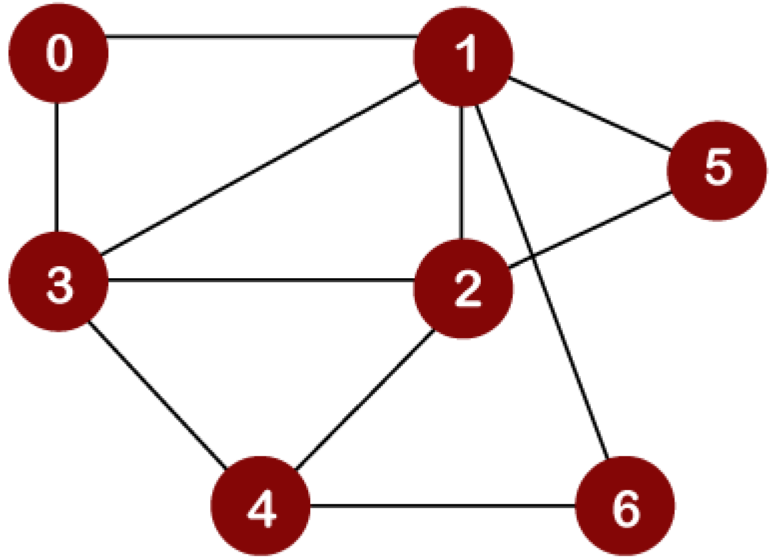
BFS




```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        print(vertex, end=" ")

        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Example graph represented as an adjacency list
graph ={
    '0': ['1','3'],
    '1': ['0','2','3','5','6'],
    '2': ['1','3','4','5'],
    '3': ['0','1','2','4'],
    '4':['2','3','6'],
    '5': ['1','2'],
    '6':['1','4']
}
# Perform BFS starting from vertex 'A'
bfs(graph, '2')
```

Output:

2 1 3 4 5 0 6 [Finished in 410ms]

DFS

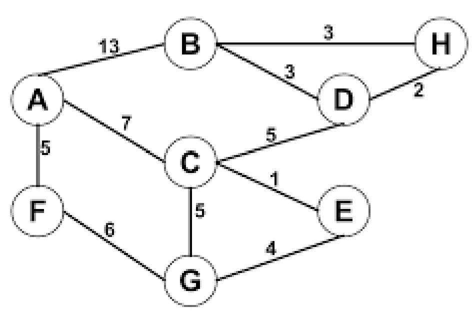
```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Example graph represented as an adjacency list
# graph = {
#     'A': ['B', 'C'],
#     'B': ['D', 'E'],
#     'C': ['F'],
#     'D': [],
#     'E': ['F'],
#     'F': []
# }


graph ={
    '0': ['1','3'],
    '1': ['0','2','3','5','6'],
    '2': ['1','3','4','5'],
    '3': ['0','1','2','4'],
    '4':['2','3','6'],
    '5': ['1','2'],
    '6':['1','4']
}


# Perform DFS starting from vertex 'A'
dfs(graph, '6')
```

Output

6 1 0 3 2 4 5 [Finished in 285ms]

Uniform Search cost

```
import heapq

def uniform_cost_search(graph, start, goal):
    # Priority queue to store (cost, node, path)
    priority_queue = [(0, start, [])]
    visited = set()

    while priority_queue:
        cost, node, path = heapq.heappop(priority_queue)

        if node in visited:
            continue

        path = path + [node]
        visited.add(node)

        if node == goal:
            return cost, path

        for neighbor, weight in graph[node]:
            if neighbor not in visited:
                heapq.heappush(priority_queue, (cost + weight, neighbor, path))

    return float("inf"), []

# Example graph represented as an adjacency list
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('A', 1), ('D', 2), ('E', 5)],
    'C': [('A', 4), ('F', 3)],
    'D': [('B', 2)],
    'E': [('B', 5), ('F', 1)],
    'F': [('C', 3), ('E', 1)]
}
```




```
graph = {
    'A': [('B', 5), ('D', 10)],
```

