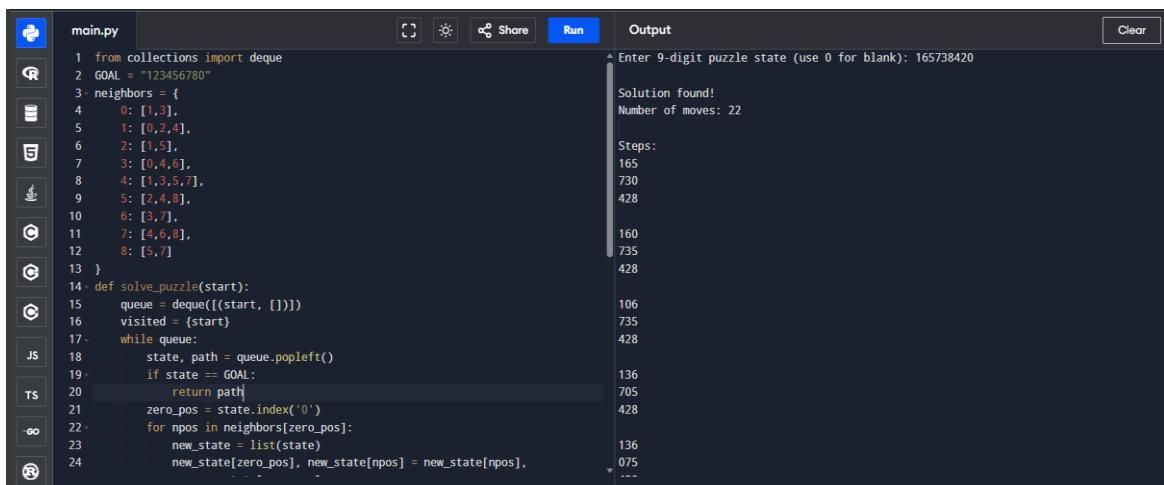


Python Programs – Day 1

- In sliding puzzle, it consists of a 3x3 grid with eight numbered tiles and one empty space. The goal of the puzzle is to rearrange the tiles from their initial, scrambled state to a goal state where the numbers are ordered from 1 to 8, with the empty space in the bottom-right corner. Implement the same using python.



```

main.py | Run | Output | Clear
1  from collections import deque
2  GOAL = "123456780"
3  neighbors = {
4      0: [1,3],
5      1: [0,2,4],
6      2: [1,5],
7      3: [0,4,6],
8      4: [1,3,5,7],
9      5: [2,4,8],
10     6: [3,7],
11     7: [4,6,8],
12     8: [5,7]
13 }
14 def solve_puzzle(start):
15     queue = deque([(start, [])])
16     visited = {start}
17     while queue:
18         state, path = queue.popleft()
19         if state == GOAL:
20             return path
21         zero_pos = state.index('0')
22         for npos in neighbors[zero_pos]:
23             new_state = list(state)
24             new_state[zero_pos], new_state[npos] = new_state[npos], ...
    
```

Enter 9-digit puzzle state (use 0 for blank): 165738420
 Solution found!
 Number of moves: 22
 Steps:
 165
 730
 428
 160
 735
 428
 106
 735
 428
 136
 705
 428
 136
 075
 ...

- The Towers of Hanoi problem involves three pegs (A, B, C) and a number of disks of different sizes that can slide onto any peg. The puzzle starts with all the disks stacked on one peg in order of decreasing size, with the largest at the bottom. The objective of the problem is to move the entire stack to another peg, following these rules:
 - Only one disk can be moved at a time.
 - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
 - No disk may be placed on top of a smaller disk.
- Implement the above Towers of Hanoi problem using Python.

The screenshot shows a Jupyter Notebook interface with a dark theme. On the left is a sidebar with icons for various file types: Python (selected), QR code, CSV, JSON, PDF, CSS, JS, TS, Go, and Docker. The main area contains a code cell titled "main.py" with the following Python code:

```

1 def towers_of_hanoi(n, source, auxiliary, destination):
2     if n == 1:
3         print(f"Move disk 1 from {source} --> {destination}")
4         return
5     |
6     towers_of_hanoi(n-1, source, destination, auxiliary)
7     |
8     print(f"Move disk {n} from {source} --> {destination}")
9     |
10    towers_of_hanoi(n-1, auxiliary, source, destination)
11
12
13 num = int(input("Enter number of disks: "))
14
15 print(f"\nSolution for {num} disks:\n")
16 towers_of_hanoi(num, 'A', 'B', 'C')
17

```

The "Run" button is highlighted in blue. To the right is the "Output" pane, which displays the execution results:

```

Enter number of disks: 2
Solution for 2 disks:
Move disk 1 from A --> B
Move disk 2 from A --> C
Move disk 1 from B --> C
== Code Execution Successful ==

```

3. Implement a Python program to perform Breadth-First Search (BFS) on a graph. The program should:
 - a) Represent the graph using an adjacency list.
 - b) Start the traversal from a given source node.
 - c) Print the order in which the nodes are visited.

