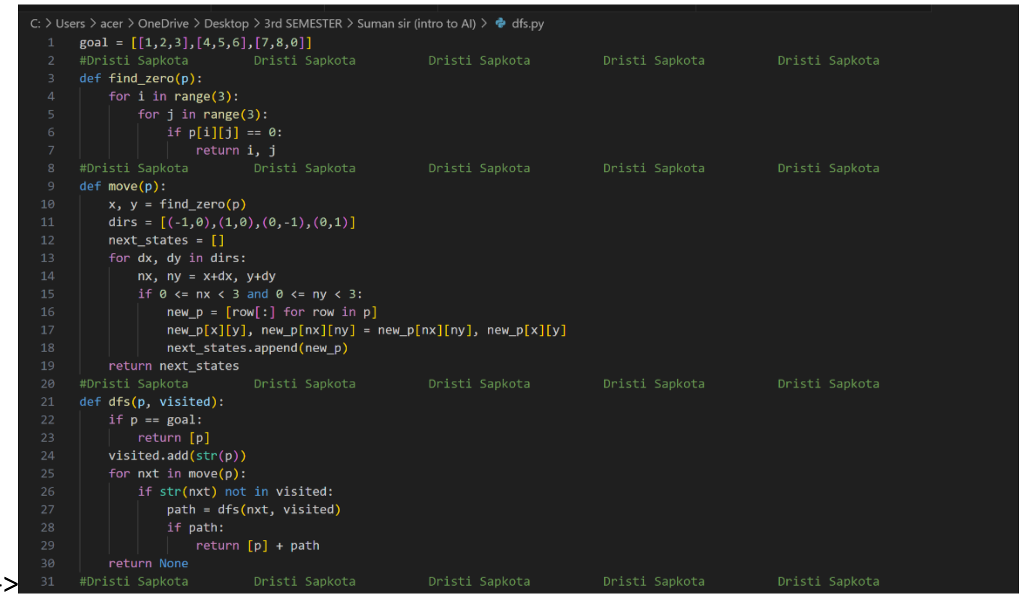
1. Depth-First Search (DFS) Algorithm:

   a. Explain how the DFS algorithm works, and describe it using pseudocode or an algorithm.

   --> Depth-First Search (DFS) is an algorithm for traversing or searching tree or graph data structures. It explores as far as possible along each branch before backtracking. Here's how it works:
   • Start at the initial node (root)
   • Explore one branch as deeply as possible
   • When you reach a node with no unvisited children, backtrack to the nearest ancestor with unexplored children
   • Repeat until the goal is found or all nodes are explored

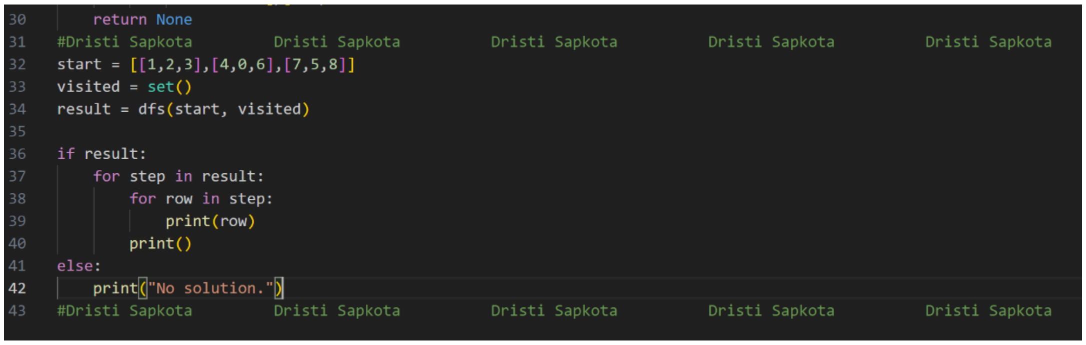
   b,c,d.