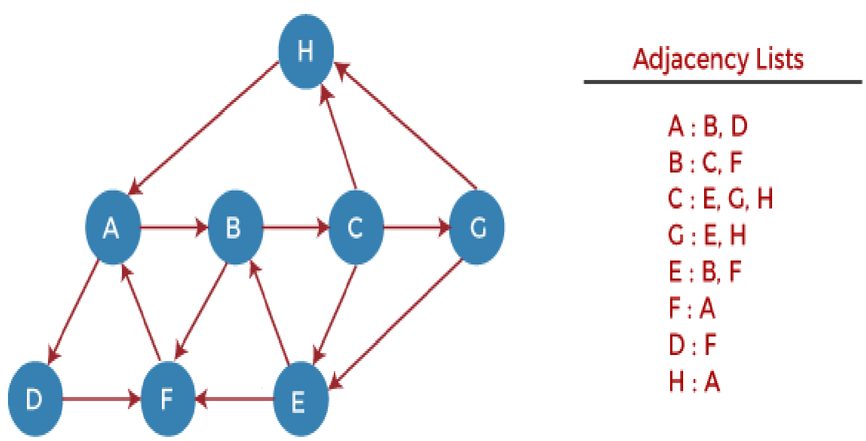
```
    'B': [('A', 5), ('C', 4), ('F', 15)],
    'C': [('B', 4), ('E', 8)],
    'D': [('A', 10), ('F', 11)],
    'E': [('C', 8), ('F', 4)],
    'F': [('B', 15), ('D', 11), ('E', 4)]
}

graph = {
    'A': [('B', 13), ('C', 7), ('F', 5)],
    'B': [('A', 13), ('D', 5), ('H', 3)],
    'C': [('A', 7), ('D', 5), ('E', 1), ('G', 5)],
    'D': [('B', 3), ('H', 2), ('C', 5)],
    'E': [('C', 1), ('G', 4)],
    'F': [('A', 5), ('C', 6)],
    'G': [('C', 5), ('E', 4),('F',6)],
    'H': [('B',3),('D', 2)]
}
# Perform UCS from 'A' to 'F'
cost, path = uniform_cost_search(graph, 'A', 'H')
print(f"Cost: {cost}, Path: {path}")
```

Output:

```
Cost: 14, Path: ['A', 'C', 'D', 'H']
[Finished in 235ms]
```

Depth Limited Search

```
def depth_limited_search(graph, start, goal, limit):
    def dls(node, goal, limit, path, visited):
        if limit < 0:
            return False
        path.append(node)
        visited.add(node)
        if node == goal:
            return True
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dls(neighbor, goal, limit - 1, path, visited):
                    return True
        path.pop()
        return False

    path = []
    visited = set()
    if dls(start, goal, limit, path, visited):
        return path
    else:
        return "No path found within depth limit"
# Example graph represented as an adjacency list
graph ={
    'A': ['B','D'],
    'B': ['C','F'],
    'C': ['E','G','H'],
    'G': ['E','H'],
    'E': ['B','F'],
    'F': ['A'],
    'D': ['F'],
    'H': ['A']
}
```




```
# Perform Depth-Limited Search from 'A' to 'F' with a depth limit of 2
result = depth_limited_search(graph, 'A', 'F',5)
print(result)
```

