1)Find all possible paths from source src to destination des

Code:

```
from collections import defaultdict

def find_all_paths(graph, src, dst, visited, path, all_paths):
    # Mark the current node as visited
    visited[src] = True
    path.append(src)

    # If the current node is the destination, save the path
    if src == dst:
        all_paths.append(path[:])
    else:
        # Recur for all neighbors of the current node
        for neighbor in graph[src]:
            if not visited[neighbor]:
                find_all_paths(graph, neighbor, dst, visited, path,
all_paths)

    # Backtrack: unmark the current node and remove it from the
current path
    visited[src] = False
    path.pop()

def main():
    # Input
    n = int(input())  # Number of vertices
    m = int(input())  # Number of edges
    graph = defaultdict(list)

    # Reading edges
    for _ in range(m):
        u, v = map(int, input().split())
        graph[u].append(v)

    # Source and destination nodes
    s = int(input())
```

```
    d = int(input())

    # Variables to store visited nodes, current path, and all paths
    visited = [False] * n
    path = []
    all_paths = []

    # Find all paths from source to destination
    find_all_paths(graph, s, d, visited, path, all_paths)

    # Output all paths
    for p in all_paths:
        print(p)

# Sample execution
if __name__ == "__main__":
    main()
```

2) BFS Traversal of cities

```
from collections import defaultdict, deque

def bfs_traversal(v, edges, source):
    # Create an adjacency list representation of the graph
    graph = defaultdict(list)
    for a, b in edges:
        graph[a].append(b)
        graph[b].append(a)  # For undirected graph, add both directions

    # Initialize visited set and BFS queue
    visited = [False] * v
    queue = deque([source])
    visited[source] = True
    traversal_order = []

    # Perform BFS
    while queue:
```

```
        current = queue.popleft()
        traversal_order.append(current)

        # Visit all neighbors of the current node
        for neighbor in sorted(graph[current]):  # Sort for consistent
order
            if not visited[neighbor]:
                visited[neighbor] = True
                queue.append(neighbor)

    return traversal_order


def main():
    # Input
    v, e = map(int, input().split())  # Number of cities and connections
    edges = [tuple(map(int, input().split())) for _ in range(e)]
    source = int(input())  # Source city

    # Perform BFS traversal
    result = bfs_traversal(v, edges, source)

    # Output the order of BFS traversal
    print(" ".join(map(str, result)))


# Sample execution
if __name__ == "__main__":
    main()
```

3) Lexicographical order traversal:

```
from collections import defaultdict, deque

def bfs_lexicographical_order(n, edges, source):
    # Create an adjacency list representation of the graph
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)  # Since the graph is undirected
```

```
    # Sort the adjacency list to ensure lexicographical order
    for node in graph:
        graph[node].sort()

    # Initialize visited set and BFS queue
    visited = set()
    queue = deque([source])
    visited.add(source)

    # Store the BFS traversal order
    traversal_order = []

    # Perform BFS
    while queue:
        current = queue.popleft()
        traversal_order.append(current)

        # Visit all unvisited neighbors in lexicographical order
        for neighbor in graph[current]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return traversal_order


def main():
    # Input reading
    n = int(input())  # Number of nodes
    m = int(input())  # Number of edges
    edges = [tuple(input().split()) for _ in range(m)]  # Edges of the
graph
    source = input().strip()  # Source node

    # Perform BFS traversal in lexicographical order
    result = bfs_lexicographical_order(n, edges, source)

    # Output the traversal order
    print(" ".join(result))


# Sample execution
```

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

4)ELENA , the Data scientist problem(Hill Climbing)

```python
import math

def hill_climbing():
    # Step size
    step = 0.05

    # Initialize the starting point
    x = 0.0
    y = math.sin(x) - 0.1 * (x ** 2)

    # Variable to track whether we can still climb
    while True:
        # Evaluate the function at points near the current point
        left_x = x - step
        right_x = x + step

        left_y = math.sin(left_x) - 0.1 * (left_x ** 2)
        right_y = math.sin(right_x) - 0.1 * (right_x ** 2)

        # If moving left or right gives a higher value, move there
        if left_y > y:
            x, y = left_x, left_y
        elif right_y > y:
            x, y = right_x, right_y
        else:
            # No further improvement possible, stop the search
            break

    # Output the result
    print(f"Maximum found at X = {x:.2f} with value Y = {y:.2f}")


# Run the hill-climbing algorithm
hill_climbing()
```