The screenshot shows a Jupyter Notebook interface with a dark theme. On the left is a sidebar with icons for various file types: Python (selected), QR code, CSV, JSON, PDF, CSS, JS, TS, Go, and Docker. The main area contains a code cell titled "main.py" with the following Python code:

```

1 from collections import deque
2 def bfs(graph, start):
3     visited = set()
4     queue = deque([start])
5     visited.add(start)
6     print("BFS Traversal:", end=" ")
7     while queue:
8         node = queue.popleft()
9         print(node, end=" ")
10        for neighbor in graph[node]:
11            if neighbor not in visited:
12                visited.add(neighbor)
13                queue.append(neighbor)
14
15 graph = [
16     'A': ['B', 'C'],
17     'B': ['D', 'E'],
18     'C': ['F'],
19     'D': [],
20     'E': ['F'],
21     'F': []
22 ]
23 start_node = input("Enter starting node: ")
24

```

The "Run" button is highlighted in blue. To the right is the "Output" pane, which displays the execution results:

```

Enter number of disks: 2
Solution for 2 disks:
Move disk 1 from A --> B
Move disk 2 from A --> C
Move disk 1 from B --> C
== Code Execution Successful ==

```

4. The Monkey and Banana problem is a classic artificial intelligence problem where a monkey needs to navigate through a room to reach a bunch of bananas hanging from the ceiling. The monkey has to move a box to stand on it to reach the bananas. Implement the above scenario using python.

The screenshot shows a Python code editor interface with a dark theme. On the left is the code editor pane containing `main.py`, which implements a breadth-first search to solve the Monkey and Banana problem. On the right is the output pane showing the steps of the solution and a success message.

```

main.py
1 from collections import deque
2 def get_next_states(state):
3     monkey, box, on_box = state
4     states = []
5     if not on_box:
6         for pos in ["door", "window", "middle"]:
7             if pos != monkey:
8                 states.append((pos, box, False))
9     if not on_box and monkey == box:
10        for pos in ["door", "window", "middle"]:
11            if pos != monkey:
12                states.append((pos, pos, False))
13    if not on_box and monkey == box:
14        states.append((monkey, box, True))
15    if on_box:
16        states.append((monkey, box, False))
17    return states
18 def solve_monkey_banana():
19     start = ("door", "window", False)
20     goal = ("middle", "middle", True)
21     queue = deque([(start, [])])
22     visited = {start}
23     while queue:
24         state, path = queue.popleft()

```

Output:

```

Monkey and Banana Problem - Solution Steps:
Step 1: Monkey=door, Box>window, OnBox=False
Step 2: Monkey>window, Box>window, OnBox=False
Step 3: Monkey=middle, Box=middle, OnBox=False
Step 4: Monkey=middle, Box=middle, OnBox=True

Monkey reached the bananas!
== Code Execution Successful ==

```

5. Write the python program to place eight queens on an 8×8 chessboard in such a way that no two queens threaten each other. In other words, no two queens can share the same row, column, or diagonal. Use backtracking to explore different possibilities and ensures that no two queens threaten each other.

The screenshot shows a Python code editor interface with a dark theme. On the left is the code editor pane containing `main.py`, which implements a backtracking algorithm to solve the 8-queens problem. On the right is the output pane showing the solution board configuration.

```

main.py
1 def is_safe(board, row, col):
2     for c in range(col):
3         if board[row][c] == 1:
4             return False
5     r, c = row, col
6     while r >= 0 and c >= 0:
7         if board[r][c] == 1:
8             return False
9         r -= 1
10        c -= 1
11    r, c = row, col
12    while r < 8 and c >= 0:
13        if board[r][c] == 1:
14            return False
15        r += 1
16        c -= 1
17    return True
18 def solve(board, col):
19     if col == 8:
20         return True
21     for row in range(8):
22         if is_safe(board, row, col):
23             board[row][col] = 1 |

```

Output:

```

8 Queens Solution:
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
== Code Execution Successful ==

```