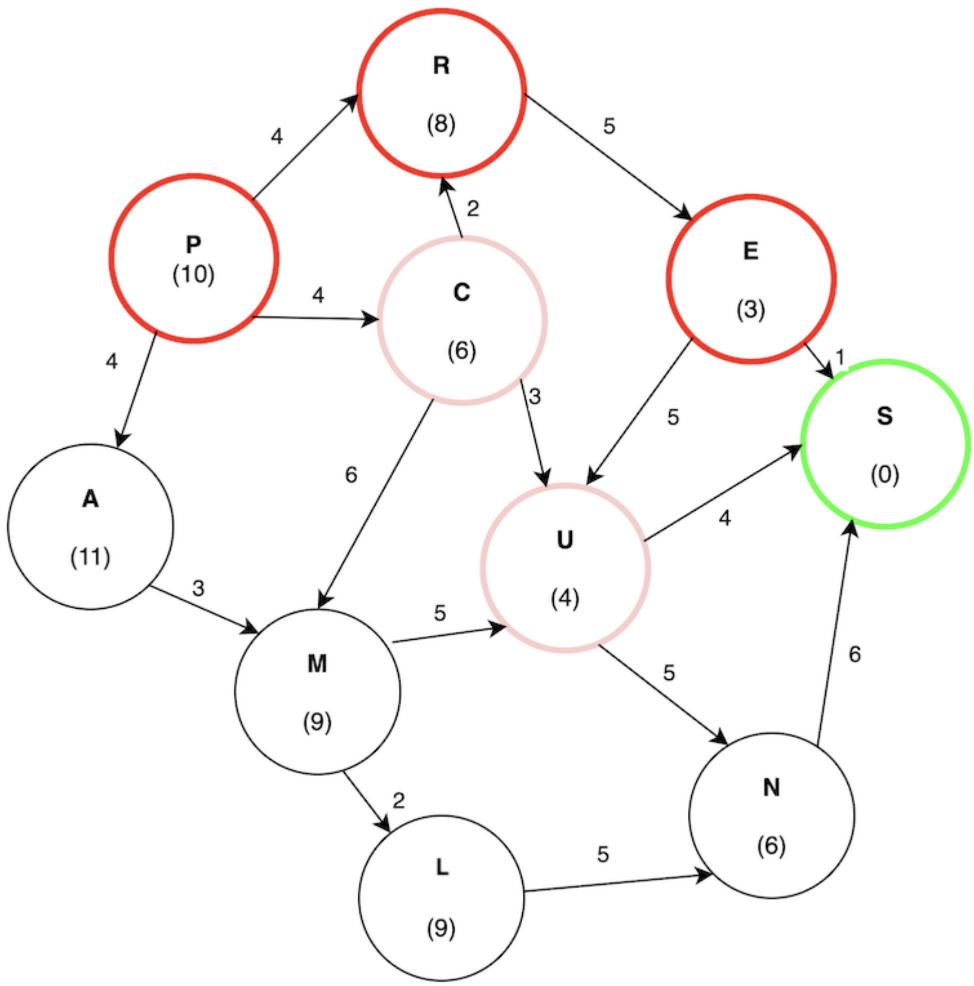
Output:

['A', 'B', 'C', 'E', 'F']   [Finished in 224ms]

Greedy Best First Search

```
import heapq

def greedy_best_first_search(graph, start, goal, heuristic):
    # Priority queue to store (heuristic value, node, path)
    priority_queue = [(heuristic[start], start, [])]
    visited = set()

    while priority_queue:
        _, node, path = heapq.heappop(priority_queue)

        if node in visited:
            continue

        path = path + [node]
        visited.add(node)

        if node == goal:
            return path

        for neighbor in graph[node]:
            if neighbor not in visited:
                heapq.heappush(priority_queue, (heuristic[neighbor], neighbor, path))

    return "No path found"

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

```
# Example heuristic values for each node
# heuristic = {
#     'A': 3,
#     'B': 2,
#     'C': 1,
#     'D': 6,
#     'E': 4,
#     'F': 0
# }

graph = {
    'A': ['M'],
    'C': ['M','R','U'],
    'E': ['S','U'],
    'L': ['N'],
    'M': ['L','U'],
    'N': ['S'],
    'P': ['C','R'],
    'R': ['E'],
    'U': ['N','S'],
    'S': []
}
heuristic = {
    'A': 11,
    'C': 6,
    'E': 3,
    'L': 6,
    'M': 9,
    'N': 6,
    'P': 10,
    'R': 8,
    'U': 4,
    'S': 0
}
```




```
# Perform Greedy Best-First Search from 'A' to 'F'
path = greedy_best_first_search(graph, 'P', 'S', heuristic)
print(f"Path: {path}")
```

Output: Path: ['P', 'C', 'U', 'S']

}

A* search

```
import heapq

def a_star_search(graph, start, goal, heuristic):
    # Priority queue to store (f, g, node, path)
    priority_queue = [(0 + heuristic[start], 0, start, [])]
    visited = set()

    while priority_queue:
        f, g, node, path = heapq.heappop(priority_queue)

        if node in visited:
            continue

        path = path + [node]
        visited.add(node)

        if node == goal:
            return path

        for neighbor, cost in graph[node]:
            if neighbor not in visited:
                g_new = g + cost
                f_new = g_new + heuristic[neighbor]
                heapq.heappush(priority_queue, (f_new, g_new, neighbor, path))

    return "No path found"
/*
# Example graph represented as an adjacency list
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('D', 3), ('E', 5)],
    'C': [('F', 2)],
    'D': [('F', 1)],
    'E': [('F', 2)],
    'F': []
}
```

