University of Bahrain
College of information technology
Department of Computer Engineering

ITNE483

ITCE464: INTELLIGENT SYSTEMS LAB-2

Name :
zainab Abdali **ID** : 202005678

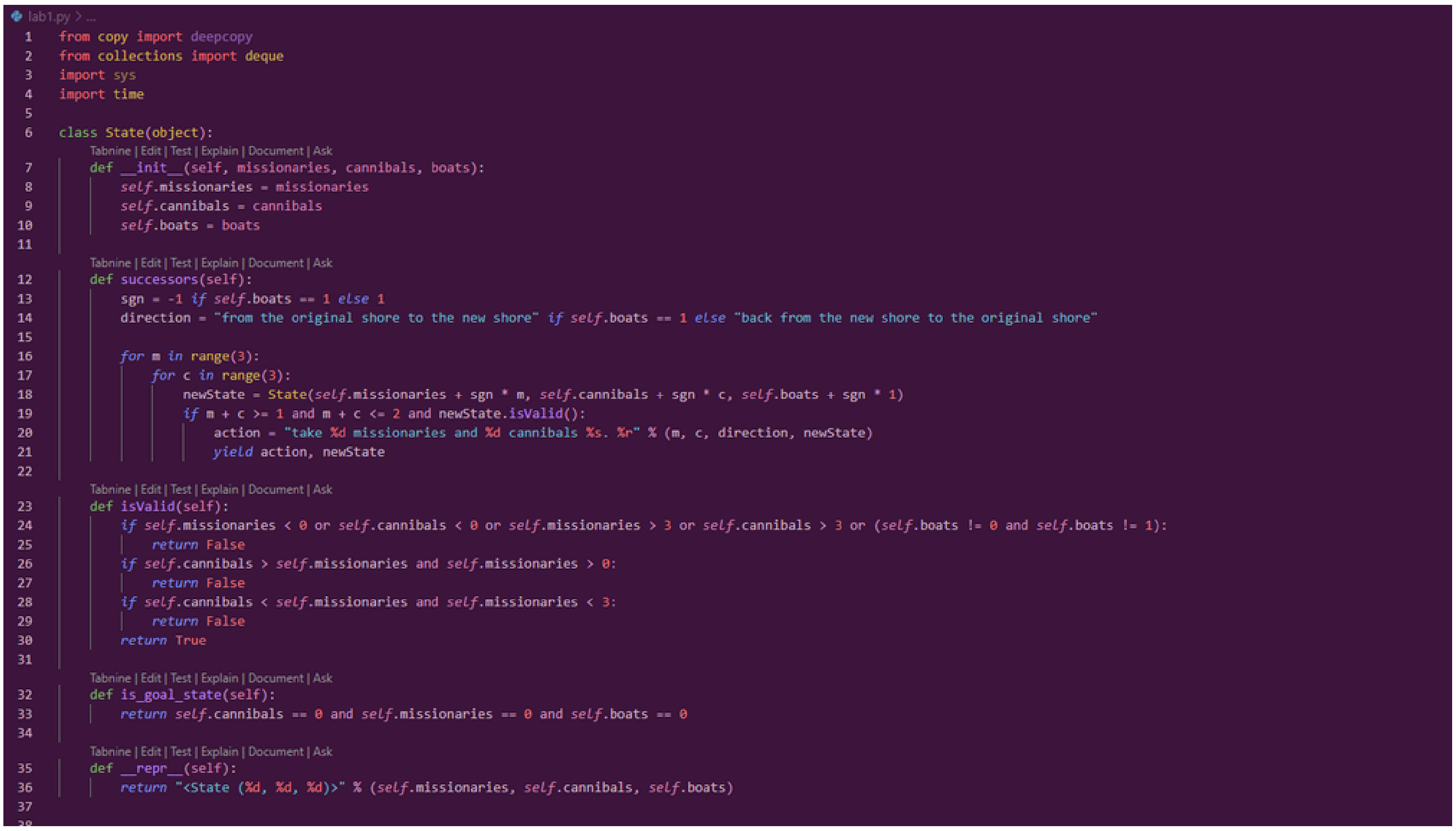
Introduction to Search Problems

In artificial intelligence and problem-solving, search issues are essential. An agent must identify a path, or series of activities, that will take it from a starting state to a desired objective state in a search issue. This entails investigating potential states or configurations of the issue domain, determining whether each state satisfies the requirements to achieve the objective, and monitoring the optimal route discovered.

