SELECTION SORT

def selection\_sort(arr):

n = len(arr)

for i in range(n):

# Find the minimum element in the unsorted portion of the array

min\_index = i

for j in range(i + 1, n):

if arr[j] < arr[min\_index]:

min\_index = j

# Swap the minimum element with the first element of the unsorted portion

arr[i], arr[min\_index] = arr[min\_index], arr[i]

return arr

# Example usage:

arr = [64, 25, 12, 22, 11]

sorted\_arr = selection\_sort(arr)

print("Sorted array is:", sorted\_arr)

PRIMS

import heapq

def prims\_algorithm(graph):

# Initialize variables

mst = []

visited = set()

start\_node = list(graph.keys())[0] # Choose any starting node

visited.add(start\_node)

edges = [(weight, start\_node, neighbor) for neighbor, weight in graph[start\_node]]

heapq.heapify(edges)

# Iterate until all nodes are visited

while len(visited) < len(graph):

# Pop the edge with the smallest weight

weight, source, destination = heapq.heappop(edges)

# If the destination is already visited, skip

if destination in visited:

continue

# Add the edge to the MST

mst.append((source, destination, weight))

visited.add(destination)

# Add new edges to the priority queue

for neighbor, weight in graph[destination]:

if neighbor not in visited:

heapq.heappush(edges, (weight, destination, neighbor))

return mst

# Example usage:

graph = {

'A': [('B', 2), ('C', 3)],

'B': [('A', 2), ('C', 5), ('D', 7)],

'C': [('A', 3), ('B', 5), ('D', 6)],

'D': [('B', 7), ('C', 6)]

}

mst = prims\_algorithm(graph)

print("Minimum Spanning Tree:")

for edge in mst:

print(edge)

JOB SHEDULING

def shortest\_job\_first(jobs):

# Sort jobs based on their durations

jobs.sort(key=lambda x: x[1])

# Initialize variables

schedule = []

current\_time = 0

# Schedule jobs

for job in jobs:

job\_name, duration = job

schedule.append((job\_name, current\_time, current\_time + duration))

current\_time += duration

return schedule

# Example usage:

jobs = [('Job1', 8), ('Job2', 4), ('Job3', 5), ('Job4', 3)]

schedule = shortest\_job\_first(jobs)

print("Job Schedule:")

for job in schedule:

print("Job:", job[0], "| Start Time:", job[1], "| End Time:", job[2])

A\*

import heapq

def heuristic(node, goal):

# Manhattan distance heuristic

return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

def astar(graph, start, goal):

# Initialize variables

open\_set = [(0, start)] # Priority queue with initial node

came\_from = {}

g\_score = {start: 0}

# Initialize g\_scores to infinity

for node in graph:

if node != start:

g\_score[node] = float('inf')

# Main loop

while open\_set:

# Get the node with the lowest f\_score

current\_cost, current\_node = heapq.heappop(open\_set)

# If current node is the goal, reconstruct path and return

if current\_node == goal:

path = []

while current\_node in came\_from:

path.append(current\_node)

current\_node = came\_from[current\_node]

path.append(start)

path.reverse()

return path

# Explore neighbors of current node

for neighbor, cost in graph[current\_node]:

tentative\_g\_score = g\_score[current\_node] + cost

if tentative\_g\_score < g\_score[neighbor]:

# Update scores

came\_from[neighbor] = current\_node

g\_score[neighbor] = tentative\_g\_score

f\_score = tentative\_g\_score + heuristic(neighbor, goal)

# Add neighbor to open set

heapq.heappush(open\_set, (f\_score, neighbor))

# If no path found

return None

# Example usage:

graph = {

(0, 0): [((0, 1), 1), ((1, 0), 1)], # Example grid graph

(0, 1): [((0, 0), 1), ((1, 1), 1)],

(1, 0): [((0, 0), 1), ((1, 1), 1)],

(1, 1): [((0, 1), 1), ((1, 0), 1), ((2, 1), 1)],

(2, 1): [((1, 1), 1), ((2, 0), 1)]

}

start = (0, 0)

goal = (2, 1)

path = astar(graph, start, goal)

if path:

print("Path found:", path)

else:

print("No path found.")