C: > Users > acer > OneDrive > Desktop > 3rd SEMESTER > Suman sir (intro to AI) > dfs.py

```python
goal = [[1,2,3],[4,5,6],[7,8,0]]
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota
def find_zero(p):
    for i in range(3):
        for j in range(3):
            if p[i][j] == 0:
                return i, j
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota
def move(p):
    x, y = find_zero(p)
    dirs = [(-1,0),(1,0),(0,-1),(0,1)]
    next_states = []
    for dx, dy in dirs:
        nx, ny = x+dx, y+dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_p = [row[:] for row in p]
            new_p[x][y], new_p[nx][ny] = new_p[nx][ny], new_p[x][y]
            next_states.append(new_p)
    return next_states
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota
def dfs(p, visited):
    if p == goal:
        return [p]
    visited.add(str(p))
    for nxt in move(p):
        if str(nxt) not in visited:
            path = dfs(nxt, visited)
            if path:
                return [p] + path
    return None
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota
```

--->



```python
    return None
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota
start = [[1,2,3],[4,0,6],[7,5,8]]
visited = set()
result = dfs(start, visited)

if result:
    for step in result:
        for row in step:
            print(row)
        print()
else:
    print("No solution.")
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota
```

1. Loops without visited tracking:
DFS can revisit the same states (e.g., moving a tile left, then right) and get stuck in infinite loops if visited states aren't recorded.

2. No guarantee of shortest path:
DFS dives deep into one path, even if it's very long, and may ignore a much shorter solution elsewhere.

3. Might miss the solution:
DFS might explore deep, unproductive paths and never reach the goal, especially in large or complex puzzles.

4. Recursion limit:
Python's recursion depth is limited. Deep searches can hit this limit and crash the program.

2. a.
--> When implementing Breadth-First Search (BFS) in Python, I built the state space dynamically during the search rather than constructing the entire tree or graph beforehand.

b.
--> Instead of generating the entire state space upfront I did the following things:-

• Initialized the BFS queue with the starting state.
• Used a loop to expand nodes level by level:
• Dequeued the current state.
• Generated its valid successors or neighbors on the fly.
• Added unvisited successors to the queue and marked them as visited.
• Continued until I found the goal state or exhausted all possibilities.

For large problems, the entire state space can be massive or even infinite, especially in puzzles or games. Dynamically generating successors reduces memory usage, as only the needed portion of the state space is created. Constructing the full state space before BFS would be computationally expensive and redundant, especially when many nodes may never be visited.

3. a.

Informed Search (also called heuristic search) is a type of search algorithm that uses additional information (a heuristic) to make decisions about which paths to explore first.

b.
Informed search differs from Breadth-First Search (BFS) by using problem-specific knowledge, called a heuristic, to guide the search process more efficiently toward the goal. While BFS is an uninformed search strategy that blindly explores all nodes at the current depth before moving to the next level, informed search algorithms like A* prioritize exploring nodes that appear more promising based on the heuristic. This heuristic estimates the cost or distance from a given node to the goal, allowing the algorithm to make smarter decisions and potentially reduce the number of explored nodes. As a result, informed search is typically faster and more efficient in finding solutions, especially in large or complex problem spaces, whereas BFS may waste time exploring irrelevant paths because it lacks goal-direction awareness.

c.
For example, consider navigating through a grid to reach a target cell in the bottom-right corner from the top-left. In Breadth-First Search (BFS), the algorithm explores all directions equally, layer by layer, without considering which direction leads closer to the goal. This means it might check many unnecessary cells before reaching the destination. In contrast, an informed search algorithm like A* uses a heuristic, such as the Manhattan distance (the number of steps horizontally and vertically to the goal), to prioritize paths that appear closer to the target. As a result, A* will focus more on the path leading toward the goal and avoid less promising directions, making it significantly more efficient in finding the shortest path, especially in larger grids or complex mazes.