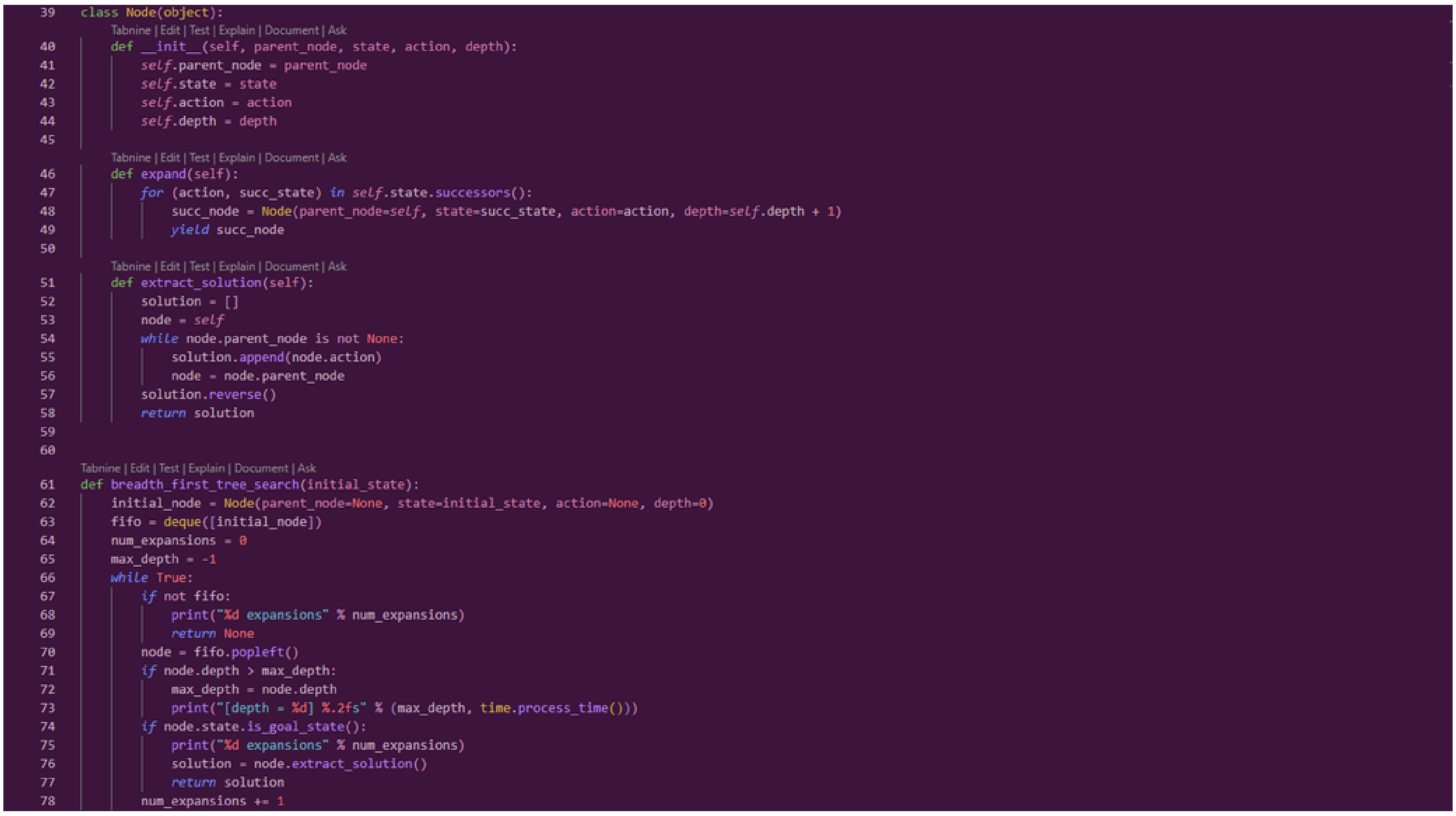
Different kinds of search tactics can be divided into the following categories:

- Uninformed Search: These searches just contain the problem specification and no other information about the objective. Breadth-first search (BFS) and depth-first search (DFS) are two examples.
- Informed Search: These more efficiently direct the search by using heuristics or knowledge unique to the problem. The greedy search and A* are two examples.

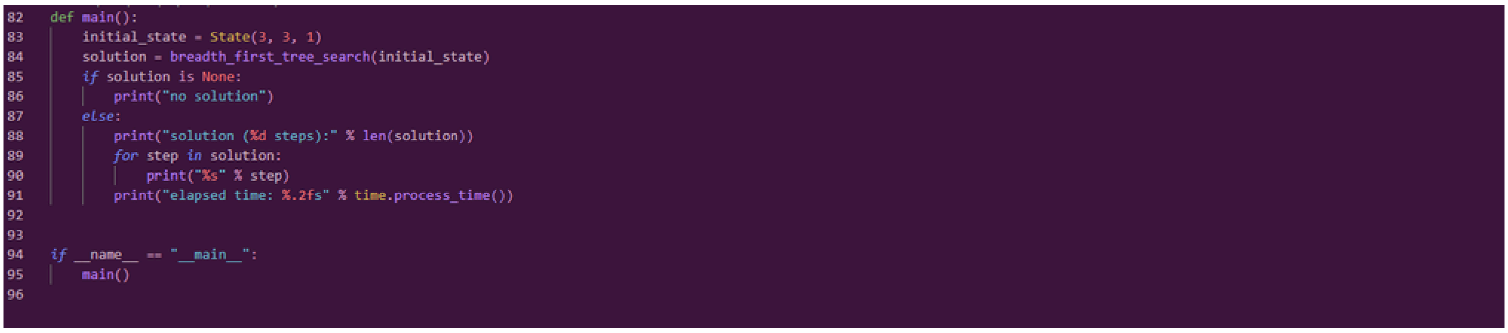
1.Missionaries and Cannibals Problem



```python
from copy import deepcopy
from collections import deque
import sys
import time

class State(object):
    def __init__(self, missionaries, cannibals, boats):
        self.missionaries = missionaries
        self.cannibals = cannibals
        self.boats = boats

    def successors(self):
        sgn = -1 if self.boats == 1 else 1
        direction = "from the original shore to the new shore" if self.boats == 1 else "back from the new shore to the original shore"

        for m in range(3):
            for c in range(3):
                newState = State(self.missionaries + sgn * m, self.cannibals + sgn * c, self.boats + sgn * 1)
                if m + c >= 1 and m + c <= 2 and newState.isValid():
                    action = "take %d missionaries and %d cannibals %s. %r" % (m, c, direction, newState)
                    yield action, newState

    def isValid(self):
        if self.missionaries < 0 or self.cannibals < 0 or self.missionaries > 3 or self.cannibals > 3 or (self.boats != 0 and self.boats != 1):
            return False
        if self.cannibals > self.missionaries and self.missionaries > 0:
            return False
        if self.cannibals < self.missionaries and self.missionaries < 3:
            return False
        return True

    def is_goal_state(self):
        return self.cannibals == 0 and self.missionaries == 0 and self.boats == 0

    def __repr__(self):
