ARTIFICIAL INTELLIGENCE
[CSE401]

SUBMITTED TO:
Dr. Garima Aggarwal

SUBMITTED BY:
Ananta Walli
A2305221322
6CSE6X

COMPUTER SCIENCE AND ENGINEERING
AMITY SCHOOL OF ENGINEERING & TECHNOLOGY
AMITY UNIVERSITY, UTTAR PRADESH

Index

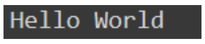
EXPERIMENT 0: BASICS OF PYTHON

Aim: Write a program to implement basics of python.

Language Used: Python

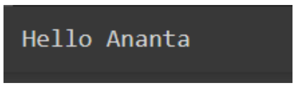
Code Along with Output:

```
#print statment
print("Hello World")
```



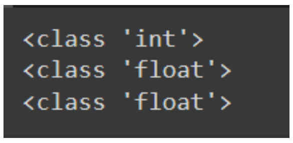
```
Hello World
```

```
#declaring a string variable
name ="Ananta"
print("Hello " +name)
```



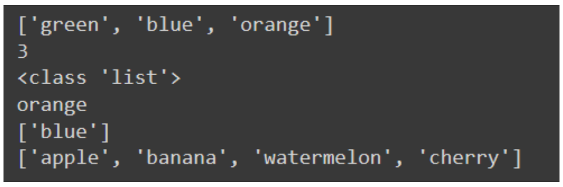
```
Hello Ananta
```

```
#python numbers
a = 1
b= 2.34
c=-23.11
print(type(a))
print(type(b))
print(type(c))
```



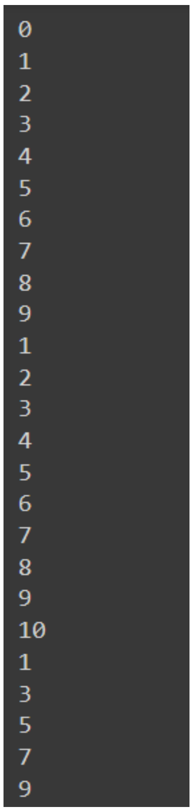
```
<class 'int'>
<class 'float'>
<class 'float'>
```

```
#Lists
testlist =["green","blue","orange"]
print(testlist)
print(len(testlist))
print(type(testlist))
print(testlist[2])
print(testlist[1:2])

thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```



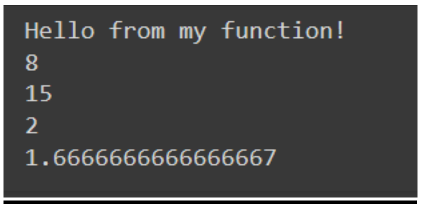
```
['green', 'blue', 'orange']
3
<class 'list'>
orange
['blue']
['apple', 'banana', 'watermelon', 'cherry']
```

```
# Loops

for x in range(10):
  print(x)

for x in range(1,11):
  print(x)

for x in range(1,11,2):
  print(x)
```



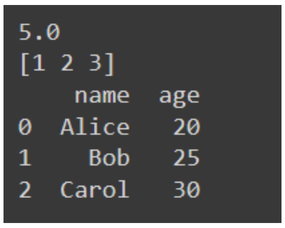
```
0
1
2
3
4
5
6
7
8
9
1
2
3
4
5
6
7
8
9
10
1
3
5
7
9
```

```
# Functions

def my_function():
  print("Hello from my function!")
my_function()

def add(a, b):
  return a + b
print(add(5, 3))

def multiply(a, b):
  return a * b
print(multiply(5, 3))

def subtract(a, b):
  return a - b
print(subtract(5, 3))

def divide(a, b):
  return a / b
print(divide(5, 3))
```



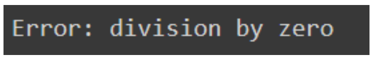
```
Hello from my function!
8
15
2
1.6666666666666667
```

```
# Modules and libraries

import math
print(math.sqrt(25))
import numpy as np
print(np.array([1, 2, 3]))
import pandas as pd
print(pd.DataFrame({"name": ["Alice", "Bob", "Carol"], "age": [20, 25, 30]}))
```



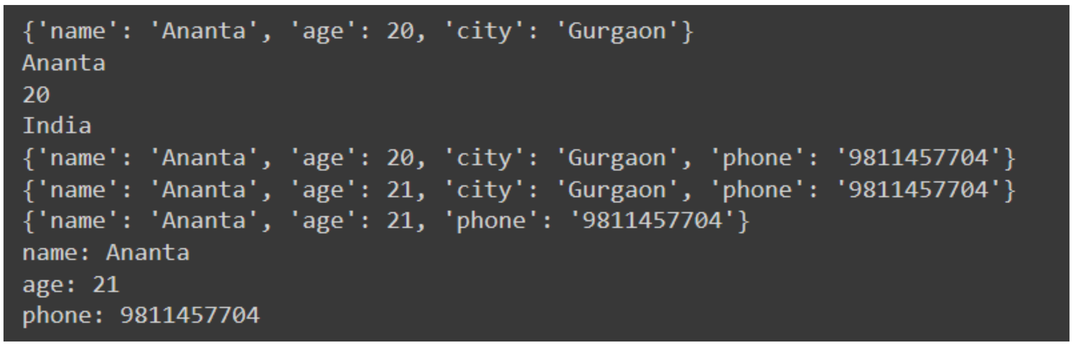
```
5.0
[1 2 3]
    name  age
0  Alice   20
1    Bob   25
2  Carol   30
```

```
#exception handling

try:
  print(10/0)
except ZeroDivisionError as e:
  print("Error:", e)
```



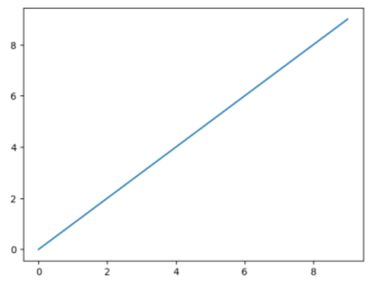
```
Error: division by zero
```

```
# dictionary in python

dict1 = {
  "name": "Ananta",
  "age": 20,
  "city": "Gurgaon"
}
print(dict1)
print(dict1["name"])
print(dict1.get("age"))
print(dict1.get("country", "India"))


dict1["phone"] = "9811457704"
print(dict1)


dict1["age"] = 21
print(dict1)


del dict1["city"]
print(dict1)

# Iterating over a dictionary
for key, value in dict1.items():
  print(f"{key}: {value}")
```

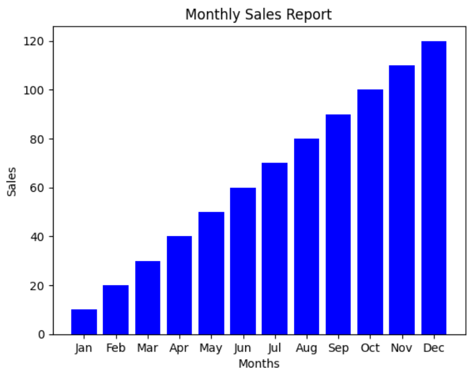


```
{'name': 'Ananta', 'age': 20, 'city': 'Gurgaon'}
Ananta
20
India
{'name': 'Ananta', 'age': 20, 'city': 'Gurgaon', 'phone': '9811457704'}
{'name': 'Ananta', 'age': 21, 'city': 'Gurgaon', 'phone': '9811457704'}
{'name': 'Ananta', 'age': 21, 'phone': '9811457704'}
name: Ananta
age: 21
phone: 9811457704
```

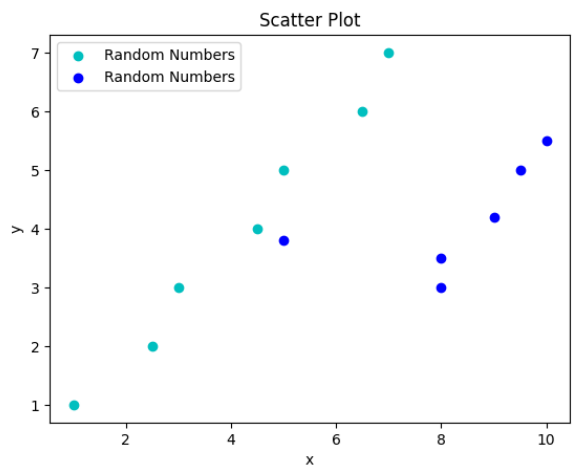
```
#Numpy and Matplotlib

import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 10) #array using numpy

