The interval [l, r] reduces each iteration. When l
> r, the loop stops.
"""

def binary_search(arr, x):
    l = 0
    r = len(arr) - 1

    while l <= r:
        mid = l + (r - l) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            r = mid - 1
        else:
            l = mid + 1

    return -1

if __name__ == "__main__":
    sorted_list = [1, 3, 5, 7, 9, 11, 13]
    target = 7
    result = binary_search(sorted_list, target)
    print(f"Index of {target}: {result}")
    target = 8
    result = binary_search(sorted_list, target)
    print(f"Index of {target}: {result}")
```

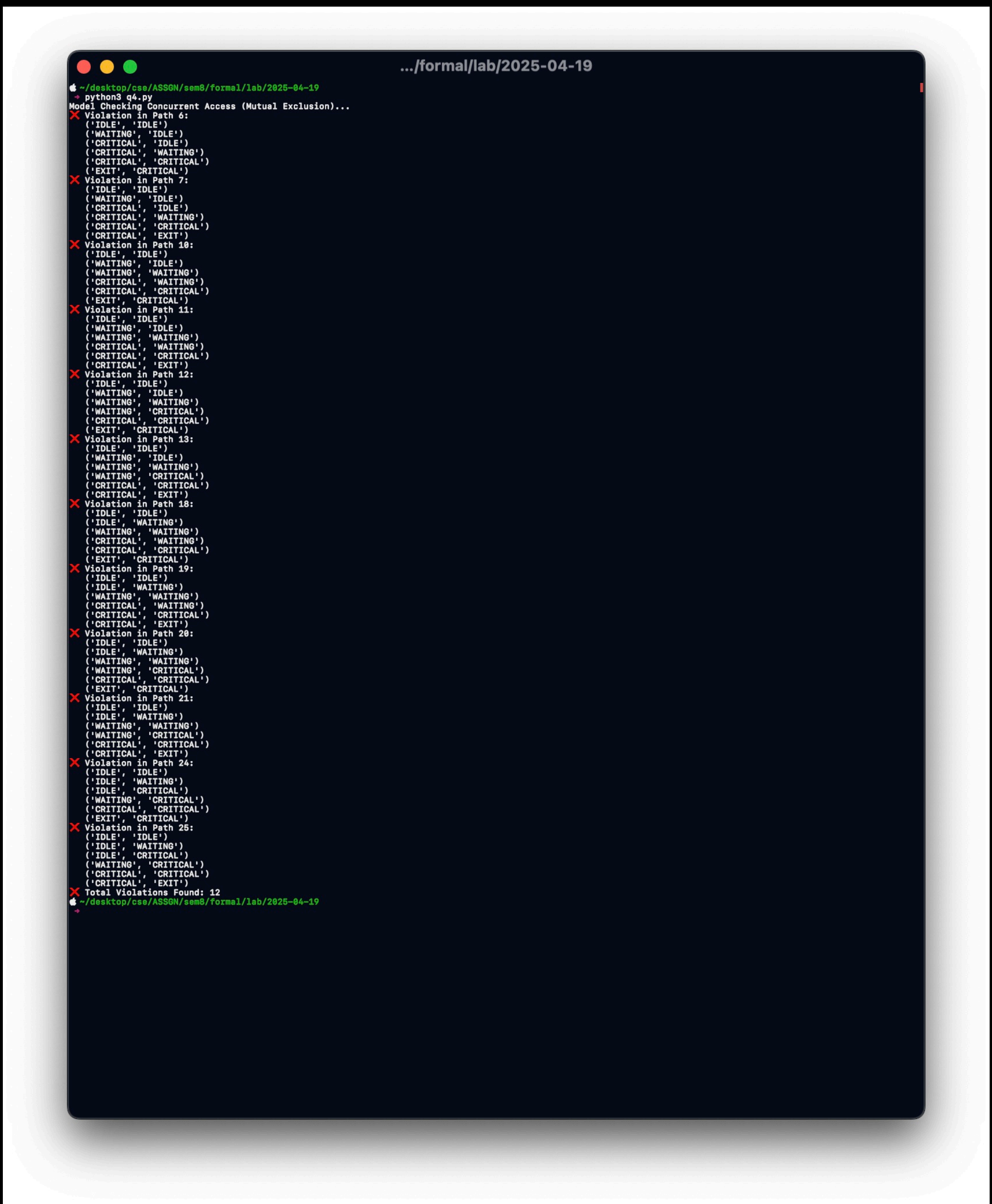
```
.../formal/lab/2025-04-19
 ~/desktop/cse/ASSGN/sem8/formal/lab/2025-04-19
 → python3 q2.py
Index of 7: 3
Index of 8: -1
 ~/desktop/cse/ASSGN/sem8/formal/lab/2025-04-19
 → |
```

```python
# Q3. Implement formal verification of loop invariants
for fixed point iterative algorithms.

import math

def g(x):
    return math.cos(x)

def is_invariant_preserved(x):
    return 0.0 <= x <= 1.0

def fixed_point_iteration(initial_guess,
tolerance=1e-7, max_iterations=100):
    x = initial_guess
    assert is_invariant_preserved(x), f"Initial guess
{x} violates loop invariant."

    for i in range(max_iterations):
        x_new = g(x)

        assert is_invariant_preserved(x_new), f"Loop
invariant violated at iteration {i+1}, x = {x_new}"

        if abs(x_new - x) < tolerance:
            print(f"Converged to {x_new} after {i+1}
iterations.")
            return x_new

        x = x_new

    raise Exception(f"Did not converge after
{max_iterations} iterations.")