        return "<State (%d, %d, %d)>" % (self.missionaries, self.cannibals, self.boats)
```



```python
class Node(object):
    def __init__(self, parent_node, state, action, depth):
        self.parent_node = parent_node
        self.state = state
        self.action = action
        self.depth = depth

    def expand(self):
        for (action, succ_state) in self.state.successors():
            succ_node = Node(parent_node=self, state=succ_state, action=action, depth=self.depth + 1)
            yield succ_node

    def extract_solution(self):
        solution = []
        node = self
        while node.parent_node is not None:
            solution.append(node.action)
            node = node.parent_node
        solution.reverse()
        return solution


def breadth_first_tree_search(initial_state):
    initial_node = Node(parent_node=None, state=initial_state, action=None, depth=0)
    fifo = deque([initial_node])
    num_expansions = 0
    max_depth = -1
    while True:
        if not fifo:
            print("%d expansions" % num_expansions)
            return None
        node = fifo.popleft()
        if node.depth > max_depth:
            max_depth = node.depth
            print("[depth = %d] %.2fs" % (max_depth, time.process_time()))
        if node.state.is_goal_state():
            print("%d expansions" % num_expansions)
            solution = node.extract_solution()
            return solution
        num_expansions += 1
```



```python
def main():
    initial_state = State(3, 3, 1)
    solution = breadth_first_tree_search(initial_state)
    if solution is None:
        print("no solution")
    else:
        print("solution (%d steps):" % len(solution))
        for step in solution:
            print("%s" % step)
        print("elapsed time: %.2fs" % time.process_time())


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