#Plotting
plt.plot(x)
plt.show()
```



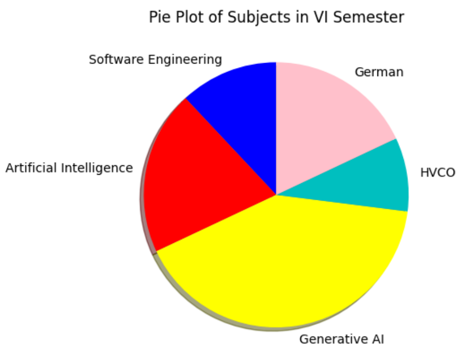
```
# Bar Graph

import matplotlib.pyplot as plt
import numpy as np

x = np.array(["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov",
"Dec"])
y = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120])

plt.bar(x, y, color='blue')
plt.xlabel('Months')
plt.ylabel('Sales')
plt.title('Monthly Sales Report')
plt.show()
```



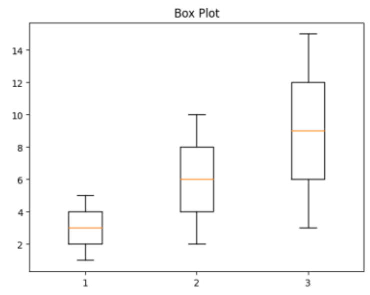
```
# Scatter plot
import matplotlib.pyplot as plt

x1 = [1, 2.5, 3, 4.5, 5, 6.5, 7]
y1 = [1, 2, 3, 4, 5, 6, 7]
x2 = [8, 8, 5, 9, 9.5, 10]
y2 = [3, 3.5, 3.8, 4.2, 5, 5.5]

plt.scatter(x1, y1, label="Random Numbers", color='c')
plt.scatter(x2, y2, label="Random Numbers", color='b')
plt.title('Scatter Plot')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.show()
```



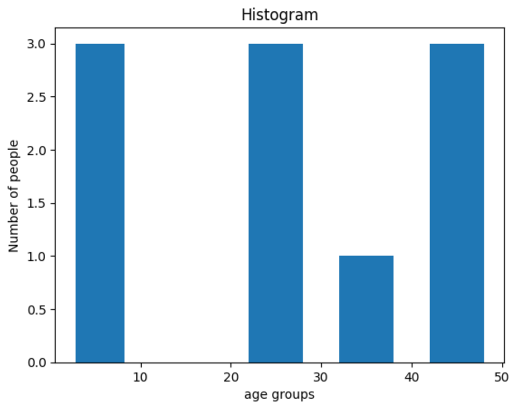
```
import matplotlib.pyplot as plt

slice = [12, 20, 41, 9, 18]
activities = ['Software Engineering', 'Artificial Intelligence', 'Generative AI', 'HVCO',
'German']
cols = ['b', 'r', 'yellow', 'c', 'pink']

plt.pie(slice, labels=activities, colors=cols, startangle=90, shadow=True)
plt.title('Pie Plot of Subjects in VI Semester')
plt.show()
```



```
# Box Plot

data = [
    [1, 2, 3, 4, 5],
    [2, 4, 6, 8, 10],
    [3, 6, 9, 12, 15]
]

plt.boxplot(data)
plt.title('Box Plot')
plt.show()
```



```
# Histogram

import matplotlib.pyplot as plt
pop = [22,55,62,45,21,22,34,42,42,4,2,8]
bins = [1,10,20,30,40,50]
pyplot.hist(pop, bins, rwidth=0.6)
pyplot.xlabel('age groups')
pyplot.ylabel('Number of people')
pyplot.title('Histogram')

pyplot.show()
```



Conclusion: Hence, the basics of python are implemented.

EXPERIMENT- 1

Aim: Write a program to implement single user guessing number game.

Language Used: Python

Theory:

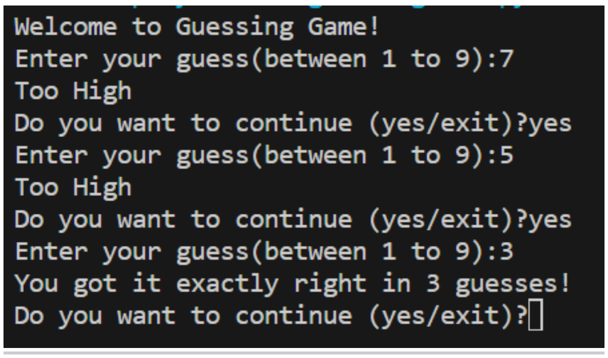
1. **Game Initialization and Definition:**
    - The game, known as "Guessing Game-One," is a simple number guessing game designed for user interaction.
    - To initialize, the computer generates a random number between 1 and 9 (inclusive), which serves as the target or correct answer.
    - The objective of the game is for the user to guess the correct number within the specified range.
2. **User Input:**
    - The user is prompted to enter their guess, providing an opportunity to interact with the game.
3. **State Representation:**
    - The state of the game is defined by two key variables:
        - number: The randomly generated target number.
        - user: The current user's guess.
4. **Conditions and Feedback:**
    - The game evaluates the user's guess in relation to the target number, providing specific feedback:
        - If user == number: The user has successfully guessed the correct number, and the game congratulates them.
        - If user > number: The user's guess is too high, prompting the game to inform them of this fact.
        - If user < number: The user's guess is too low, and the game informs them accordingly.
        - An error message is displayed if none of the above conditions are met.
5. **Game Loop:**
    - The game features a loop that continues until the user decides to exit. After each guess, the user has the option to continue or exit.
6. **Guess Count:**
    - The game keeps track of the number of guesses made by the user. This count is incremented with each guess.
7. **Exit Condition:**
    - The user is given the choice to continue playing or exit the game. If the user chooses to exit, the game concludes.
8. **Conclusion:**
    - The game concludes with a "Thank you for playing!" message, providing a clear endpoint to the user's interaction with the game.

Code:

```
import random
```

```
number = random.randint(1, 9)
print("Welcome to Guessing Game!")
answer = "yes"
guess_count = 0

while answer == "yes":
    user = int(input("Enter your guess(between 1 to 9):"))
    guess_count += 1

    if user == number:
        print(f"You got it exactly right in {guess_count} guesses!")
    elif user > number:
        print("Too High")
    elif user < number:
        print("Too Low")
    else:
        print("Error")

    answer = input("Do you want to continue (yes/exit)?")

print("Thank you for playing!")
```

Output:



```
Welcome to Guessing Game!
Enter your guess(between 1 to 9):7
Too High
Do you want to continue (yes/exit)?yes
Enter your guess(between 1 to 9):5
Too High
Do you want to continue (yes/exit)?yes
Enter your guess(between 1 to 9):3
You got it exactly right in 3 guesses!
Do you want to continue (yes/exit)?
```

Conclusion: Hence the desired game is executed in python.

EXPERIMENT 2- UNIFORMED SEARCH TECHNIQUES

EXPERIMENT- 2.1

Aim: Write a program to implement water-jug problem using BFS.

Language Used: Python

Theory:

The Water Jug Problem is a classic problem in artificial intelligence and computer science that involves finding a sequence of actions to measure a specific amount of water using a set of jugs with known capacities. The goal is to determine a series of pouring actions that will result in one or more jugs containing the desired amount of water. The problem is often used as an illustration of problem-solving strategies, such as searching algorithms.

State Representation: Define the state of the problem. A state is a configuration of water levels in the jugs. For example, a state can be represented as (a, b), where 'a' is the water level in Jug A and 'b' is the water level in Jug B.

Action Representation:
Define the set of possible actions that can be taken. Actions involve pouring water from one jug to another or filling/emptying a jug. Represent actions as tuples (a, b), where 'a' is the source jug, and 'b' is the target jug.

Initial State:
Specify the initial state of the problem, i.e., the starting configuration of water levels in the jugs. (0,0)

Goal State:
Define the goal state, i.e., the configuration of water levels that satisfies the problem requirements. (2,0)

Path Cost:
Assign costs to actions if applicable. For example, pouring water might have a cost associated with it, and the objective could be to find the solution with the minimum total cost.
Rules for curing jugs:
Fill 4-gallon jug: (4,0)
Fill 3-gallon jug: (3,0)
Empty 4-gallon jug: (0,y)
Empty 3-gallon jug: (x,0)
Pour 4 gallon in 3 gallon: (4-x,x)
Pour 3 gallon in 4 gallon: (y,3-y)

Search Algorithm:
Choose an appropriate search algorithm to find a sequence of actions from the initial state to the goal state. Common search algorithms include Breadth-First Search, Depth-First Search, and A* Search. In this case we will use BFS.

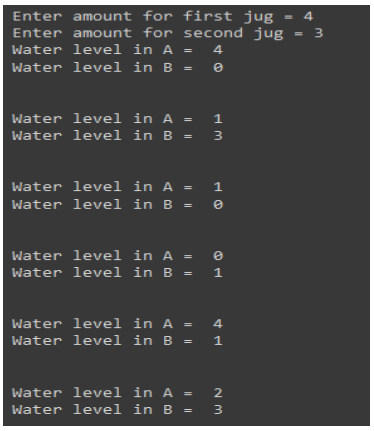
Optimality and Completeness:

Analyse the optimality and completeness of the chosen search algorithm. Optimality refers to finding the most cost-effective solution, and completeness ensures that a solution is guaranteed to be found if one exists.

Code:

```
global amount1
global a
global amount2
global b

amount1 = int(input("Enter amount for first jug = "))
amount2 = int(input("Enter amount for second jug = "))

a = 0
b = 0

def fill_jug1():
    global amount1, amount2,a,b
    a = amount1
    print("Water level in A = ",a)
    print("Water level in B = ",b)

def fill_jug2():
    global amount1, amount2,a,b
    b =  amount2
    print("Water level in A = ",a)
    print("Water level in B = ",b)

def empty_jug1():
    global amount1, amount2,a,b
    a = 0
    print("Water level in A = ",a)
    print("Water level in B = ",b)

def empty_Jug2():
    global amount1, amount2,a,b
    b = 0
    print("Water level in A = ",a)
    print("Water level in B = ",b)

def transderJug1toJug2():
    global amount1, amount2,a,b
    if (b == 0 and a >=  amount2):
        b =  amount2
        a = a -  amount2
    elif (b == 0 and a <  amount2):
        b = a
        a = 0
    elif (b > 0 and b <  amount2 and ( amount2-b)<a):
        a = a-( amount2-b)
        b = b+( amount2-b)
    elif (b > 0 and b <  amount2 and ( amount2-b)>a):
        b = b + a
        a = 0
    print("\n")
    print("Water level in A = ",a)
    print("Water level in B = ",b)
    print("\n")

def transferJug2toJug1():
    global amount1, amount2,a,b
    if (a == 0 and b ==  amount2):
        a =  amount2
        b = 0
    elif (a == 0 and b <  amount2):
        a = b
        b = 0
    elif (a > 0 and a + b <= amount1):
        a = a+b
        b = 0
    elif (a > 0 and a + b > amount1):
        b = b - (amount1-a)
        a = amount1

    print("Water level in A = ",a)
    print("Water level in B = ",b)


while (a != 2):
    if (a == 0):
        fill_jug1()
    if (b ==  amount2):
        empty_Jug2()
    transderJug1toJug2()

print("Water level in A = ",a)
print("Water level in B = ",b)
```

Output:



```
Enter amount for first jug = 4
Enter amount for second jug = 3
Water level in A =  4
Water level in B =  0


Water level in A =  1
Water level in B =  3


Water level in A =  1
Water level in B =  0


Water level in A =  0
Water level in B =  1


Water level in A =  4
Water level in B =  1


Water level in A =  2
Water level in B =  3
```

Conclusion: Hence using BFS the desired solution is obtained in the Water-jug Problem.

EXPERIMENT- 2.2

Aim: Write a program to implement recursive maze problem using DFS.

Language Used: Python

Theory:

The Recursive Maze Problem is a classic computational problem that involves navigating through a maze using recursive techniques. In this context, a maze is a network of passages or paths with walls, dead-ends, and a designated start and end point. The goal is to find a path from the initial state (starting point) to the goal state (ending point) while navigating through the maze.

Movement: The explorer can typically move in four directions - up, down, left, and right. Some mazes might allow diagonal movements as well.

Walls: The maze contains walls that restrict the movement of the explorer. The explorer cannot pass through walls, and navigating around them is necessary to reach the goal.

Initial State: The explorer starts at an initial state, representing the starting point within the maze ie n x n matrix.

Goal State: There is a designated goal state, representing the destination or endpoint that the explorer aims to reach ie n-1 x n-1 matrix.

Path: The objective is to find a path from the initial state to the goal state. The path should navigate through open passages while avoiding walls.

Obstacles: In addition to walls, there might be other obstacles or features in the maze that the explorer needs to navigate around.

Path Cost: The cost associated with traversing each path may vary. It could be determined by the number of steps taken, distance traveled, or other measures of effort.

Search Algorithms: The choice of a search algorithm, such as Depth-First Search (DFS) or others, influences the strategy employed to explore the maze and find a solution.

Completeness and Optimality: The completeness and optimality of the chosen algorithm determine whether a solution will be found if it exists and if it will be the most efficient one.

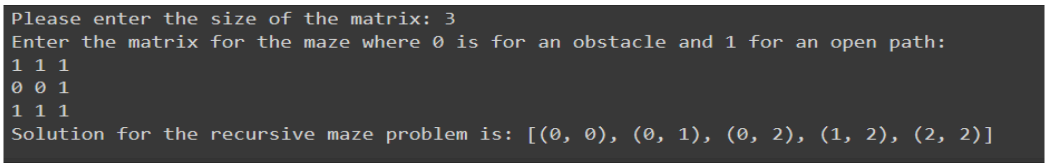
Backtracking: In algorithms like DFS, backtracking is a crucial concept. If the explorer reaches a dead-end or encounters an obstacle, it must backtrack to the most recent decision point and explore alternative paths.

Code:

```
def move(a, b, m, vis):
    k = len(m)   # square matrix
    return 0 <= a < k and 0 <= b < k and m[a][b] == 1 and not vis[a][b]
```

```
def maze_problem(m):
    k = len(m)
    vis = [[False for _ in range(k)] for _ in range(k)]

    def dfs(a, b):
        if a == k - 1 and b == k - 1:
            return [(a, b)]

        vis[a][b] = True
        #define moves in directions- right, down, left, up
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        for df, dg in directions:
            f, g = a + df, b + dg
            if move(f, g, m, vis):
                path = dfs(f, g)
                if path:
                    return [(a, b)] + path
        return []

    solution = dfs(0, 0)  # initial state

    if solution:
        return solution
    else:
        return "Path does not exist"

# User input block
k = int(input("Please enter the size of the matrix: "))
maze = []

print("Enter the matrix for the maze where 0 is for an obstacle and 1 for an open path:")
for _ in range(k):
    r = list(map(int, input().split()))
    maze.append(r)

sol_path = maze_problem(maze)
print("Solution for the recursive maze problem is:", sol_path)
```

Output:



```
Please enter the size of the matrix: 3
Enter the matrix for the maze where 0 is for an obstacle and 1 for an open path:
1 1 1
0 0 1
1 1 1
Solution for the recursive maze problem is: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2)]
```

Conclusion: Hence using DFS, desired solution is obtained in Recursive Maze Problem.

EXPERIMENT-3: INFORMED SEARCH TECHNIQUES

EXPERIMENT 3.1

Aim: Write a program to implement 8 puzzle problem using best first search.

Language Used: Python

Theory:

The 8-puzzle problem is a classic problem in artificial intelligence involving a 3x3 grid with eight numbered tiles and an empty space. The goal is to reach a specified goal state from a given initial state by moving tiles into the empty space. Best-first search is one of the fundamental search algorithms used to solve this problem. Below is a theoretical explanation of how best-first search can be applied to solve the 8-puzzle problem, including the initial and goal states.

Initial State:

Let's represent the initial state of the 8-puzzle problem as follows:

```
1  2  3
4  0  5
7  8  6
```

In this representation, the empty space is denoted by the symbol 0.

Goal State:

The goal state of the 8-puzzle problem is typically defined as follows:

```
1  2  3
4  5  6
7  8  0
```

Applying best-first search algorithm:

Best-first search is a graph search algorithm that explores nodes based on an evaluation function. In the case of the 8-puzzle problem, the evaluation function usually measures the closeness of a state to the goal state.

Steps:

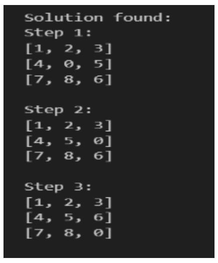
1. **Initialize**: Start with the initial state of the puzzle.
2. **Evaluate**: Compute the evaluation function for the initial state.
3. **Expand**: Generate all possible successor states by moving tiles in the puzzle.
4. **Evaluate**: Compute the evaluation function for each successor state.
5. **Sort**: Sort the successor states based on their evaluation function values.
6. **Select**: Choose the successor state with the lowest evaluation function value.
7. **Repeat**: Repeat steps 3-6 until either the goal state is reached or no more states can be expanded.

8. This process continues until the goal state is reached, at which point the algorithm terminates.

Code:

```
import heapq

class Puzzle:
    def __init__(self, puzzle_state, parent=None, move=None):
        self.puzzle_state = puzzle_state
        self.parent = parent
        self.move = move
        self.cost = self.calculate_cost()

    def calculate_cost(self):
        goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
        cost = 0
        for i in range(3):
            for j in range(3):
                if self.puzzle_state[i][j] != 0:
                    row, col = divmod(self.puzzle_state[i][j] - 1, 3)
                    cost += abs(row - i) + abs(col - j)
        return cost

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

def get_neighbors(node):
    neighbors = []
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    for i in range(3):
        for j in range(3):
            if node.puzzle_state[i][j] == 0:
                zero_row, zero_col = i, j
                break

    for direction in directions:
        new_row, new_col = zero_row + direction[0], zero_col + direction[1]
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_state = [row[:] for row in node.puzzle_state]
            new_state[zero_row][zero_col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[zero_row][zero_col]
            neighbors.append(PuzzleNode(new_state, node, direction))
    return neighbors

def reconstruct_path(node):
    path = []
    while node:
        path.append(node.puzzle_state)
        node = node.parent
    return path[::-1]

def best_first_search(initial_state):
    heap = []
    visited_states = set()
    heapq.heappush(heap, initial_state)

    while heap:
        current_node = heapq.heappop(heap)
        visited_states.add(tuple(map(tuple, current_node.puzzle_state)))

        if current_node.puzzle_state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]:
            return reconstruct_path(current_node)

        for neighbor in get_neighbors(current_node):
            if tuple(map(tuple, neighbor.puzzle_state)) not in visited_states:
                heapq.heappush(heap, neighbor)
                visited_states.add(tuple(map(tuple, neighbor.puzzle_state)))

    return None

initial_state = PuzzleNode([[1, 2, 3], [4, 0, 5], [7, 8, 6]])
solution = best_first_search(initial_state)

if solution:
    print("Solution found:")
    for i, state in enumerate(solution):
        print(f"Step {i + 1}:")
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

