AI PRACTICAL SLIPS

1] BFS:

```
#Write a program to implement BFS
graph = {
    'A' : ['B', 'C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
}

visited = []  # List to keep track of visited nodes.
queue = []    # Initialize a queue

def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)

    while queue:
        s = queue.pop(0)
        print(s, end=" ")

        for neighbour in graph[s]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

# Driver Code
bfs(visited, graph, 'A')
```

2] DFS :

```
# Using a Python dictionary to act as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

visited = set()  # Set to keep track of visited nodes.

def dfs(visited, graph, node):
    if node not in visited:
```

```
        print(node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
dfs(visited, graph, 'A')
```

3] ASTAR ALGO:

```
def astaralgo(start_node,stop_node):

    open_set = set(start_node)
    closed_set= set()

    g={}
    parents = {}

    g[start_node] = 0

    parents[start_node]  = start_node

    while len(open_set) > 0:
        n=None

        for v in open_set:
            if n==None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n=v
        if n==stop_node or Graph_nodes[n]==None:
                pass

        else:

            for(m,weight) in get_neighbors(n):
                if m not in open_set  and m not in closed_set:
                        open_set.add(m)
                        parents[m]=n
                        g[m] = g[n] + weight
                else:
                        if g[m] > g[n] +  weight:
                            g[m] = g[n] + weight
                            parents[m]=n

                            if m in closed_set:
                                closed_set.remove(m)
                                open_set.add(m)
        if n==None:
            print("path does not exist!")
            return None
```

```
        if n==stop_node:
            path=[]

            while parents[n]!=n:
                path.append(n)
                n=parents[n]

            path.append(start_node)

            path.reverse()

            print("Path found: {}".format(path))
            return path


        open_set.remove(n)
        closed_set.add(n)
    print("Path does not exist!")
    return None


def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]

    else:
        return None

def heuristic(n):
    H_dist = {
        'A' : 11,
        'B' : 6,
        'C' : 99,
        'D' : 1,
        'E' : 7,
        'G' : 0,
    }
    return H_dist[n]


Graph_nodes = {
    'A' : [('B',2),('E',3)],
    'B' : [('C',1),('G',9)],
    'C' : None,
    'E' : [('D',6)],
    'D' : [('G',1)],
}
astaralgo('A','G')
```

4] AOSTAR:

```
import heapq

class Node:
    def __init__(self, name, cost=0, is_or_node=True):
        self.name = name
        self.cost = cost
        self.is_or_node = is_or_node
        self.children = []

    def add_child(self, child, cost):
        self.children.append((child, cost))

    def __lt__(self, other):
        return self.cost < other.cost

def a_star_search(start, goal):
    open_set = []
    heapq.heappush(open_set, (start.cost, start))
    came_from = {}
    cost_so_far = {start: 0}

    while open_set:
        current_cost, current_node = heapq.heappop(open_set)

        if current_node.name == goal.name:
            return reconstruct_path(came_from, current_node)

        for child, edge_cost in current_node.children:
            new_cost = cost_so_far[current_node] + edge_cost
            if child not in cost_so_far or new_cost < cost_so_far[child]:
                cost_so_far[child] = new_cost
                priority = new_cost + heuristic(child, goal)
                heapq.heappush(open_set, (priority, child))
                came_from[child] = current_node

    return None

def heuristic(node, goal):
    # Simple heuristic: distance to the goal
    return abs(hash(node.name) - hash(goal.name)) % 10

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    return path[::-1]
```

```
# Example usage
if __name__ == "__main__":
    # Create nodes
    start = Node("A", cost=0, is_or_node=True)
    b = Node("B", cost=1, is_or_node=True)
    c = Node("C", cost=2, is_or_node=True)
    goal = Node("D", cost=0, is_or_node=False)

    # Create the graph
    start.add_child(b, cost=1)
    start.add_child(c, cost=2)
    b.add_child(goal, cost=3)
    c.add_child(goal, cost=1)

    # Perform AO* search
    path = a_star_search(start, goal)
    if path:
        print("Path found:")
        for node in path:
            print(node.name)
    else:
        print("No path found")
```