```
# Example heuristic values for each node
heuristic = {
    'A': 7,
    'B': 6,
    'C': 2,
    'D': 1,
    'E': 3,
    'F': 0
}*/

graph = {
    'A': [('M',3)],
    'C': [('M',6),('R',2),('U',3)],
    'E': [('S',1),('U',5)],
    'L': [('N',5)],
    'M': [('L',2),('U',5)],
    'N': [('S',6)],
    'P': [('C',4),('R',4)],
    'R': [('E',5)],
    'U': [('N',5),('S',4)],
    'S': []
}


heuristic = {
    'A': 11,
    'C': 6,
    'E': 3,
    'L': 6,
    'M': 9,
    'N': 6,
    'P': 10,
    'R': 8,
    'U': 4,
    'S': 0
}


# Perform A* Search from 'A' to 'F'
path = a_star_search(graph, 'P', 'S', heuristic)
```

```
print(f"Path: {path}")
```

Output:

```
Path: ['P', 'C', 'U', 'S']
[Finished in 226ms]
```

Hill climbing

```
import random

def hill_climbing(problem, initial_state):
    current = initial_state
    while True:
        neighbors = problem.get_neighbors(current)
        if not neighbors:
            break
        neighbor = max(neighbors, key=problem.value)
        if problem.value(neighbor) <= problem.value(current):
            break
        current = neighbor
    return current

class Problem:
    def __init__(self, state_space, value_function):
        self.state_space = state_space
        self.value_function = value_function

    def get_neighbors(self, state):
        neighbors = []
        for i in range(len(state)):
            if state[i] > 0:
                neighbor = state[:]
                neighbor[i] -= 1
                neighbors.append(neighbor)
            if state[i] < self.state_space[i] - 1:
                neighbor = state[:]
                neighbor[i] += 1
```

```
            neighbors.append(neighbor)
        return neighbors

    def value(self, state):
        return self.value_function(state)

# Example usage
state_space = [10, 10]  # Example state space
initial_state = [random.randint(0, 9), random.randint(0, 9)]

def value_function(state):
    # Example value function: sum of the state values
    return sum(state)

problem = Problem(state_space, value_function)
solution = hill_climbing(problem, initial_state)
print(f"Initial state: {initial_state}")
print(f"Solution: {solution}")
print(f"Value of solution: {problem.value(solution)}")
```

Output:

```
Initial state: [8, 3]
Solution: [9, 9]
Value of solution: 18
[Finished in 257ms]
```

Cryptarithmetic

```
from itertools import permutations

def is_valid_solution(perm):
    s, e, n, d, m, o, r, y = perm
    send = s * 1000 + e * 100 + n * 10 + d
    more = m * 1000 + o * 100 + r * 10 + e
    money = m * 10000 + o * 1000 + n * 100 + e * 10 + y
    return send + more == money

def solve_cryptarithmetic():
    letters = 'sendmory'
    digits = range(10)
```

```
    for perm in permutations(digits, len(letters)):
        if perm[letters.index('m')] == 0 or perm[letters.index('s')] == 0:
            continue
        if is_valid_solution(perm):
            return {letters[i]: perm[i] for i in range(len(letters))}
    return None

solution = solve_cryptarithmetic()
if solution:
    print("Solution found:")
    for letter, digit in solution.items():
        print(f"{letter.upper()} = {digit}")
else:
    print("No solution found.")
```

Output:

```
Solution found:
S = 9
E = 5
N = 6
D = 7
M = 1
O = 0
R = 8
Y = 2
[Finished in 1.3s]
```

Frame

```
class Job:
    def __init__(self, company, position, salary, location):
        self.company = company
        self.position = position
        self.salary = salary
        self.location = location

    def __str__(self):
        return (f"Company: {self.company}, Position: {self.position}, "
                f"Salary: {self.salary}, Location: {self.location}")
```

```
class Person:
    def __init__(self, name, location, birthdate, height, weight, job):
        self.name = name
        self.location = location
        self.birthdate = birthdate
        self.height = height
        self.weight = weight
        self.job = job

    def __str__(self):
        return (f"Name: {self.name}, Location: {self.location}, Birthdate: {self.birthdate}, "
                f"Height: {self.height}, Weight: {self.weight}, Job: [{self.job}]")

# Creating the Job object
ram_job = Job(company="ABC company", position="AI Researcher", salary="1.5 lakhs per month",
location="Kathmandu")