Output:



```
Solution found:
Step 1:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

Step 2:
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

Step 3:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

Conclusion: Hence using Best first search, desired solution is obtained in 8 Puzzle Problem.

EXPERIMENT 3.2

Aim: Write a program to implement 8 puzzle problem using A* algorithm.

Language Used: Python

Theory:

The 8 puzzle problem is a sliding puzzle that consists of a 3x3 grid with 8 numbered tiles and an empty space. The objective is to move the tiles to rearrange them into the goal state using the minimum number of moves. Tiles can only be moved into the empty space, and the allowed moves are up, down, left, and right.

The A* algorithm is a widely used search algorithm that finds the shortest path from a start state to a goal state in a weighted graph. It uses both the actual cost from the start node and a heuristic estimate of the remaining cost to the goal node to guide its search. In this implementation, the heuristic function is used, which calculates the sum of the distances of each tile from its goal position.

Steps:

1. **Initialization**: The initial state of the puzzle is provided, along with the goal state.
2. **Node Representation**: Each state of the puzzle is represented as a node in a search tree. Each node contains the puzzle state, a reference to its parent node, the move that led to this state, and the depth in the search tree.
3. **Frontier and Explored Sets**: A priority queue is used to store nodes that are candidates for expansion. Nodes are prioritized based on their cost (depth + heuristic value). Explored set is used to keep track of already visited nodes.
4. **Expansion and Heuristic Evaluation**: Nodes are expanded one by one. For each node, possible actions (up, down, left, right) are determined, and new states are generated. Each new state is evaluated using the heuristic function (Manhattan distance) to estimate its cost.
5. **Search**: A* iteratively expands the node with the lowest total cost (actual cost + heuristic) until the goal state is reached or no more nodes are left in the frontier.
6. **Backtracking**: Once the goal state is reached, the solution path is reconstructed by tracing back through the parent nodes from the goal node to the initial node.

Final State:

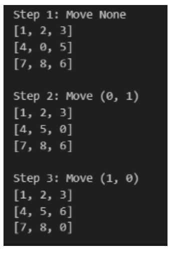
The final state represents the configuration of the puzzle after it has been solved. It should match the goal state.

Code:

```
class PuzzleusingAstar:
    def __init__(self, puzzle_state, parent=None, move=None, g=0):
        self.puzzle_state = puzzle_state
        self.parent = parent
        self.move = move
        self.g = g
        self.h = self.calculate_heuristic()
        self.f = self.g + self.h

    def calculate_heuristic(self):
        goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
        h = 0
        for i in range(3):
            for j in range(3):
                if self.puzzle_state[i][j] != 0:
                    row, col = divmod(self.puzzle_state[i][j] - 1, 3)
                    h += abs(row - i) + abs(col - j)
        return h

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

    def __le__(self, other):
        return self.f <= other.f

    def __gt__(self, other):
        return self.f > other.f

    def __ge__(self, other):
        return self.f >= other.f

    def __eq__(self, other):
        return self.f == other.f

    def __ne__(self, other):
        return self.f != other.f

def get_neighbors(node):
    neighbors = []
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    for i in range(3):
        for j in range(3):
            if node.puzzle_state[i][j] == 0:
                zero_row, zero_col = i, j
                break

    for direction in directions:
        new_row, new_col = zero_row + direction[0], zero_col + direction[1]
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_state = [row[:] for row in node.puzzle_state]
            new_state[zero_row][zero_col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[zero_row][zero_col]
            move = (new_row - zero_row, new_col - zero_col)
            neighbors.append(PuzzleusingAstar(new_state, node, move, node.g + 1))
    return neighbors
```

```python
def reconstruct_path(node):
    path = []
    while node:
        path.append((node.move, node.puzzle_state))
        node = node.parent
    return path[::-1]

def a_star_search(initial_state):
    open_list = [initial_state]
    closed_set = set()

    while open_list:
        open_list.sort(key=lambda x: x.f)
        current_node = open_list.pop(0)

        if current_node.puzzle_state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]:
            return reconstruct_path(current_node)

        closed_set.add(tuple(map(tuple, current_node.puzzle_state)))

        for neighbor in get_neighbors(current_node):
            if tuple(map(tuple, neighbor.puzzle_state)) not in closed_set:
                open_list.append(neighbor)

    return None

initial_state = PuzzleusingAstar([[1, 2, 3], [4, 0, 5], [7, 8, 6]])
solution = a_star_search(initial_state)
if solution:
    print("Solution found:")
    for i, (move, state) in enumerate(solution):
        print(f"Step {i + 1}: Move {move}")
        for row in state:
            print(row)
        print()
else:
    print("No solution found")
```

Output:



```
Step 1: Move None
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

Step 2: Move (0, 1)
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

Step 3: Move (1, 0)
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

Conclusion: Hence using A* algorithm, desired solution is obtained in 8 puzzle problem.

EXPERIMENT 4: CONSTRAINT SATISFACTION PROBLEM

EXPERIMENT 4.1

Aim: Write a program to implement crypt arithmetic problem.

Language Used: Python

Theory:

The cryptarithmetic problem is a classic example in recreational mathematics where each letter represents a unique digit, and the goal is to find the digit substitution that satisfies the equation.

Initial State:

- Cryptarithmetic equation: SEND + MORE = MONEY
- Variables: S, E, N, D, M, O, R, Y represent digits (0-9).

Final State:

- A digit substitution that satisfies the equation SEND + MORE = MONEY, where each letter represents a unique digit from 0 to 9.

Steps:

1. Constraints:
    - All letters represent digits from 0 to 9, and no two letters represent the same digit.
    - Since the sum of two 4-digit numbers (SEND and MORE) equals another 5-digit number (MONEY), the digit substitution must be such that the sum of the two 4-digit numbers results in a 5-digit number. This implies that M must be either 1 or 2 (because if M = 0, the sum will not be a 5-digit number).
    - Other constraints arise from carrying over digits in addition. For example, when E + R yields a number greater than 9, it requires carrying over to the next column.
2. Approach:
    - The problem can be solved through a systematic approach, typically involving backtracking or constraint satisfaction algorithms.
    - Start with the assumption that M equals 1. Explore all possible digit substitutions for the other letters under this assumption. If no solution is found, try M equals 2.
3. Backtracking:
    - Begin by assigning digits to letters, starting with the most significant digits (M, S).
    - Proceed with assigning digits to other letters, checking for validity as you go.
    - If a conflict arises (e.g., two letters are assigned the same digit), backtrack and try a different digit.
    - Continue this process until all letters are assigned valid digits or until it's evident that the current path leads to a contradiction. If a contradiction is reached, backtrack to the previous decision point and explore a different path.

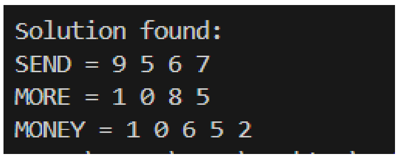
4. Optimization:
   - The problem space can be pruned by recognizing patterns and constraints. For instance, certain letter combinations can only result in specific digit assignments. Identifying and exploiting such patterns can significantly reduce the search space.
5. Solution Verification:
   - Once a potential solution is found, verify it by substituting the digits back into the original equation and ensuring that it holds true.

By systematically exploring the possible digit substitutions and utilizing backtracking or constraint satisfaction techniques, it's possible to find the solution that satisfies the cryptarithmetic equation SEND + MORE = MONEY.

Code:

```
def crypt_a():

    for s in range(1, 10):
        for e in range(10):
            for n in range(10):
                for d in range(10):
                    for m in range(1, 10):
                        for o in range(10):
                            for r in range(10):
                                for y in range(10):
                                    if len(set([s, e, n, d, m, o, r, y])) == 8:
                                        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
                                        if send + more == money:
                                            return {'S': s, 'E': e, 'N': n, 'D': d, 'M': m, 'O': o, 'R': r, 'Y': y}

    return None
solution = solve_cryptarithmetic()
if solution:
    print("Solution found:")
    print("SEND =", solution['S'], solution['E'], solution['N'], solution['D'])
    print("MORE =", solution['M'], solution['O'], solution['R'], solution['E'])
    print("MONEY =", solution['M'], solution['O'], solution['N'], solution['E'], solution['Y'])
else:
    print("No solution found.")
```

Output:



```
Solution found:
SEND = 9 5 6 7
MORE = 1 0 8 5
MONEY = 1 0 6 5 2
```

Conclusion: Hence the crypt arithmetic problem is solved where: S= 9, E= 5, N=6, D =7, M=1, O=0, R=8, Y=2.