4. a.
In the context of informed search algorithms, heuristic values are estimates of the cost or distance from a given state (or node) to the goal state. These values are provided by a heuristic function h(n), which helps the algorithm decide which paths are more promising to explore first. Heuristics guide the search process, making it more efficient by prioritizing nodes that are likely to lead to the goal faster, rather than blindly exploring all options like uninformed search.

b.
- Manhattan Distance
- Definition: Sum of horizontal and vertical steps to reach the goal.
- Applications: Grid-based pathfinding (e.g., maze solving, robot navigation).
- Euclidean Distance

• Definition: Straight-line distance between two points (as-the-crow-flies).
• Applications: GPS navigation, real-world movement tasks.
• Hamming Distance
• Definition: Number of differing bits between two binary strings.
• Applications: Error correction, binary string matching (e.g., 8-puzzle).
• Misplaced Tiles
• Definition: Number of tiles in the wrong position compared to the goal.
• Applications: Puzzle-solving problems (e.g., 8-puzzle, sliding tiles).

5.
To match one binary string to another (e.g., from 10110101 to 01100011), A* can be used by treating each state as a binary string, and each action as flipping one bit.

State: A binary string.

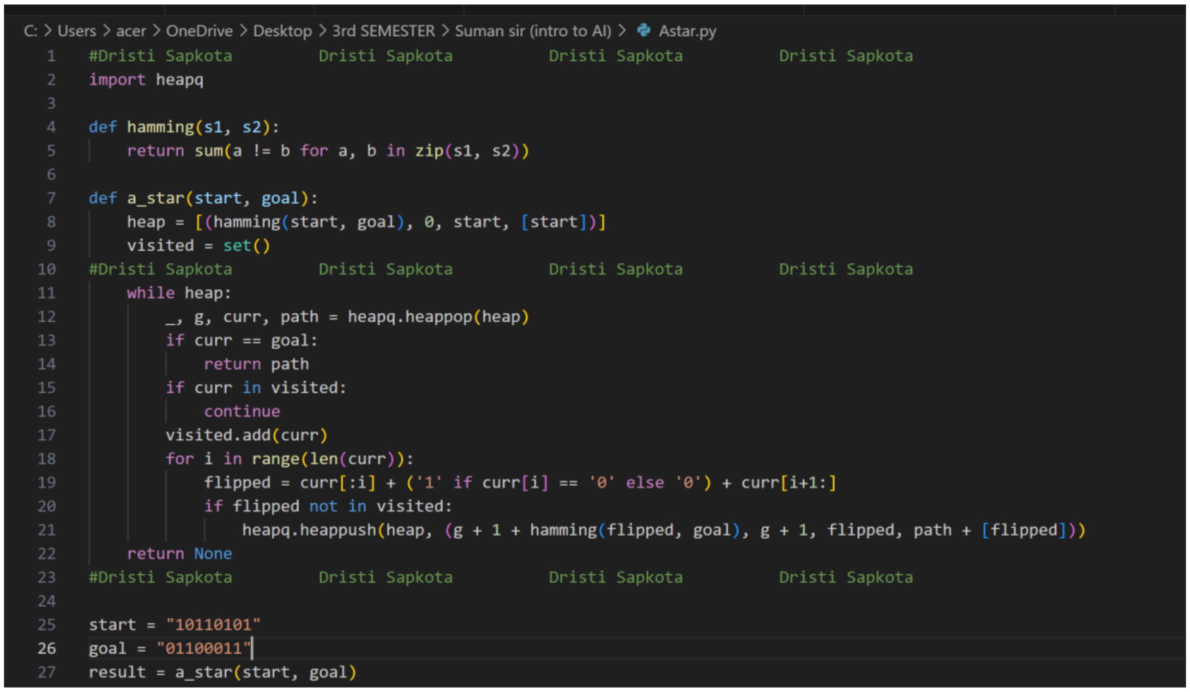
Action: Flip one bit.

Cost: Each flip costs 1.

Goal Test: When the current string equals the goal string.

Heuristic: Hamming distance – counts how many bits differ from the goal.