5] Selection

```
def selection_sort(arr):
    # Traverse through all array elements
    for i in range(len(arr)):
        # Find the minimum element in the remaining unsorted array
        min_index = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j

        # Swap the found minimum element with the first unsorted element
        arr[i], arr[min_index] = arr[min_index], arr[i]

    return arr

# Example usage
if __name__ == "__main__":
    arr = [64, 25, 12, 22, 11]
    print("Original array:", arr)
    sorted_arr = selection_sort(arr)
    print("Sorted array:", sorted_arr)
```

6] Adjacency List

```
# Define the undirected graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

visited = set()  # Set to keep track of visited nodes.

def dfs(visited, graph, node):
    if node not in visited:
        print(node)          # Print the node when it is first visited
        visited.add(node)   # Mark the node as visited
        for neighbour in graph[node]:
            if neighbour not in visited:  # Check if the neighbour has been visited
                dfs(visited, graph, neighbour)  # Recursively visit the neighbour

# Driver Code
print("DFS traversal of the graph:")
dfs(visited, graph, 'A')
```

7] MSTKruskals

```
class DisjointSet:
    def __init__(self, vertices):
        self.parent = {v: v for v in vertices}
        self.rank = {v: 0 for v in vertices}

    def find(self, item):
        if self.parent[item] == item:
            return item
        else:
            self.parent[item] = self.find(self.parent[item])
            return self.parent[item]

    def union(self, set1, set2):
        root1 = self.find(set1)
        root2 = self.find(set2)

        if root1 != root2:
            if self.rank[root1] > self.rank[root2]:
                self.parent[root2] = root1
            elif self.rank[root1] < self.rank[root2]:
                self.parent[root1] = root2
            else:
```

```
                self.parent[root2] = root1
                self.rank[root1] += 1


def kruskal(vertices, edges):
    # Sort edges by weight
    edges = sorted(edges, key=lambda edge: edge[2])

    # Initialize Disjoint Set
    disjoint_set = DisjointSet(vertices)

    mst = []

    for edge in edges:
        u, v, weight = edge
        # Check if including this edge would form a cycle
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            mst.append(edge)

    return mst

    # List of vertices in the graph
vertices = ['A', 'B', 'C', 'D', 'E']

# List of edges in the graph (u, v, weight)
edges = [
    ('A', 'B', 1),
    ('A', 'C', 3),
    ('B', 'C', 3),
    ('B', 'D', 6),
    ('C', 'D', 4),
    ('C', 'E', 2),
    ('D', 'E', 5)
]

# Compute the Minimum Spanning Tree using Kruskal's Algorithm
mst = kruskal(vertices, edges)

# Print the result
print("Edges in the Minimum Spanning Tree:")
for edge in mst:
    print(edge)


8] Shortest Path.py

def dijkstra(graph, start):
    # Priority queue to hold nodes to explore and their current shortest distance
    priority_queue = [(0, start)]
```

```
    # Dictionary to store the shortest distance from start to each node
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    while priority_queue:
        # Get the node with the smallest distance
        current_distance, current_node = heapq.heappop(priority_queue)

        # Nodes can only be added to the priority queue once, so if this distance is
not optimal, skip it
        if current_distance > distances[current_node]:
            continue

        # Explore neighbors
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            # Only consider this new path if it's better
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example usage
if __name__ == "__main__":
    # Define a graph as an adjacency list
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'A': 1, 'C': 2, 'D': 5},
        'C': {'A': 4, 'B': 2, 'D': 1},
        'D': {'B': 5, 'C': 1}
    }

    start_node = 'A'
    shortest_paths = dijkstra(graph, start_node)
    print("Shortest paths from node", start_node, ":", shortest_paths)


9] Simple Chatbot.py

def chatbot():
    print("Chatbot: Hi! I’m your friendly bot. Type 'bye' to exit.")

    responses =
   {
        "hello": "Hello! How can I help you today?",
        "hi": "Hi there! What’s up?",
        "how are you": "I’m just code, but I’m doing great. How about you?",
        "name": "I’m ChatPy, your Python chatbot.",
```

```
    }

    while True:
        user_input = input("You: ").lower()

        if user_input == "bye":
            print("Chatbot: Goodbye!")
            break

        # look for a matching keyword
        reply = None
        for key in responses:
            if key in user_input:
                reply = responses[key]
                break

        if reply:
            print("Chatbot:", reply)
        else:
            print("Chatbot: I’m not sure I understand. Can you rephrase?")


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