EXPERIMENT 4.2

Aim: Write a program to implement graph colouring problem.

Language Used: Python

Theory:

Graph coloring problem is one of the fundamental problems in graph theory. It involves assigning colours to the vertices of a graph in such a way that no two adjacent vertices share the same colour.

Given an undirected graph G=(V,E)G=(V,E), where VV is the set of vertices and EE is the set of edges, the goal is to find a proper coloring for the vertices of the graph.

A proper coloring is an assignment of colors to vertices such that no two adjacent vertices share the same color. The minimum number of colors required to properly color the graph is known as the chromatic number of the graph.

The initial state of the graph coloring problem involves the following:

1. **Graph Representation**: The input graph is represented either as an adjacency matrix or as an adjacency list. In programming, adjacency lists are commonly used for efficient representation.
2. **Coloring**: Initially, no colors are assigned to any vertex.

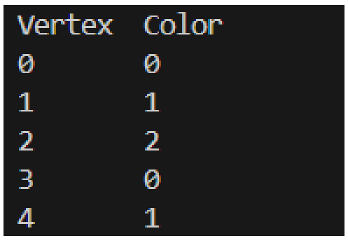
The constraints of the graph coloring problem include:

1. **Adjacent Vertices Constraint**: No two adjacent vertices in the graph can be assigned the same color. This constraint ensures that adjacent vertices are distinguishable from each other.
2. **Minimization of Colors Constraint**: The objective is to minimize the number of colors used to properly color the graph. Minimizing the number of colors reflects the efficiency and optimality of the coloring.

Code:

```
class GraphColouring:
  def __init__(self, vertices):
    self.V = vertices
    self.graph = [[] for _ in range(vertices)]

  def add_edge(self, u, v):
    self.graph[u].append(v)
    self.graph[v].append(u)

  def greedy_coloring(self):
    result = [-1] * self.V
    result[0] = 0  #Assign the 1st colour

        def is_safe(v, c):
            for i in self.graph[v]:
                if result[i] == c:
                    return False
            return True

        for u in range(1, self.V):
            for c in range(self.V):
                if is_safe(u, c):
                    result[u] = c
                    break

        return result


if __name__ == "__main__":
    # Create a graph
    g = Graph(5)
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(1, 3)
    g.add_edge(2, 3)
    g.add_edge(3, 4)

    colors = g.greedy_coloring()

    print("Vertex\tColor")
    for i in range(len(colors)):
        print(f"{i}\t{colors[i]}")
```

Output:



```
Vertex  Color
0       0
1       1
2       2
3       0
4       1
```

Conclusion: Hence, the problem of graph colouring is solved using the code.

EXPERIMENT 5: GAME THEORY

EXPERIMENT 5.1: MIN MAX ALGORITHM

Aim: Write a program to implement minimum maximum algorithm.

Language Used: Python

Theory:

The Minimax algorithm is a decision-making algorithm used in two-player zero-sum games, where the goal is to find the optimal move for a player, assuming that the opponent is also playing optimally. Here's how the algorithm works:

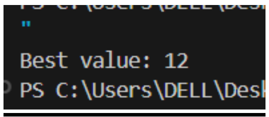
1. **Game Tree Representation:**
   - The game is represented as a tree where each level represents a player's turn, and each node represents a possible game state after a player's move.
   - The tree starts from the current game state and expands further based on all possible moves that can be made by both players.
2. **Maximizing and Minimizing Players:**
   - In a two-player game, one player tries to maximize their score (Maximizing Player), while the other player tries to minimize the score (Minimizing Player).
   - The Maximizing Player aims to maximize the utility value (score) of the game state, while the Minimizing Player aims to minimize it.
3. **Recursive Evaluation:**
   - The Minimax algorithm evaluates each possible move recursively by alternating between maximizing and minimizing players.
   - It explores the game tree depth-first, evaluating the utility of each leaf node (terminal game state) and propagating the values back up the tree.
4. **Terminal Nodes:**
   - When the algorithm reaches a terminal node (end of the game or maximum depth), it evaluates the utility of that node directly.
   - The utility value is determined based on the game's rules and can represent factors like winning, losing, or a heuristic evaluation of the game state.
5. **Backpropagation:**
   - As the algorithm backtracks up the tree, it propagates the utility values upward, updating the values of parent nodes based on the values of their children.
   - Maximizing players select the maximum value among their children, while minimizing players select the minimum value.
6. **Best Move Selection:**
   - Once the entire tree is evaluated, the algorithm selects the move that leads to the highest utility value if it's the Maximizing Player's turn or the lowest utility value if it's the Minimizing Player's turn.
   - This selected move represents the optimal decision for the current game state under the assumption that both players will play optimally,

The Minimax algorithm systematically explores the game tree, considering all possible moves and their consequences, to determine the best possible move for a player. By recursively evaluating each game state, it provides a strategy for decision-making in two-player zero-sum games.

Code:

```
def max_min(depth, node_index, maximizing_player, values, alpha, beta):
    if depth == 0:
        return values[node_index]

    if maximizing_player:
        best_value = -float("inf")
        for i in range(0, 2):
            child_value = max_min(depth - 1, 2 * node_index + i, False, values, alpha, beta)
            best_value = max(best_value, child_value)
            alpha = max(alpha, best_value)
            if beta <= alpha:
                break
        return best_value
    else:
        best_value = float("inf")
        for i in range(0, 2):
            child_value = max_min(depth - 1, 2 * node_index + i, True, values, alpha, beta)
            best_value = min(best_value, child_value)
            beta = min(beta, best_value)
            if beta <= alpha:
                break
        return best_value


values = [6, 5, 11, -2, -1, 12, 0, 18]
best_value = max_min(3, 0, True, values, -float("inf"), float("inf"))
print("Best value:", best_value)
```

Output:



```
PS C:\Users\DELL\Des
"
Best value: 12
PS C:\Users\DELL\Des
```

Conclusion: Hence the min max algorithm is implemented successfully.

EXPERIMENT 5.2: ALPHA BETA PRUNING

Aim: Write a program to implement alpha beta pruning algorithm.

Language Used: Python

Theory:

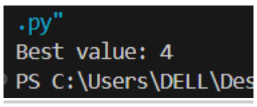
Alpha-beta pruning is an optimization technique used in the Minimax algorithm to reduce the number of nodes that need to be evaluated in the search tree. It prunes branches of the tree that do not need to be explored further, thus reducing the computational effort required to find the optimal move.

1. **Minimax Algorithm:** Alpha-beta pruning is used in conjunction with the Minimax algorithm, which is a decision-making algorithm commonly used in two-player zero-sum games (such as chess or tic-tac-toe).
2. **Search Tree:** In Minimax, a search tree is constructed where each level alternates between representing the moves of the maximizing player and the minimizing player. The leaves of the tree represent terminal game states, and each node represents a possible state of the game.
3. **Alpha and Beta Values:** During the search, two values, alpha and beta, are maintained. Alpha represents the maximum score that the maximizing player is assured of, while beta represents the minimum score that the minimizing player is assured of.
4. **Pruning Condition:** At each level of the search tree, if it is found that the current player has a choice that leads to a value worse (i.e., lower for the maximizing player, higher for the minimizing player) than a previously explored choice, then further exploration of that branch is unnecessary. This is the condition under which pruning occurs.
5. **Efficiency:** By eliminating branches that cannot possibly lead to a better solution than one already found, alpha-beta pruning reduces the number of nodes that need to be evaluated, resulting in significantly faster computation, particularly in large search trees.

Code:

```
def alphabeta(node, depth, alpha, beta, maximizingPlayer, values):
    if depth == 0 or node >= len(values):
        return values[node]

    if maximizingPlayer:
        maxEval = -float('inf')
        for child in [2 * node + i for i in range(2)]:
            eval = alphabeta(child, depth - 1, alpha, beta, False, values)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return maxEval
    else:
        minEval = float('inf')
        for child in [2 * node + i for i in range(2)]:
            eval = alphabeta(child, depth - 1, alpha, beta, True, values)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return minEval

# Example usage:
values = [15, 2, 1, -8, -9, 16, 0, 4]
best_value = alphabeta(0, 3, -float('inf'), float('inf'), True, values)
print("Best value:", best_value)
```

Output:



```
.py"
Best value: 4
PS C:\Users\DELL\Des
```

Conclusion: Hence, the alpha beta pruning is implemented successfully.

EXPERIMENT 6: GREEDY AND DYNAMIC APPROACH

EXPERIMENT 6.1: FRACTIONAL KNAPSACK USING GREEDY APPROACH

Aim: Write a program to implement fractional knapsack using greedy approach.

Language Used: Python

Theory:

The fractional knapsack problem is a classic optimization problem in combinatorial optimization and is a variant of the knapsack problem. In this problem, we are given a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity. The objective is to fill the knapsack with a combination of items that maximizes the total value of the items while not exceeding the weight capacity of the knapsack.