N-QUEEN(BACKTRACKING)

def is\_safe(board, row, col, N):

# Check if there's a queen in the same column

for i in range(row):

if board[i][col] == 1:

return False

# Check upper left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check upper right diagonal

for i, j in zip(range(row, -1, -1), range(col, N)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, row, N, result):

if row == N:

# Convert board to a printable format and add to result

result.append(["".join(["Q" if cell == 1 else "." for cell in row]) for row in board])

return

for col in range(N):

if is\_safe(board, row, col, N):

board[row][col] = 1

solve\_n\_queens\_util(board, row + 1, N, result)

board[row][col] = 0 # Backtrack

def solve\_n\_queens(N):

board = [[0] \* N for \_ in range(N)]

result = []

solve\_n\_queens\_util(board, 0, N, result)

return result

# Example usage:

N = 4

solutions = solve\_n\_queens(N)

print("Number of solutions for", N, "queens:", len(solutions))

for idx, solution in enumerate(solutions, 1):

print("Solution", idx, ":")

for row in solution:

print(row)

N-QUEEN (BRANCH AND BOUND)

def is\_safe(board, row, col, n):

# Check if there's a queen in the same column

for i in range(row):

if board[i][col]:

return False

# Check upper left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j]:

return False

# Check upper right diagonal

for i, j in zip(range(row, -1, -1), range(col, n)):

if board[i][j]:

return False

return True

def solve\_n\_queens(n):

def backtrack(row):

nonlocal min\_cost

if row == n:

# Found a solution

min\_cost += 1

return

for col in range(n):

if is\_safe(board, row, col, n):

board[row][col] = True

backtrack(row + 1)

board[row][col] = False

# Initialize board

board = [[False] \* n for \_ in range(n)]

min\_cost = 0

backtrack(0)

return min\_cost

# Example usage:

n = 8 # Board size

solutions = solve\_n\_queens(n)

print(f"Number of solutions for {n}-Queens problem:", solutions)

CHATBOT

def greet(bot\_name, birth\_year):

print("Hello! My name is {0}.".format(bot\_name))

print("I was created in {0}.".format(birth\_year))

def remind\_name():

print('Please, remind me your name.')

name = input()

print("What a great name you have, {0}!".format(name))

def guess\_age():

print('Let me guess your age.')

print('Enter remainders of dividing your age by 3, 5 and 7.')

rem3 = int(input())

rem5 = int(input())

rem7 = int(input())

age = (rem3 \* 70 + rem5 \* 21 + rem7 \* 15) % 105

print("Your age is {0}; that's a good time to start programming!".format(age))

def count():

print('Now I will prove to you that I can count to any number you want.')

counter = 0

n= int(input("enter a number"))

while counter < n:

print(counter)

counter += 1

def test():

print("Let's test your programming knowledge.")

print("Why do we use methods?")

print("1. To repeat a statement multiple times.")

print("2. To decompose a program into several small subroutines.")

print("3. To determine the execution time of a program.")

print("4. To interrupt the execution of a program.")

answer = 2

guess = int(input())

while guess != answer:

print("Please, try again.")

guess = int(input())

print('Completed, have a nice day!')

def end():

print('Congratulations, have a nice day!')

input()

greet('TE-Chatbot', '2022') # change it as you need

remind\_name()

guess\_age()

count()

test()

end()

EXPERT SYSTEM

EMPLOYEE MANAGEMENT

def evaluate\_performance():

print("Welcome to the Employee Performance Evaluation System\n")

productivity = int(input("Rate the employee's productivity on a scale of 1 to 10: "))

communication = int(input("Rate the employee's communication skills on a scale of 1 to 10: "))

teamwork = int(input("Rate the employee's teamwork abilities on a scale of 1 to 10: "))

punctuality = int(input("Rate the employee's punctuality on a scale of 1 to 10: "))

total\_score = productivity + communication + teamwork + punctuality

if total\_score >= 35:

performance = "Excellent"

elif total\_score >= 30:

performance = "Good"

elif total\_score >= 25:

performance = "Satisfactory"

else:

performance = "Needs Improvement"

print("\nEvaluation Result:")

print("Total Score:", total\_score)

print("Performance Level:", performance)

evaluate\_performance()

BFS

g={

0:[1,2],

1:[0,3,4],

2:[3,0],

3:[2,1,4],

4:[3,1]

}

def bfs(g,s):

q=[s]

vis=[s]

while q:

cur = q.pop(0)

print(cur)

for c in g[cur]:

if c not in vis:

q.append(c)

vis.append(c)

bfs(g,0)

DFS

g={

0:[1,2],

1:[0,3,4],

2:[3,0],

3:[2,1,4],

4:[3,1]

}

def dfs(g,s):

vis[s]=1

print(s)

for c in g[s]:

if not vis[c]:

dfs(g,c)

vis=[0]\*5

dfs(g,0)