Aliter

5) AO * code

```
from collections import defaultdict
import heapq


def parse_graph_input():
    # Step 1: Parse nodes and their costs
    nodes = {}
    while True:
        line = input().strip()
        if line.lower() == "done":
            break
        node, cost = line.split()
        nodes[node] = int(cost)

    # Step 2: Parse relationships between nodes
    graph = defaultdict(list)
    while True:
        line = input().strip()
        if line.lower() == "done":
            break
        parent, children = line.split(" ", 1)
        children = children.split(",")
        for child in children:
            child_node, cost = child.split(":")
            graph[parent].append((child_node, int(cost)))

    return nodes, graph


def ao_star_algorithm(nodes, graph, start, goal):
    # Priority queue for AO* search (min-heap)
    pq = []
    heapq.heappush(pq, (0, start, [start]))  # (cumulative cost, current
node, path)

    visited = set()
```

```
    while pq:
        cumulative_cost, current_node, path = heapq.heappop(pq)

        # If goal is reached, return the shortest path
        if current_node == goal:
            return path

        # Skip if the node has already been visited
        if current_node in visited:
            continue

        visited.add(current_node)

        # Explore children of the current node
        for child, cost in graph[current_node]:
            if child not in visited:
                new_cost = cumulative_cost + cost + nodes[child]
                heapq.heappush(pq, (new_cost, child, path + [child]))

    # If no path is found
    return None


def main():
    # Parse input
    nodes, graph = parse_graph_input()

    # Read the start and goal nodes
    start = input().strip()
    goal = input().strip()

    # Run the AO* algorithm
    result = ao_star_algorithm(nodes, graph, start, goal)

    # Output result
    if result:
        print(f"Shortest Path: {' -> '.join(result)}")
    else:
        print("No path found")


# Run the program
if __name__ == "__main__":
```

```
    main()
```

6) A* Algorithm

```
from collections import defaultdict
import heapq

def a_star_search(graph, heuristics, start, goal):
    # Priority queue for A* search (min-heap)
    pq = []
    heapq.heappush(pq, (0 + heuristics[start], 0, start, [start]))  #
(f_score, g_score, current_node, path)

    visited = set()

    while pq:
        f_score, g_score, current_node, path = heapq.heappop(pq)

        # If goal is reached, return the result
        if current_node == goal:
            print(f"Visiting: {current_node}")
            print(f"Goal reached: {goal}")
            print(f"Path: {path}")
            print(f"Cost: {g_score}")
            return

        # Skip if the node has already been visited
        if current_node in visited:
            continue

        visited.add(current_node)
        print(f"Visiting: {current_node}")

        # Explore neighbors of the current node
        for neighbor, cost in graph[current_node]:
            if neighbor not in visited:
                new_g_score = g_score + cost
                new_f_score = new_g_score + heuristics[neighbor]
                heapq.heappush(pq, (new_f_score, new_g_score, neighbor,
path + [neighbor]))
```

```python
    # If no path is found
    print("No path found")


def main():
    # Input: Number of nodes
    n = int(input().strip())

    # Input: Number of edges
    e = int(input().strip())

    # Input: Graph edges
    graph = defaultdict(list)
    for _ in range(e):
        from_node, to_node, cost = map(int, input().strip().split())
        graph[from_node].append((to_node, cost))

    # Input: Number of heuristics
    h = int(input().strip())

    # Input: Heuristic values
    heuristics = {}
    for _ in range(h):
        node, heuristic = map(int, input().strip().split())
        heuristics[node] = heuristic

    # Input: Start and goal nodes
    start = int(input().strip())
    goal = int(input().strip())

    # Run A* search
    a_star_search(graph, heuristics, start, goal)


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

7)Minimax Algorithm:

```
def power(num):
    count = 0
    num = num // 2
    while num:
        count += 1
        num = num // 2
    return count


def minimax(score, flag):
    # flag 1 for max player
    # flag 0 for min player
    while len(score) != 1:
        if flag == 0:
            li = []
            for i in range(0, len(score), 2):
                li.append(min(score[i], score[i + 1]))
            flag = 1
            score = li
        else:
            li = []
            for i in range(0, len(score), 2):
                li.append(max(score[i], score[i + 1]))
            score = li
            flag = 0
    return score


def main():
    n = int(input())
    score = list(map(int, input().split()))
    p = power(n)

    if p % 2 == 0:
        ans = minimax(score, 0)
    else:
        ans = minimax(score, 1)
    print("The optimal value is:", ans[0])
```

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

8) Alpha Beta Pruning

```
def minimax_with_pruning(node_values, depth, is_maximizing, alpha, beta):
    # Base case: if we're at a leaf node
    if len(node_values) == 1:  # A single value in node_values means it's a
leaf
        return node_values[0]

    if is_maximizing:
        max_eval = float('-inf')
        for i in range(2):  # Two children
            child_values = node_values[i * len(node_values) // 2:(i + 1) *
len(node_values) // 2]
            eval = minimax_with_pruning(child_values, depth + 1, False,
alpha, beta)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(2):  # Two children
            child_values = node_values[i * len(node_values) // 2:(i + 1) *
len(node_values) // 2]
            eval = minimax_with_pruning(child_values, depth + 1, True,
alpha, beta)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval


def calculate_optimal_yield(base_yields):
    # Call the minimax function starting with the full list of base_yields
    base_optimal_value = minimax_with_pruning(base_yields, 0, True,
```

```
float('-inf'), float('inf'))

    # Calculate the value after the 10% increase
    increased_optimal_value = int(base_optimal_value * 1.1)

    return base_optimal_value, increased_optimal_value


if __name__ == "__main__":
    # Input the base yields from the user
    base_yields = list(map(int, input().strip().split()))

    # Validate the input size
    if len(base_yields) != 8:
        print("Error: Please enter exactly 8 integers for base yields.")
    else:
        # Calculate the optimal values
        base_optimal_value, increased_optimal_value =
calculate_optimal_yield(base_yields)

        # Print the output in the required format
        print(f"Base optimal value: {base_optimal_value}")
        print(f"After 10% increase: {increased_optimal_value}")
```

9.Alpha beta pruning George

```
def minimax_with_pruning(node_values, depth, is_maximizing, alpha, beta,
current_depth):
    if len(node_values) == 1:  # Base case: leaf node
        return node_values[0]

    if is_maximizing:
        best_value = float('-inf')
        for i in range(2):  # Two children
            child_values = node_values[i * len(node_values) // 2:(i + 1) *
len(node_values) // 2]
            eval_value = minimax_with_pruning(child_values, depth, False,
alpha, beta, current_depth + 1)
            if eval_value > best_value:
```

```
                best_value = eval_value
                print(f"Maximizer updated best value to {best_value} at
depth {current_depth}")
            alpha = max(alpha, best_value)
            if beta <= alpha:
                print(f"Pruning at depth {current_depth} by Maximizer")
                break
        return best_value
    else:
        best_value = float('inf')
        for i in range(2):  # Two children
            child_values = node_values[i * len(node_values) // 2:(i + 1) *
len(node_values) // 2]
            eval_value = minimax_with_pruning(child_values, depth, True,
alpha, beta, current_depth + 1)
            if eval_value < best_value:
                best_value = eval_value
                print(f"Minimizer updated best value to {best_value} at
depth {current_depth}")
            beta = min(beta, best_value)
            if beta <= alpha:
                print(f"Pruning at depth {current_depth} by Minimizer")
                break
        return best_value


if __name__ == "__main__":
    # Input the depth of the tree and the leaf node values
    d = int(input().strip())
    node_values = list(map(int, input().strip().split()))