# Creating the Person object
ram = Person(name="Ram", location="Nepal", birthdate="15th December 1990", height="6 inches",
weight="75 kg", job=ram_job)

# Printing the details
print(ram)
```

Output:

```
Name: Ram, Location: Nepal, Birthdate: 15th December 1990, Height: 6 inches, Weight: 75 kg,
Job: [Company: ABC company, Position: AI Researcher, Salary: 1.5 lakhs per month, Location:
Kathmandu]
[Finished in 222ms]
```

Medical Expert System

```
class MedicalExpertSystem:
    def __init__(self):
        self.diseases = {
            "Common Cold": ["cough", "sneezing", "runny nose", "sore throat"],
            "Flu": ["fever", "chills", "muscle aches", "cough", "congestion"],
            "COVID-19": ["fever", "dry cough", "tiredness", "loss of taste or smell"],
            "Allergy": ["sneezing", "itchy eyes", "runny nose", "rash"],
            "Pneumonia": ["fever", "chills", "cough", "shortness of breath"]
        }

    def diagnose(self, symptoms):
        possible_diseases = []
        for disease, disease_symptoms in self.diseases.items():
            matching_symptoms = set(symptoms).intersection(set(disease_symptoms))
            if matching_symptoms:
                possible_diseases.append((disease, len(matching_symptoms)))

        # Sort possible diseases by the number of matching symptoms, descending
        possible_diseases.sort(key=lambda x: x[1], reverse=True)

        # Return only the disease names in the sorted order
        return [disease for disease, _ in possible_diseases]


if __name__ == "__main__":
    expert_system = MedicalExpertSystem()

    # Test with symptoms that should trigger a match
    symptoms = ["fever", "cough", "tiredness"]  # Example symptoms

    diagnosis = expert_system.diagnose(symptoms)
    if diagnosis:
        print("Possible diseases based on symptoms:", diagnosis)
    else:
        print("No matching diseases found.") class MedicalExpertSystem:
    def __init__(self):
        self.diseases = {
            "Common Cold": ["cough", "sneezing", "runny nose", "sore throat"],
```

```
            "Flu": ["fever", "chills", "muscle aches", "cough", "congestion"],
            "COVID-19": ["fever", "dry cough", "tiredness", "loss of taste or smell"],
            "Allergy": ["sneezing", "itchy eyes", "runny nose", "rash"],
            "Pneumonia": ["fever", "chills", "cough", "shortness of breath"]
        }

    def diagnose(self, symptoms):
        possible_diseases = []
        for disease, disease_symptoms in self.diseases.items():
            matching_symptoms = set(symptoms).intersection(set(disease_symptoms))
            if matching_symptoms:
                possible_diseases.append((disease, len(matching_symptoms)))

        # Sort possible diseases by the number of matching symptoms, descending
        possible_diseases.sort(key=lambda x: x[1], reverse=True)

        # Return only the disease names in the sorted order
        return [disease for disease, _ in possible_diseases]


if __name__ == "__main__":
    expert_system = MedicalExpertSystem()

    # Test with symptoms that should trigger a match
    symptoms = ["fever", "cough", "tiredness"]  # Example symptoms

    diagnosis = expert_system.diagnose(symptoms)
    if diagnosis:
        print("Possible diseases based on symptoms:", diagnosis)
    else:
        print("No matching diseases found.")
```

Output:

Possible diseases based on symptoms: ['Flu', 'COVID-19', 'Pneumonia', 'Common Cold']
[Finished in 228ms]

Realization of Logic Gates

```
def AND_gate(x1, x2):
    w1, w2, theta = 1, 1, 1.5
    y = w1 * x1 + w2 * x2
    return 1 if y >= theta else 0

def OR_gate(x1, x2):
    w1, w2, theta = 1, 1, 0.5
    y = w1 * x1 + w2 * x2
    return 1 if y >= theta else 0

def NOT_gate(x):
    w, theta = -1, -0.5
    y = w * x
    return 1 if y >= theta else 0