Since each flip fixes one bit, the Hamming distance gives the minimum number of moves needed, making it a good and admissible heuristic for A*.



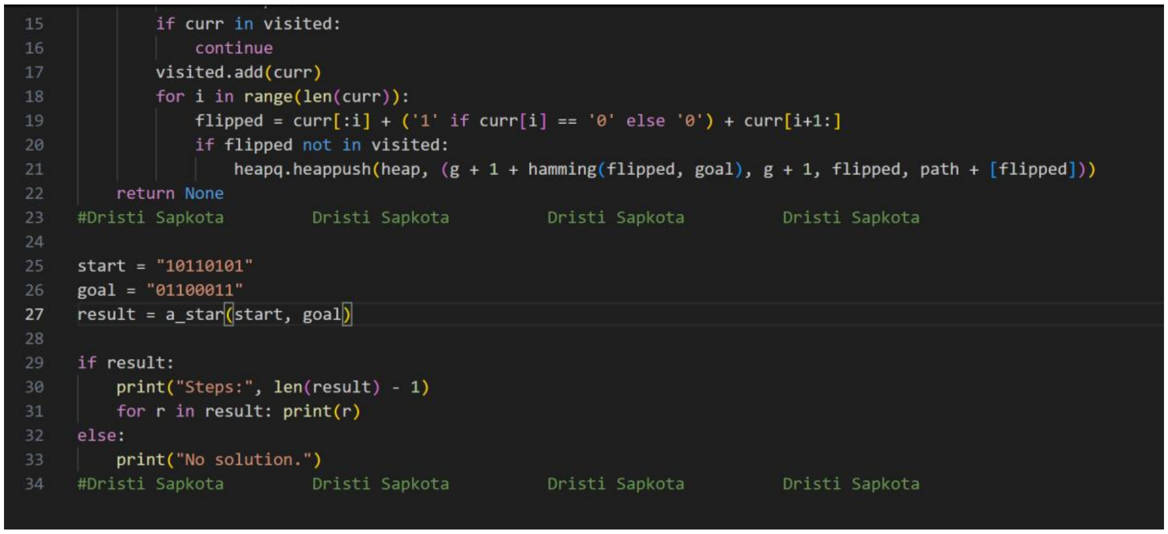
C: > Users > acer > OneDrive > Desktop > 3rd SEMESTER > Suman sir (intro to AI) > Astar.py

```python
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota
import heapq

def hamming(s1, s2):
    return sum(a != b for a, b in zip(s1, s2))

def a_star(start, goal):
    heap = [(hamming(start, goal), 0, start, [start])]
    visited = set()
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota
    while heap:
        _, g, curr, path = heapq.heappop(heap)
        if curr == goal:
            return path
        if curr in visited:
            continue
        visited.add(curr)
        for i in range(len(curr)):
            flipped = curr[:i] + ('1' if curr[i] == '0' else '0') + curr[i+1:]
            if flipped not in visited:
                heapq.heappush(heap, (g + 1 + hamming(flipped, goal), g + 1, flipped, path + [flipped]))
    return None
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota

start = "10110101"
goal = "01100011"
result = a_star(start, goal)
```



```python
        if curr in visited:
            continue
        visited.add(curr)
        for i in range(len(curr)):
            flipped = curr[:i] + ('1' if curr[i] == '0' else '0') + curr[i+1:]
            if flipped not in visited:
                heapq.heappush(heap, (g + 1 + hamming(flipped, goal), g + 1, flipped, path + [flipped]))
    return None
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota

start = "10110101"
goal = "01100011"
result = a_star(start, goal)

if result:
    print("Steps:", len(result) - 1)
    for r in result: print(r)
else:
    print("No solution.")
#Dristi Sapkota          Dristi Sapkota          Dristi Sapkota          Dristi Sapkota
```

Heuristic Used:

The code uses Hamming distance as the heuristic. It counts how many bits differ from the goal string and gives an estimate of how many steps are left. The hamming_distance() function does this comparison.