The greedy approach is a strategy for solving optimization problems that makes the locally optimal choice at each step with the hope of finding a global optimum. In the case of the fractional knapsack problem, the greedy approach involves selecting items based on their value-to-weight ratio, i.e., the ratio of the value of an item to its weight.

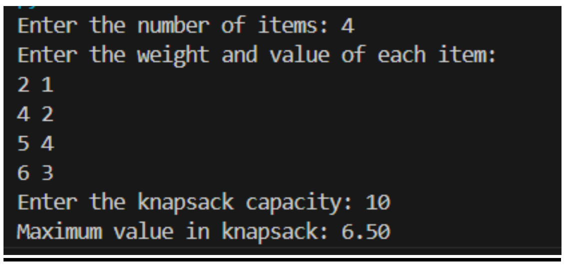
Algorithm:

1. **Compute Value-to-Weight Ratios:** Calculate the value-to-weight ratio for each item by dividing the value of the item by its weight.
2. **Sort Items:** Sort the items in non-increasing order of their value-to-weight ratios. This step ensures that items with higher value-to-weight ratios are considered first.
3. **Fill the Knapsack:** Iterate over the sorted items and add items to the knapsack until either the knapsack is full or there are no more items left. If adding a whole item exceeds the remaining capacity of the knapsack, add a fraction of that item to the knapsack such that the remaining capacity is fully utilized.
4. **Compute Total Value:** Compute the total value of the items in the knapsack, including any fractions of items.

Code:

```
class Item:
    def __init__(self, weight, value):
        self.weight = weight
        self.value = value
        self.ratio = value / weight

def swap(a, b):
    temp = a
    a = b
    b = temp

def partition(items, low, high):
    pivot = items[high].ratio
    i = low - 1
    for j in range(low, high):
        if items[j].ratio > pivot:
            i += 1
            items[i], items[j] = items[j], items[i]

    items[i + 1], items[high] = items[high], items[i + 1]
    return i + 1

def quickSort(items, low, high):
    if low < high:
        pi = partition(items, low, high)
        quickSort(items, low, pi - 1)
        quickSort(items, pi + 1, high)

def fractionalKnapsack(items, n, capacity):
    quickSort(items, 0, n - 1)

    currentWeight = 0
    totalValue = 0.0

    for i in range(n):
        if currentWeight + items[i].weight <= capacity:
            currentWeight += items[i].weight
            totalValue += items[i].value
        else:
            remainingCapacity = capacity - currentWeight
            totalValue += (remainingCapacity / items[i].weight) * items[i].value
            break

    print("Maximum value in knapsack:", "{:.2f}".format(totalValue))

if __name__ == "__main__":
    n = int(input("Enter the number of items: "))

    items = []
    print("Enter the weight and value of each item:")
    for _ in range(n):
        weight, value = map(int, input().split())
        items.append(Item(weight, value))

    capacity = int(input("Enter the knapsack capacity: "))

    fractionalKnapsack(items, n, capacity)
```

<u>Output:</u>



```
Enter the number of items: 4
Enter the weight and value of each item:
2 1
4 2
5 4
6 3
Enter the knapsack capacity: 10
Maximum value in knapsack: 6.50
```

Conclusion: Hence, the fractional knapsack using greedy approach is successfully implemented.

EXPERIMENT 6.2: 0/1 KNAPSACK USING DYNAMIC APPROACH

Aim: Write a program to implement 0/1 knapsack using dynamic approach.

Language Used: Python

Theory:

Dynamic programming is a powerful technique commonly used to solve optimization problems, including the 0/1 knapsack problem. The 0/1 knapsack problem involves selecting items to maximize the total value while ensuring that the total weight of the selected items does not exceed a given capacity. Here's the theory behind solving the 0/1 knapsack problem using dynamic programming:

Given a set of items, each with a weight wiwi and a value vivi, and a knapsack of capacity WW, the goal is to select items to maximize the total value while ensuring that the total weight of the selected items does not exceed the capacity of the knapsack. Each item can either be selected (0/1 choice) or not selected.

Dynamic programming breaks down the problem into smaller subproblems and solves each subproblem only once, storing the solutions in a table to avoid redundant calculations. The key idea is to exploit the principle of optimality, which states that an optimal solution to a problem contains optimal solutions to its subproblems.

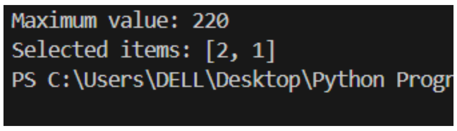
Steps:

1. **Define Subproblems:** Define subproblems that represent different combinations of items and capacities. For the 0/1 knapsack problem, the subproblem can be defined as finding the maximum value that can be obtained using the first ii items and a knapsack capacity of jj.
2. **Formulate Recurrence:** Formulate a recurrence relation that expresses the solution to each subproblem in terms of solutions to smaller subproblems. In the case of the 0/1 knapsack problem, the recurrence relation typically involves considering whether to include the iith item in the knapsack or not.
3. **Base Case:** Define the base case of the recurrence, which represents the smallest subproblems that can be solved directly. For the 0/1 knapsack problem, the base case usually involves considering either the first item only or a knapsack with zero capacity.
4. **Fill the Table Bottom-Up:** Use dynamic programming to fill the table of solutions in a bottom-up manner, starting from the base case and iteratively computing solutions for larger subproblems based on solutions to smaller subproblems.
5. **Extract Solution:** Once the table is filled, extract the solution to the original problem from the table. This typically involves tracing back through the table to identify which items were selected to achieve the optimal solution.

Code:

```
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - weights[i - 1]])
            else:
                dp[i][w] = dp[i - 1][w]

    selected_items = []
    w = capacity
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i - 1][w]:
            selected_items.append(i - 1)
            w -= weights[i - 1]

    return dp[n][capacity], selected_items

weights = [10, 20, 30]
values = [60, 100, 120]
capacity = 50
max_value, selected_items = knapsack(weights, values, capacity)

print("Maximum value:", max_value)
print("Selected items:", selected_items)
```

Output:



```
Maximum value: 220
Selected items: [2, 1]
PS C:\Users\DELL\Desktop\Python Progr
```

Conclusion: Hence, 0/1 Knapsack using dynamic approach is implemented successfully.

EXPERIMENT 7: TIC TAC TOE GAME USING MIN MAX ALGORITHM

Aim: Write a program to implement tic tac toe game using min max algorithm.

Language Used: Python

Theory:

The minimax algorithm is a decision rule for minimizing the possible loss for a worst-case scenario. It's widely used in turn-based two-player games like Tic Tac Toe.

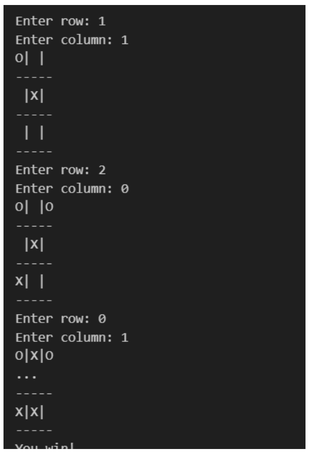
Steps:

1. **Represent the game state**: You need to represent the current state of the Tic Tac Toe board. This could be done using a 3x3 grid where each cell can be empty, X, or O.
2. **Define terminal states**: Determine what conditions constitute a terminal state (win, lose, or draw). In Tic Tac Toe, this means either a player has won, it's a draw, or there are still moves left.
3. **Define the utility function**: Define a function that evaluates the utility of a given state for a player. For Tic Tac Toe, you could assign a value of +1 for a win, -1 for a loss, and 0 for a draw.
4. **Implement the minimax algorithm**: Recursively search the game tree to determine the best move. The algorithm consists of two main functions: minimize and maximize.
    - maximize function tries to find the move with the maximum utility for the player.
    - minimize function tries to find the move with the minimum utility for the opponent.
5. **Make the best move**: Use the minimax algorithm to determine the best move for the current player.