# Test the gates
if __name__ == "__main__":
    print("AND Gate")
    print(f"AND(0, 0) = {AND_gate(0, 0)}")
    print(f"AND(0, 1) = {AND_gate(0, 1)}")
    print(f"AND(1, 0) = {AND_gate(1, 0)}")
    print(f"AND(1, 1) = {AND_gate(1, 1)}")

    print("\nOR Gate")
    print(f"OR(0, 0) = {OR_gate(0, 0)}")
    print(f"OR(0, 1) = {OR_gate(0, 1)}")
    print(f"OR(1, 0) = {OR_gate(1, 0)}")
    print(f"OR(1, 1) = {OR_gate(1, 1)}")

    print("\nNOT Gate")
    print(f"NOT(0) = {NOT_gate(0)}")
    print(f"NOT(1) = {NOT_gate(1)}")
```

Output:

```
AND Gate
AND(0, 0) = 0
AND(0, 1) = 0
AND(1, 0) = 0
```

AND(1, 1) = 1

OR Gate
OR(0, 0) = 0
OR(0, 1) = 1
OR(1, 0) = 1
OR(1, 1) = 1

NOT Gate
NOT(0) = 1
NOT(1) = 0
[Finished in 227ms]

Back Propagation Learning:

```
import numpy as np

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)

# Training data
inputs = np.array([[0, 0],
                   [0, 1],
                   [1, 0],
                   [1, 1]])

outputs = np.array([[0], [1], [1], [0]])

# Initialize weights randomly with mean 0
np.random.seed(1)
input_layer_neurons = inputs.shape[1]
hidden_layer_neurons = 2
output_layer_neurons = 1
```

```
# Weights and biases
hidden_weights = np.random.uniform(size=(input_layer_neurons, hidden_layer_neurons))
hidden_bias = np.random.uniform(size=(1, hidden_layer_neurons))
output_weights = np.random.uniform(size=(hidden_layer_neurons, output_layer_neurons))
output_bias = np.random.uniform(size=(1, output_layer_neurons))

# Learning rate
lr = 0.1

# Training the neural network
for epoch in range(10000):
    # Forward propagation
    hidden_layer_activation = np.dot(inputs, hidden_weights)
    hidden_layer_activation += hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output, output_weights)
    output_layer_activation += output_bias
    predicted_output = sigmoid(output_layer_activation)

    # Backpropagation
    error = outputs - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(output_weights.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # Updating weights and biases
    output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
    output_bias += np.sum(d_predicted_output, axis=0, keepdims=True) * lr
    hidden_weights += inputs.T.dot(d_hidden_layer) * lr
    hidden_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) * lr

# Output after training
print("Output after training:")
print(predicted_output)
```

Output

```
Output after training:
[[0.06368082]
 [0.94085536]
 [0.94108726]
 [0.06402009]]
[Finished in 2.1s]
```

N Queen Problem

```
def print_solution(board):
    for row in board:
        print(" ".join(str(x) for x in row))
    print()

def is_safe(board, row, col):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_nq_util(board, col):
    if col >= len(board):
        return True
```

```
    for i in range(len(board)):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_nq_util(board, col + 1):
                return True
            board[i][col] = 0

    return False

def solve_nq(n):
    board = [[0 for _ in range(n)] for _ in range(n)]
    if not solve_nq_util(board, 0):
        print("Solution does not exist")
        return False

    print_solution(board)
    return True

# Example usage
n = 4
solve_nq(n)
```

Output

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

[Finished in 228ms]
```

Water Jug Problem

```
from collections import deque

def water_jug_BFS():
    # Initialize the queue with the starting state (0, 0)
    queue = deque([(0, 0)])
    visited = set((0, 0))

    while queue:
        a, b = queue.popleft()

        # If we have exactly 2 gallons in the 4-gallon jug, return the solution
        if a == 2:
            return True

        # Possible states after performing the operations
        states = [
            (4, b),  # Fill the 4-gallon jug
            (a, 3),  # Fill the 3-gallon jug
            (0, b),  # Empty the 4-gallon jug
            (a, 0),  # Empty the 3-gallon jug
            (a - min(a, 3 - b), b + min(a, 3 - b)),  # Pour water from 4-gallon to 3-gallon jug
            (a + min(b, 4 - a), b - min(b, 4 - a))   # Pour water from 3-gallon to 4-gallon jug
        ]

        for state in states:
            if state not in visited:
                visited.add(state)
                queue.append(state)

    return False

# Run the function
if water_jug_BFS():
    print("Solution found: You can get exactly 2 gallons in the 4-gallon jug.")
else:
    print("No solution exists.")
```