    # Validate the input
    if len(node_values) != 2**d:
        print("Error: The number of leaf nodes must be 2^d for a complete
binary tree.")
    else:
        # Execute minimax with alpha-beta pruning and print the optimal
value
        optimal_value = minimax_with_pruning(node_values, d, True,
float('-inf'), float('inf'), 0)
        print(f"Optimal Value: {optimal_value}")
```

10) Alex KB

```python
def main():
    knowledge_base = {
        "USA": "Washington, D.C.",
        "India": "New Delhi",
        "Japan": "Tokyo",
        "France": "Paris"
    }

    while True:
        option = input().strip()
        if option == "1":
            country = input().strip()
            capital = knowledge_base.get(country, "Unknown")
            print(capital)

        elif option == "2":
            country = input().strip()
            capital = input().strip()
            if country and capital:
                knowledge_base[country] = capital
                print(f"Added successfully: {country} - {capital}")

        elif option == "3":
            return

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

11) James KB

```python
def main():
    raining = input().strip().lower()
    temperature = int(input().strip())

    if raining == "yes" and temperature < 30:
        print("Take an umbrella.")
    elif raining == "no" and temperature >= 20:
        print("No need for an umbrella.")
    else:
        print("Check the weather again.")

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

12) Jamie KB

```python
def main():
    location = input().strip().lower()
    lunch_prepared = input().strip().lower()

    if location == "home":
        print("You can have lunch at home.")
    elif location == "school" and lunch_prepared == "yes":
        print("Bring your lunch.")
    elif location == "school" and lunch_prepared == "no":
        print("Buy lunch at school.")
    else:
        print("Check your location and lunch status.")

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

13,14,15) DFS, BFS , A*

```
from collections import deque
import heapq

def is_valid_move(maze, visited, x, y):
    rows, cols = len(maze), len(maze[0])
    return 0 <= x < rows and 0 <= y < cols and maze[x][y] == 0 and not
visited[x][y]

def dfs(maze, start, goal):
    stack = [start]
    visited = [[False for _ in range(len(maze[0]))] for _ in
range(len(maze))]
    visited[start[0]][start[1]] = True

    while stack:
        x, y = stack.pop()

        if (x, y) == goal:
            return True

        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            nx, ny = x + dx, y + dy
            if is_valid_move(maze, visited, nx, ny):
                visited[nx][ny] = True
                stack.append((nx, ny))

    return False

def bfs(maze, start, goal):
    queue = deque([start])
    visited = [[False for _ in range(len(maze[0]))] for _ in
range(len(maze))]
    visited[start[0]][start[1]] = True

    while queue:
        x, y = queue.popleft()

        if (x, y) == goal:
            return True
```

```
        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            nx, ny = x + dx, y + dy
            if is_valid_move(maze, visited, nx, ny):
                visited[nx][ny] = True
                queue.append((nx, ny))

    return False

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(maze, start, goal):
    open_set = []
    heapq.heappush(open_set, (0, start))
    visited = [[False for _ in range(len(maze[0]))] for _ in
range(len(maze))]
    g_score = {start: 0}

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            return True

        x, y = current
        if visited[x][y]:
            continue
        visited[x][y] = True

        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            nx, ny = x + dx, y + dy
            neighbor = (nx, ny)
            if is_valid_move(maze, visited, nx, ny):
                temp_g_score = g_score[current] + 1
                if neighbor not in g_score or temp_g_score <
g_score[neighbor]:
                    g_score[neighbor] = temp_g_score
                    f_score = temp_g_score + heuristic(neighbor, goal)
                    heapq.heappush(open_set, (f_score, neighbor))

    return False
```

```python
# Example Usage
maze = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]

start = (0, 0)
goal = (4, 4)

print("DFS:", dfs(maze, start, goal))
print("BFS:", bfs(maze, start, goal))
print("A*:", astar(maze, start, goal))
```