Code:

```
import math

class Board:
    def __init__(self):
        self.board = [[' ' for _ in range(3)] for _ in range(3)]

    def print_board(self):
        for row in self.board:
            print("|".join(row))
            print("-" * 5)


    def game_over(self):
        for row in self.board:
            if row.count('X') == 3 or row.count('O') == 3:
                return True
        for col in range(3):
            if self.board[0][col] == self.board[1][col] == self.board[2][col] and self.board[0][col] != ' ':
                return True
        if self.board[0][0] == self.board[1][1] == self.board[2][2] and self.board[0][0] != ' ':
            return True
        if self.board[0][2] == self.board[1][1] == self.board[2][0] and self.board[0][2] != ' ':
            return True
        return False

    # Gets empty cells on the board
    def get_empty_cells(self):
        return [(i, j) for i in range(3) for j in range(3) if self.board[i][j] == ' ']

    # Makes a move on the board
    def make_move(self, row, col, player):
        self.board[row][col] = player

    # Undo the move on the board
    def undo_move(self, row, col):
        self.board[row][col] = ' '

def minimax(board, depth, is_maximizing_player, alpha, beta):
    if board.game_over() or depth == 0:
        if board.game_over():
            if is_maximizing_player:
                return -1
            else:
                return 1
        else:
            return 0

    if is_maximizing_player:
        max_eval = -math.inf
        for row, col in board.get_empty_cells():
            board.make_move(row, col, 'O')
            eval = minimax(board, depth - 1, False, alpha, beta)
            board.undo_move(row, col)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = math.inf
        for row, col in board.get_empty_cells():
            board.make_move(row, col, 'X')
            eval = minimax(board, depth - 1, True, alpha, beta)
            board.undo_move(row, col)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval

def find_best_move(board):
    best_move = (-1, -1)
    best_eval = -math.inf
    for row, col in board.get_empty_cells():
        board.make_move(row, col, 'O')
        eval = minimax(board, 5, False, -math.inf, math.inf)  # Depth is set to 5
        board.undo_move(row, col)
        if eval > best_eval:
            best_eval = eval
            best_move = (row, col)
    return best_move

# Main function to play Tic-Tac-Toe
def play_tic_tac_toe():
    board = Board()
    while not board.game_over():
        board.print_board()
        row = int(input("Enter row: "))
        col = int(input("Enter column: "))
        board.make_move(row, col, 'X')
        if board.game_over():
            break
        best_move = find_best_move(board)
        board.make_move(best_move[0], best_move[1], 'O')
    board.print_board()
    if 'X' in board.board[0]:
        print("You win!")
    elif 'O' in board.board[0]:
        print("You lose!")
    else:
        print("It's a draw!")


play_tic_tac_toe()
```

Output:



```
Enter row: 1
Enter column: 1
O| |
-----
 |X|
-----
 | |
-----
Enter row: 2
Enter column: 0
O| |O
-----
 |X|
-----
X| |
-----
Enter row: 0
Enter column: 1
O|X|O
...
-----
X|X|
-----
You win!
```

<u>Conclusion:</u> Hence, tic tac toe game is implemented using min max algorithm successfully.

EXPERIMENT 8: NATURAL LANGUAGE PROCESSING

EXPERIMENT 8.1: TOKENIZATION, LEMMATIZATION, STEMMING AND REMOVAL OF STOP WORDS

Aim: Write a program to implement 0/1 knapsack using dynamic approach.

Language Used: Python

Theory:

Natural Language Processing (NLP is a field of artificial intelligence that focuses on enabling computers to understand, interpret, and generate human language in a way that is both meaningful and useful. It involves the intersection of computer science, linguistics, and artificial intelligence.

Tokenization**,** Stop Words Removal**,** Stemming**,** and Lemmatization are all fundamental techniques used in NLP for text preprocessing. These techniques help in preparing text data for further analysis or processing by reducing noise, normalizing text, and extracting essential information.

Tokenization is crucial in NLP because it breaks down a text into smaller units (tokens), making it easier to process and analyze the text at a granular level. This step is foundational for various NLP tasks such as text classification, named entity recognition, and machine translation.

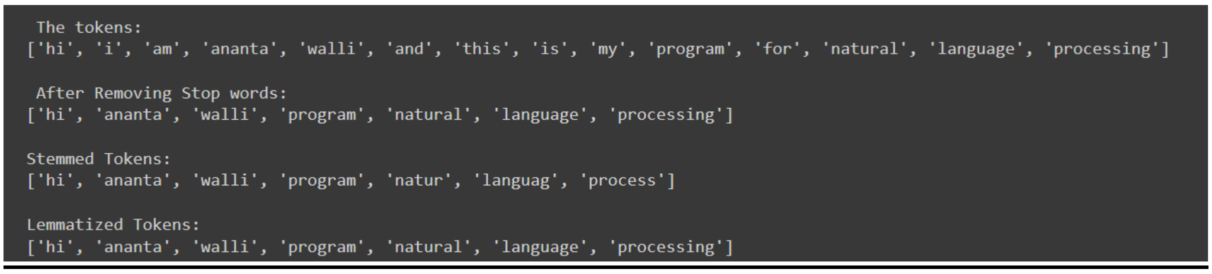
Stop Words Removal is important in NLP because stop words often do not contribute significant semantic meaning to the text and can add noise to the analysis. By removing them, we can focus on the more relevant content of the text, which is essential for tasks like sentiment analysis or topic modeling.

Stemming and **Lemmatization** are important for normalizing text by reducing inflected words to their base or root forms. This helps in reducing the dimensionality of the feature space and improving the generalization of NLP models. While stemming is a more heuristic approach that simply chops off prefixes or suffixes, lemmatization considers the context and meaning of words, providing more accurate results.

Code:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer, WordNetLemmatizer

nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')

def preprocess_text(text):

    tokens = word_tokenize(text.lower())
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word not in stop_words]


    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(word) for word in filtered_tokens]


    lemmatizer = WordNetLemmatizer()
    lemmatized_tokens = [lemmatizer.lemmatize(word) for word in filtered_tokens]

    return tokens, filtered_tokens, stemmed_tokens, lemmatized_tokens

text = "Hi I am Ananta Walli and this is my program for Natural Language Processing"
tokens, filtered_tokens, stemmed_tokens, lemmatized_tokens = preprocess_text(text)

print(" The tokens:")
print(tokens)
print("\n After Removing Stop words:")
print(filtered_tokens)
print("\nStemmed Tokens:")
print(stemmed_tokens)
print("\nLemmatized Tokens:")
print(lemmatized_tokens)
```

Output:



```
 The tokens:
['hi', 'i', 'am', 'ananta', 'walli', 'and', 'this', 'is', 'my', 'program', 'for', 'natural', 'language', 'processing']

 After Removing Stop words:
['hi', 'ananta', 'walli', 'program', 'natural', 'language', 'processing']

Stemmed Tokens:
['hi', 'ananta', 'walli', 'program', 'natur', 'languag', 'process']

Lemmatized Tokens:
['hi', 'ananta', 'walli', 'program', 'natural', 'language', 'processing']
```

Conclusion: Hence, the program of Tokenization, Lemmatization, Stemming and Removal Of Stop words are successfully implemented.

EXPERIMENT 8.2: BAG OF WORDS ALGORITHM

Aim: Write a program to implement 0/1 knapsack using dynamic approach.

Language Used: Python

Theory:

Bag of Words is a widely used technique in Natural Language Processing (NLP) for text representation. It's a simple and effective way to convert text data into numerical vectors that can be used as input for machine learning algorithms. The fundamental idea behind BoW is to represent text as a "bag" of its constituent words, disregarding grammar and word order, while focusing only on word frequencies.

Key Concepts:

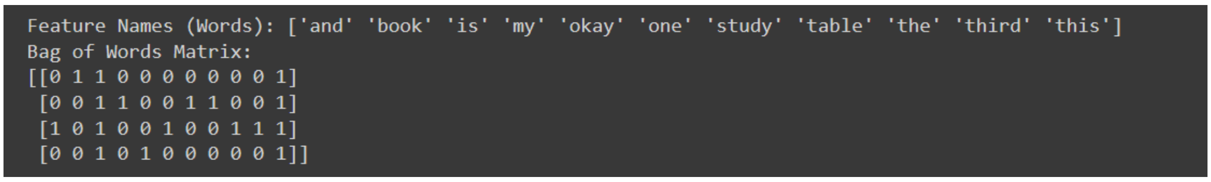
1. **Corpus**: A corpus is a collection of text documents. It can contain various documents such as articles, emails, tweets, etc. In BoW, the corpus serves as the dataset from which we extract features.
2. **Tokenization**: Tokenization is the process of breaking down a text into individual words or tokens. This step is essential in BoW because it enables us to create a vocabulary of unique words present in the corpus.
3. **Vocabulary**: The vocabulary is a set of all unique words present in the corpus. Each word in the vocabulary serves as a feature.
4. **Document-Term Matrix (DTM)**: The Document-Term Matrix (DTM) is the main output of the BoW technique. It's a matrix where each row represents a document from the corpus, and each column represents a word from the vocabulary. The cells of the matrix contain the frequency of each word in the corresponding document.
5. **Vectorization**: Vectorization is the process of converting text data into numerical vectors. In BoW, each document is represented as a vector, typically by using the word frequencies as components of the vector.
6. **Feature Extraction**: BoW is a feature extraction technique. It transforms unstructured text data into structured numerical features that can be used as input.
7. **Usage**: BoW is a versatile technique used in various NLP tasks such as text classification, sentiment analysis, information retrieval, and document clustering. It provides a simple yet effective way to represent text data numerically, making it suitable for machine learning algorithms. However, it does not capture semantic meaning or word relationships, which can be a limitation in certain contexts.

Code:

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = [
    "This is a book",
    "This is my study table.",
    "And this is the third one.",
    "Is this okay?"
]

vectorizer = CountVectorizer()
```

```
X = vectorizer.fit_transform(corpus)

feature_names = vectorizer.get_feature_names_out()

print("Feature Names (Words):", feature_names)

# Bag of Words representation
print("Bag of Words Matrix:")
print(X.toarray())
```

Output:



```
Feature Names (Words): ['and' 'book' 'is' 'my' 'okay' 'one' 'study' 'table' 'the' 'third' 'this']
Bag of Words Matrix:
[[0 1 1 0 0 0 0 0 0 0 1]
 [0 0 1 1 0 0 1 1 0 0 1]
 [1 0 1 0 0 1 0 0 1 1 1]
 [0 0 1 0 1 0 0 0 0 0 1]]
```

Conclusion: Hence, the code for bag of words is implemented successfully.