Problem Description
One of the classic search problems is Missionaries and Cannibals. In it, three cannibals and three missionaries must use a boat to cross a river. Transporting every person across without breaking any rules is the aim:

- There can only be two passengers on the boat at once.
- On either side of the river, there cannot be more cannibals than missionaries since doing so would put the missionaries in danger.

This issue can be represented as a search problem in which the agent has to figure out a series of actions that will safely carry every person over the river.

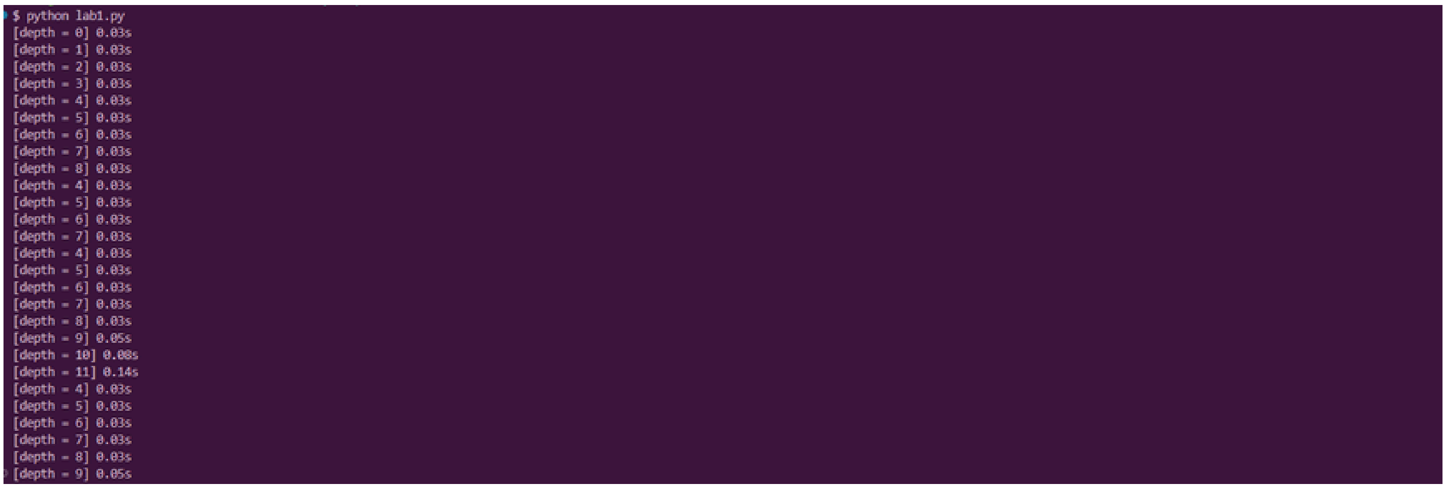
Analysis of Code
The given code uses the breadth-first search (BFS) algorithm to tackle the issue. All missionaries, cannibals, and the boat are on the original shore in the initial state. The code consists of:

- State Representation: A tuple represents the position of the boat, the number of missionaries, and the number of cannibals.
- Valid Actions: By moving a predetermined number of missionaries and cannibals in each direction, the successors function creates potential movements.
- Goal Check: The is_goal_state function determines whether or not all cannibals and missionaries have arrived at the new coast.

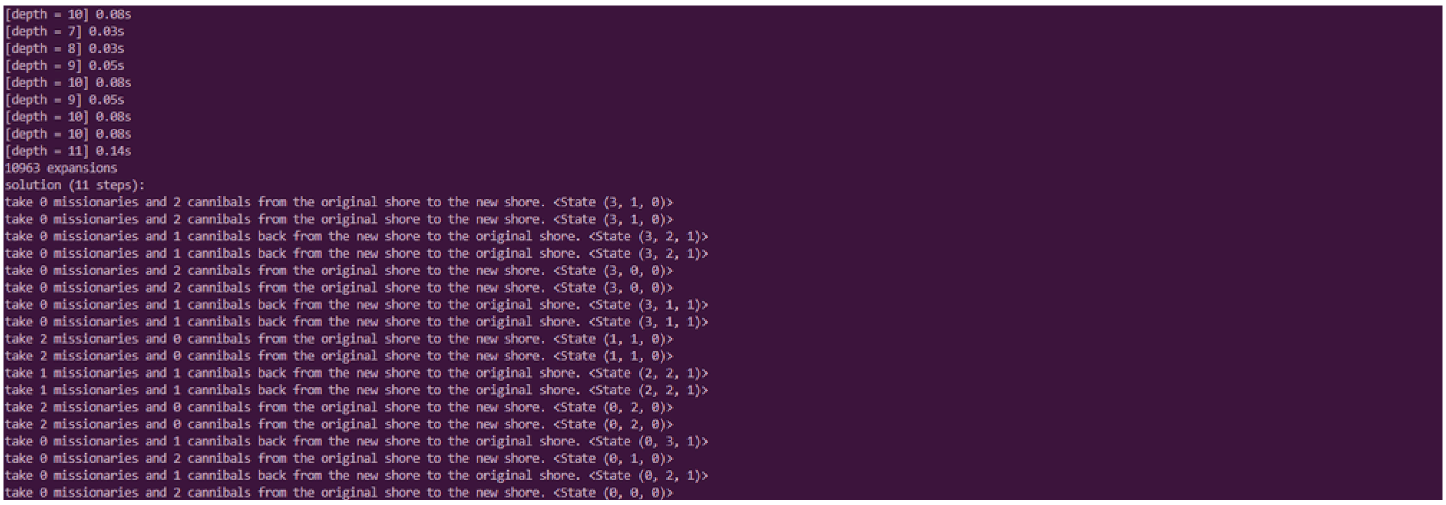
In order to identify the shortest solution to the problem, the BFS algorithm investigates every potential state one level at a time.

Search Type Used
To determine the shortest path between the beginning and the objective states, the code uses breadth-first search (BFS). Since BFS ensures the shortest solution when all actions have the same cost, it is a good fit for this problem. Additionally, BFS makes sure that every potential state is only investigated once, which makes it effective for simpler issues.

the output

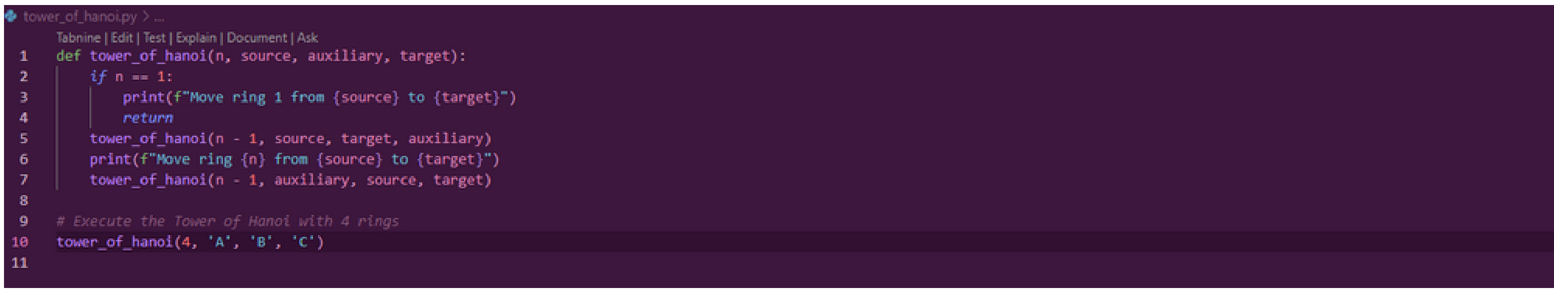