Output

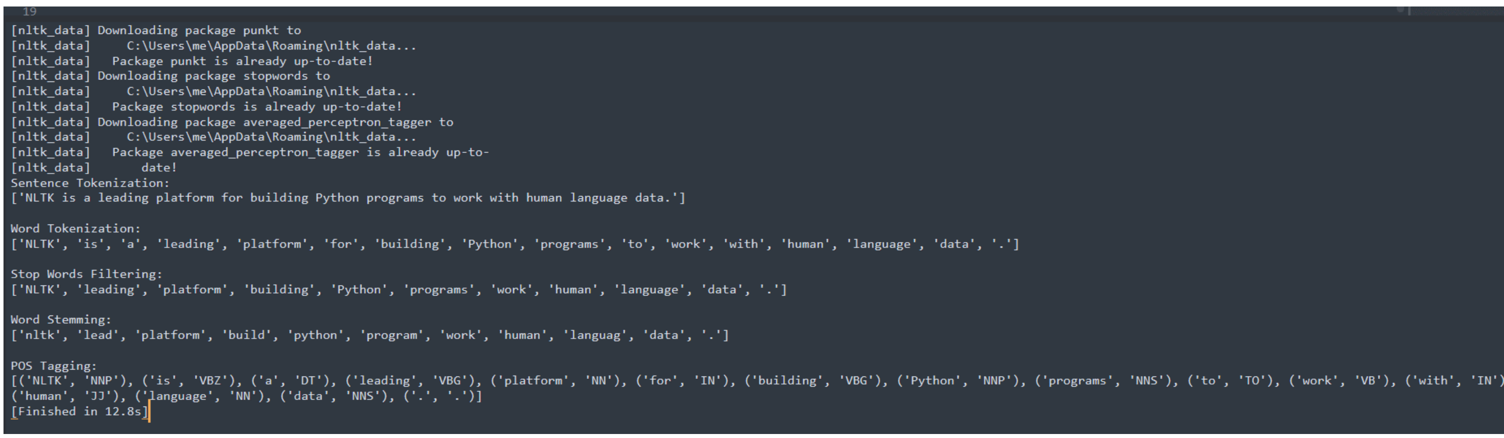
Solution found: You can get exactly 2 gallons in the 4-gallon jug.
[Finished in 226ms]

NLP tasks

```
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.tag import pos_tag

# Download necessary NLTK data files
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger')

# Sample text
text = "NLTK is a leading platform for building Python programs to work with human language data."

# Sentence Tokenization
sentences = sent_tokenize(text)
print("Sentence Tokenization:")
print(sentences)

# Word Tokenization
words = word_tokenize(text)
print("\nWord Tokenization:")
print(words)

# Stop Words Filtering
stop_words = set(stopwords.words('english'))
filtered_words = [word for word in words if word.lower() not in stop_words]
print("\nStop Words Filtering:")
print(filtered_words)

# Word Stemming
```

```
ps = PorterStemmer()
stemmed_words = [ps.stem(word) for word in filtered_words]
print("\nWord Stemming:")
print(stemmed_words)

# POS Tagging
pos_tags = pos_tag(words)
print("\nPOS Tagging:")
print(pos_tags)
```

Output:



```
19
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\me\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\me\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\me\AppData\Roaming\nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
Sentence Tokenization:
['NLTK is a leading platform for building Python programs to work with human language data.']

Word Tokenization:
['NLTK', 'is', 'a', 'leading', 'platform', 'for', 'building', 'Python', 'programs', 'to', 'work', 'with', 'human', 'language', 'data', '.']

Stop Words Filtering:
['NLTK', 'leading', 'platform', 'building', 'Python', 'programs', 'work', 'human', 'language', 'data', '.']

Word Stemming:
['nltk', 'lead', 'platform', 'build', 'python', 'program', 'work', 'human', 'languag', 'data', '.']

POS Tagging:
[('NLTK', 'NNP'), ('is', 'VBZ'), ('a', 'DT'), ('leading', 'VBG'), ('platform', 'NN'), ('for', 'IN'), ('building', 'VBG'), ('Python', 'NNP'), ('programs', 'NNS'), ('to', 'TO'), ('work', 'VB'), ('with', 'IN')
('human', 'JJ'), ('language', 'NN'), ('data', 'NNS'), ('.', '.')]
[Finished in 12.8s]
```