Aim:

To write and execute the python program for theriver crozzing puzzle.

Procedure:

1. **Define State Representation:**
   * The state of the problem is represented as a tuple **(missionaries left, cannibals left, boat, missionaries right, cannibals right)**.
2. **Define Validity Check:**
   * The function **is\_valid(s)** checks if a given state **s** is valid based on the problem constraints, such as the number of missionaries and cannibals on each side of the river.
3. **Define Successor Generation:**
   * The function **get\_successors(s)** generates valid successor states from a given state **s** by applying possible transitions. It considers moving missionaries and cannibals across the river in different combinations.
4. **Breadth-First Search (BFS):**
   * The **solve()** function performs a BFS starting from the initial state **(3, 3, 1, 0, 0)** to reach the goal state **(0, 0, 0, 3, 3)**.
   * It uses a deque (**q**) to keep track of the states to explore and a set (**seen**) to avoid revisiting already explored states.
5. **Explore States:**
   * The BFS explores states, generating successors and adding them to the queue for further exploration.
   * The process continues until the goal state is reached or all possible states are explored.
6. **Print Solution:**
   * The **print\_solution(solution)** function prints the solution path if a solution is found.
   * It displays each state in the path, indicating the number of missionaries and cannibals on both sides of the river and the boat's position.
7. **Display Result:**
   * The main part of the code calls **solve()** to find a solution and then prints the solution path using **print\_solution(solution)**.

Code:

from collections import deque

def is\_valid(s):

# s: (missionaries left, cannibals left, boat, missionaries right, cannibals right)

return all(0 <= x <= 3 for x in s[:5]) and (s[0] >= s[1] or s[0] == 0) and (s[3] >= s[4] or s[3] == 0)

def get\_successors(s):

transitions = [(-1, -1, -1, 1, 1), (-1, 0, -1, 1, 0), (0, -1, -1, 0, 1), (-2, 0, -1, 2, 0), (0, -2, -1, 0, 2)]

if s[2] == 0: # Adjust transitions for opposite boat direction

transitions = [(x[3], x[4], 1, x[0], x[1]) for x in transitions]

return [(s[0]+m, s[1]+c, (s[2]+b) % 2, s[3]-m, s[4]-c) for m, c, b, \_, \_ in transitions if is\_valid((s[0]+m, s[1]+c, (s[2]+b) % 2, s[3]-m, s[4]-c))]

def solve():

start, goal = (3, 3, 1, 0, 0), (0, 0, 0, 3, 3)

q = deque([(start, [])])

seen = set([start])

while q:

state, path = q.popleft()

if state == goal:

return path + [goal]

for next\_state in get\_successors(state):

if next\_state not in seen:

seen.add(next\_state)

q.append((next\_state, path + [state]))

return None

def print\_solution(solution):

if not solution:

print("No solution.")

else:

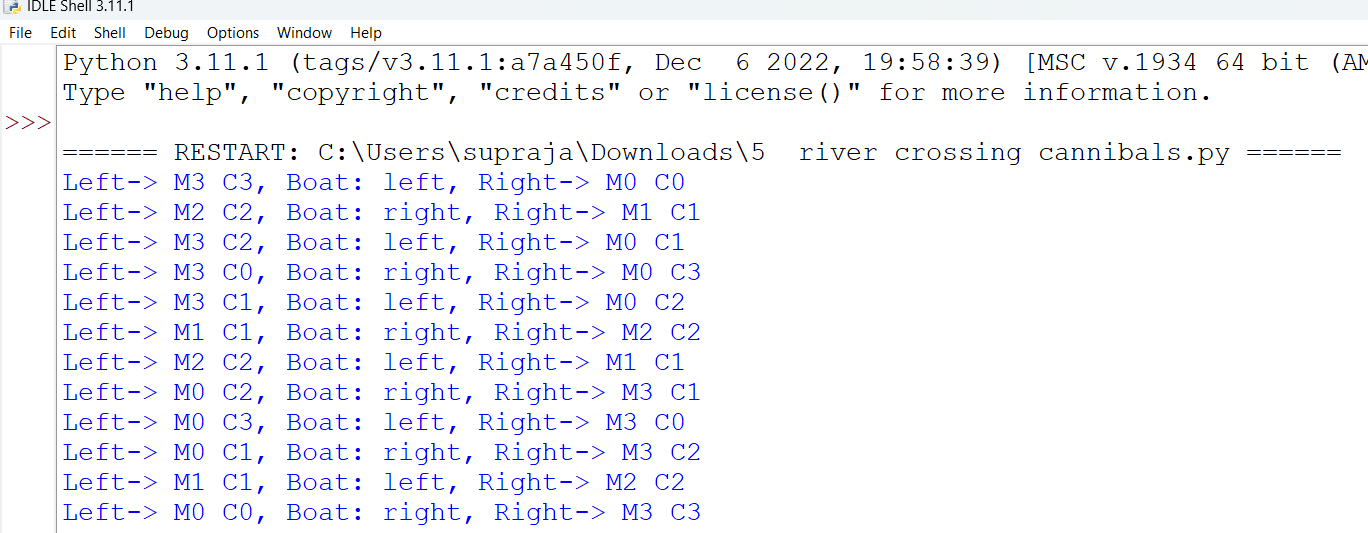
for s in solution:

print(f"Left-> M{s[0]} C{s[1]}, Boat: {'left' if s[2] else 'right'}, Right-> M{s[3]} C{s[4]}")

solution = solve()

print\_solution(solution)

output:



Result:

Hence the program has been successfully executed and verified.