```
$ python lab1.py
 [depth = 0] 0.03s
 [depth = 1] 0.03s
 [depth = 2] 0.03s
 [depth = 3] 0.03s
 [depth = 4] 0.03s
 [depth = 5] 0.03s
 [depth = 6] 0.03s
 [depth = 7] 0.03s
 [depth = 8] 0.03s
 [depth = 4] 0.03s
 [depth = 5] 0.03s
 [depth = 6] 0.03s
 [depth = 7] 0.03s
 [depth = 4] 0.03s
 [depth = 5] 0.03s
 [depth = 6] 0.03s
 [depth = 7] 0.03s
 [depth = 8] 0.03s
 [depth = 9] 0.05s
 [depth = 10] 0.08s
 [depth = 11] 0.14s
 [depth = 4] 0.03s
 [depth = 5] 0.03s
 [depth = 6] 0.03s
 [depth = 7] 0.03s
 [depth = 8] 0.03s
 [depth = 9] 0.05s
```



```
[depth = 10] 0.08s
[depth = 7] 0.03s
[depth = 8] 0.03s
[depth = 9] 0.05s
[depth = 10] 0.08s
[depth = 9] 0.05s
[depth = 10] 0.08s
[depth = 10] 0.08s
[depth = 11] 0.14s
10963 expansions
solution (11 steps):
take 0 missionaries and 2 cannibals from the original shore to the new shore. <State (3, 1, 0)>
take 0 missionaries and 2 cannibals from the original shore to the new shore. <State (3, 1, 0)>
take 0 missionaries and 1 cannibals back from the new shore to the original shore. <State (3, 2, 1)>
take 0 missionaries and 1 cannibals back from the new shore to the original shore. <State (3, 2, 1)>
take 0 missionaries and 2 cannibals from the original shore to the new shore. <State (3, 0, 0)>
take 0 missionaries and 2 cannibals from the original shore to the new shore. <State (3, 0, 0)>
take 0 missionaries and 1 cannibals back from the new shore to the original shore. <State (3, 1, 1)>
take 0 missionaries and 1 cannibals back from the new shore to the original shore. <State (3, 1, 1)>
take 2 missionaries and 0 cannibals from the original shore to the new shore. <State (1, 1, 0)>
take 2 missionaries and 0 cannibals from the original shore to the new shore. <State (1, 1, 0)>
take 1 missionaries and 1 cannibals back from the new shore to the original shore. <State (2, 2, 1)>
take 1 missionaries and 1 cannibals back from the new shore to the original shore. <State (2, 2, 1)>
take 2 missionaries and 0 cannibals from the original shore to the new shore. <State (0, 2, 0)>
take 2 missionaries and 0 cannibals from the original shore to the new shore. <State (0, 2, 0)>
take 0 missionaries and 1 cannibals back from the new shore to the original shore. <State (0, 3, 1)>
take 0 missionaries and 2 cannibals from the original shore to the new shore. <State (0, 1, 0)>
take 0 missionaries and 1 cannibals back from the new shore to the original shore. <State (0, 2, 1)>
take 0 missionaries and 2 cannibals from the original shore to the new shore. <State (0, 0, 0)>
```

2. Tower of Hanoi Problem