def main():
    try:
        result =
fixed_point_iteration(initial_guess=0.5)
        print(f"Fixed-point result: {result}")
    except AssertionError as ae:
        print(f"Verification failed: {ae}")
    except Exception as e:
        print(e)

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


```python
# Q4. Among the formal methods — Model Checking (with Temporal Logic), Petri Nets, Process
# Algebra (e.g., CSP, CCS, π-Calculus), Theorem Proving (e.g., Coq, Isabelle, TLA+), and
# Abstract Interpretation (for static checking) — which is the most suitable for verifying
# concurrent access control mechanisms in a multi-threaded system? Justify your choice and
# demonstrate verification using the selected method.

# Why Model Checking (with Temporal Logic, e.g., TLA+)?
# 1.Concurrency Modeling: Model checking excels in handling the state-space explosion that occurs due to concurrency.
# 2.Access Control Properties: Safety (e.g., mutual exclusion) and liveness (e.g., progress, no deadlock) can be specified using temporal logic.
# 3.Automation: Model checking provides automated tools to exhaustively explore all possible thread interleavings.
# 4.Expressiveness: TLA+ (Temporal Logic of Actions) allows modeling concurrent processes and specifying high-level properties.
# 5.Practical Use: Used by industry giants like Amazon and Microsoft to verify real-world concurrent systems.

from itertools import product
```

```
IDLE = "IDLE"
WAITING = "WAITING"
CRITICAL = "CRITICAL"
EXIT = "EXIT"

thread_states = [IDLE, WAITING, CRITICAL, EXIT]

state_transitions = {
    IDLE: WAITING,
    WAITING: CRITICAL,
    CRITICAL: EXIT,
    EXIT: IDLE
}

def check_mutual_exclusion(path):
    for state in path:
        if state[0] == CRITICAL and state[1] ==
CRITICAL:
            return False
    return True

def generate_state_space(depth=3):
    initial = (IDLE, IDLE)
    paths = [[initial]]

    for _ in range(depth):
        new_paths = []
        for path in paths:
            curr = path[-1]
            for i in [0, 1]:
                new_state = list(curr)
                current_state = curr[i]
                next_state =
state_transitions.get(current_state)
                if next_state:
                    new_state[i] = next_state
                    new_paths.append(path +
[tuple(new_state)])
        paths = new_paths
    return paths

def main():
    print("Model Checking Concurrent Access (Mutual
Exclusion)...")
    all_paths = generate_state_space(depth=5)
```

```python
    violated = 0

    for i, path in enumerate(all_paths):
        if not check_mutual_exclusion(path):
            print(f"❌ Violation in Path {i}:")
            for state in path:
                print("  ", state)
            violated += 1

    if violated == 0:
        print("✅ No mutual exclusion violations
found.")
    else:
        print(f"❌ Total Violations Found: {violated}")

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



```
# Q5. Develop a formal specification for a job
scheduling system and verify correctness.

from typing import List
from dataclasses import dataclass

@dataclass
class Job:
    job_id: int
    arrival_time: int
    burst_time: int

@dataclass
class ScheduledJob:
    job_id: int
    start_time: int
    finish_time: int

def sjf_schedule(jobs: List[Job]) ->
List[ScheduledJob]:
    time = 0
    scheduled_jobs = []
    remaining_jobs = sorted(jobs, key=lambda x:
(x.arrival_time, x.burst_time, x.job_id))

    while remaining_jobs:
        available_jobs = [job for job in remaining_jobs
if job.arrival_time <= time]

        if not available_jobs:
            time = remaining_jobs[0].arrival_time
            continue

        next_job = min(available_jobs, key=lambda x:
(x.burst_time, x.arrival_time, x.job_id))
        remaining_jobs.remove(next_job)

        start_time = max(time, next_job.arrival_time)
        finish_time = start_time + next_job.burst_time

scheduled_jobs.append(ScheduledJob(next_job.job_id,
start_time, finish_time))

        time = finish_time
```

```
    return scheduled_jobs

def verify_schedule(jobs: List[Job], schedule:
List[ScheduledJob]) -> None:
    job_ids = set(job.job_id for job in jobs)
    scheduled_ids = set(job.job_id for job in schedule)
    assert job_ids == scheduled_ids, "Not all jobs
scheduled correctly"

    for i in range(len(schedule)):
        for j in range(i + 1, len(schedule)):
            assert schedule[i].finish_time <=
schedule[j].start_time or schedule[j].finish_time <=
schedule[i].start_time, \
                f"Jobs {schedule[i].job_id} and
{schedule[j].job_id} overlap"

    job_map = {job.job_id: job for job in jobs}
    for sched in schedule:
        assert sched.start_time >=
job_map[sched.job_id].arrival_time, \
            f"Job {sched.job_id} started before
arrival"

    print("All verification checks passed!")

if __name__ == "__main__":
    job_list = [
        Job(job_id=1, arrival_time=0, burst_time=8),
        Job(job_id=2, arrival_time=1, burst_time=4),
        Job(job_id=3, arrival_time=2, burst_time=9),
        Job(job_id=4, arrival_time=3, burst_time=5)
    ]

    schedule = sjf_schedule(job_list)
    for job in schedule:
        print(f"Job {job.job_id}: Start
{job.start_time}, Finish {job.finish_time}")

    verify_schedule(job_list, schedule)
```

…/formal/lab/2025-04-19

```
 ~/desktop/cse/ASSGN/sem8/formal/lab/2025-04-19
 ➜ python3 q5.py
Job 1: Start 0, Finish 8
Job 2: Start 8, Finish 12
Job 4: Start 12, Finish 17
Job 3: Start 17, Finish 26
All verification checks passed!
 ~/desktop/cse/ASSGN/sem8/formal/lab/2025-04-19
 ➜ |
```