```
tower_of_hanoi.py > ...
Tabnine | Edit | Test | Explain | Document | Ask
def tower_of_hanoi(n, source, auxiliary, target):
    if n == 1:
        print(f"Move ring 1 from {source} to {target}")
        return
    tower_of_hanoi(n - 1, source, target, auxiliary)
    print(f"Move ring {n} from {source} to {target}")
    tower_of_hanoi(n - 1, auxiliary, source, target)

# Execute the Tower of Hanoi with 4 rings
tower_of_hanoi(4, 'A', 'B', 'C')
```

Problem Description
Three rods and several rings of varying diameters are built on top of one rod in descending order to form the Tower of Hanoi, a mathematical puzzle. Using an auxiliary rod as a bridge, the goal is to transfer every ring from the beginning rod to a target rod. The difficulty is in following two guidelines:

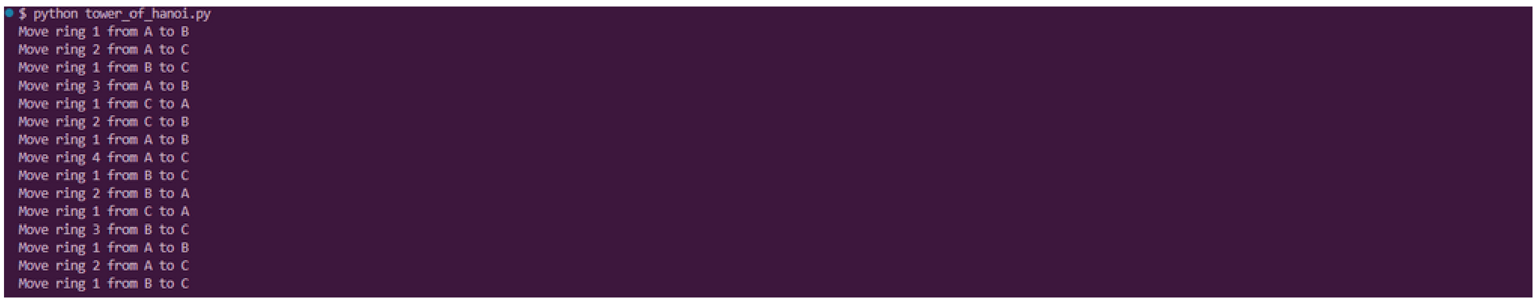
- You can only move one ring at a time.
- You can't put a bigger ring on top of a smaller one.

Recursion is frequently used to tackle this issue; at each step, a subset of rings is moved to the desired position using a divide-and-conquer strategy.

This code demonstrates a recursive approach to solving the problem, where:

From the source to the auxiliary rod, the function **tower_of_hanoi** transfers **n-1** rings.
The **n**th (biggest) ring is moved to the target rod.
The n-1 rings are then transferred from the auxiliary rod to the target rod.

output:



```
$ python tower_of_hanoi.py
Move ring 1 from A to B
Move ring 2 from A to C
Move ring 1 from B to C
Move ring 3 from A to B
Move ring 1 from C to A
Move ring 2 from C to B
Move ring 1 from A to B
Move ring 4 from A to C
Move ring 1 from B to C
Move ring 2 from B to A
Move ring 1 from C to A
Move ring 3 from B to C
Move ring 1 from A to B
Move ring 2 from A to C
Move ring 1 from B to C
```

Conclusion:

The Tower of Hanoi and the Missionaries and Cannibals dilemma are two well-known search problems that were examined in this lab project. Both needed the application of solutions that methodically investigate potential situations in order to arrive at a predetermined objective:

- A breadth-first search was used in the Missionaries and Cannibals issue to determine the shortest path to a destination.
- Recursion, which is intrinsically depth-first since it investigates every path in the recursive tree to arrive at the solution, was used to solve the Tower of Hanoi issue.

Through this lab, I learned how to apply these concepts to structured problem-solving, comprehend the differences between BFS and recursive depth-first algorithms, and use Python to solve AI search issues.