EXPERIMENT 9: XOR GATE

Aim: Write a program to implement XOR Gate.

Language Used: Python

Theory:

The XOR gate is a fundamental logic gate that outputs true (1) only when the number of true inputs is odd. It can be implemented using a variety of techniques, including Boolean algebra, truth tables, or programmatically.

The truth table for an XOR gate is as follows:

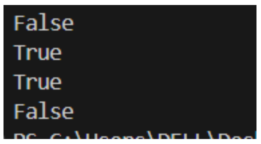
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

From the truth table, you can see that the output is 1 if and only if one of the inputs is 1.

Code:

```
def xor_gate(a, b):
    return (a and not b) or (not a and b)

# cases
print(xor_gate(False, False))
print(xor_gate(False, True))
print(xor_gate(True, False))
print(xor_gate(True, True))
```

Output:



```
False
True
True
False
```

Conclusion: Hence, the xor gate is successfully implemented.

EXPERIMENT 10: OPEN ENDED EXPERIMENT

Aim: To compare the performance of Logistic Regression, Random Forest, and Support Vector Machine classifiers on the Iris Classification Model by evaluating their accuracies.

Language Used: Python

Theory:

Classification is a type of supervised learning where the goal is to predict the categorical class labels of new instances based on past observations. Classification models learn from labeled training data and then predict the class labels of unseen data.

Key Concepts:

1. **Classes**:
   - Classes represent the categories or groups into which instances are classified. For example, in the Iris dataset, the classes are different species of Iris flowers: Setosa, Versicolor, and Virginica.
2. **Features**:
   - Features are the measurable properties or attributes of the data used to make predictions. In the Iris dataset, features include sepal length, sepal width, petal length, and petal width.
3. **Training Data**:
   - Training data consists of labeled instances used to train the classification model. Each instance is associated with a class label. In the case of Iris dataset, training data includes measurements of various Iris flowers along with their species labels.
4. **Model Training**:
   - During training, a classification model learns patterns and relationships between features and class labels from the training data. The model adjusts its parameters to minimize the prediction error.
5. **Model Evaluation**:
   - After training, the model's performance is evaluated on a separate set of data called the test set. Evaluation metrics such as accuracy, precision, recall, and F1-score are commonly used to assess the model's performance.
6. **Prediction**:
   - Once trained and evaluated, the classification model can be used to predict the class labels of new, unseen instances based on their features.

The Iris dataset is a well-known dataset in the field of machine learning and is often used for practicing classification algorithms. It contains measurements of iris flowers (sepal length, sepal width, petal length, and petal width) along with their species labels (Setosa, Versicolor, and Virginica)

Code:

```
import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Load the Iris dataset
iris_data = pd.read_csv("IRIS.csv")
X = iris_data.drop('species', axis=1)
y = iris_data['species']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize and train Logistic Regression
logistic_reg = LogisticRegression()
logistic_reg.fit(X_train_scaled, y_train)
logistic_reg_train_pred = logistic_reg.predict(X_train_scaled)
logistic_reg_test_pred = logistic_reg.predict(X_test_scaled)
logistic_reg_train_accuracy = accuracy_score(y_train, logistic_reg_train_pred)
logistic_reg_test_accuracy = accuracy_score(y_test, logistic_reg_test_pred)

# Display results for Logistic Regression
print("Logistic Regression Train Accuracy:", logistic_reg_train_accuracy)
print("Logistic Regression Test Accuracy:", logistic_reg_test_accuracy)
print("\nLogistic Regression Classification Report:")
print(classification_report(y_test, logistic_reg_test_pred))
print("\nLogistic Regression Confusion Matrix:")
print(confusion_matrix(y_test, logistic_reg_test_pred))

# Initialize and train Random Forest
random_forest = RandomForestClassifier(n_estimators=100, random_state=42)
random_forest.fit(X_train_scaled, y_train)
random_forest_train_pred = random_forest.predict(X_train_scaled)
random_forest_test_pred = random_forest.predict(X_test_scaled)
random_forest_train_accuracy = accuracy_score(y_train, random_forest_train_pred)
random_forest_test_accuracy = accuracy_score(y_test, random_forest_test_pred)

# Display results for Random Forest
print("\nRandom Forest Train Accuracy:", random_forest_train_accuracy)
print("Random Forest Test Accuracy:", random_forest_test_accuracy)
print("\nRandom Forest Classification Report:")
print(classification_report(y_test, random_forest_test_pred))
print("\nRandom Forest Confusion Matrix:")
print(confusion_matrix(y_test, random_forest_test_pred))

# Initialize and train Support Vector Machine (SVM)
svm_classifier = SVC(kernel='linear')
svm_classifier.fit(X_train_scaled, y_train)
svm_train_pred = svm_classifier.predict(X_train_scaled)
svm_test_pred = svm_classifier.predict(X_test_scaled)
svm_train_accuracy = accuracy_score(y_train, svm_train_pred)
svm_test_accuracy = accuracy_score(y_test, svm_test_pred)

# Display results for SVM
```

print("\nSVM Train Accuracy:", svm_train_accuracy)

print("SVM Test Accuracy:", svm_test_accuracy)

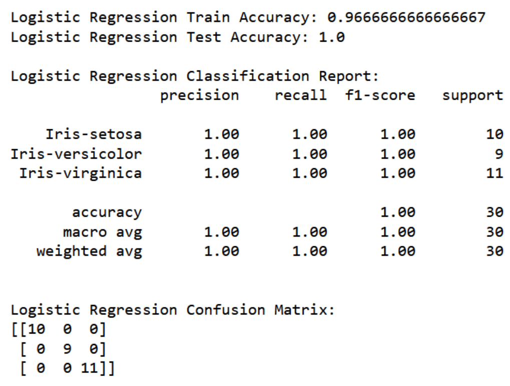
print("\nSVM Classification Report:")

print(classification_report(y_test, svm_test_pred))

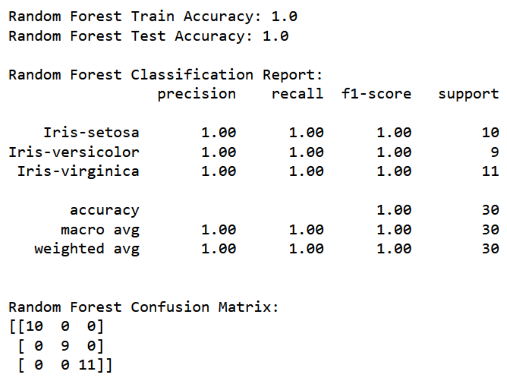
print("\nSVM Confusion Matrix:")

print(confusion_matrix(y_test, svm_test_pred))

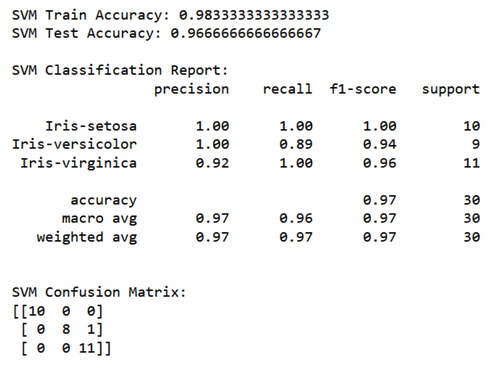
Output:



```
Logistic Regression Train Accuracy: 0.9666666666666667
Logistic Regression Test Accuracy: 1.0

Logistic Regression Classification Report:
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        10
Iris-versicolor       1.00      1.00      1.00         9
 Iris-virginica       1.00      1.00      1.00        11

       accuracy                           1.00        30
      macro avg       1.00      1.00      1.00        30
   weighted avg       1.00      1.00      1.00        30


Logistic Regression Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```



```
Random Forest Train Accuracy: 1.0
Random Forest Test Accuracy: 1.0

Random Forest Classification Report:
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        10
Iris-versicolor       1.00      1.00      1.00         9
 Iris-virginica       1.00      1.00      1.00        11

       accuracy                           1.00        30
      macro avg       1.00      1.00      1.00        30
   weighted avg       1.00      1.00      1.00        30


Random Forest Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```



```
SVM Train Accuracy: 0.9833333333333333
SVM Test Accuracy: 0.9666666666666667

SVM Classification Report:
                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        10
Iris-versicolor       1.00      0.89      0.94         9
 Iris-virginica       0.92      1.00      0.96        11

       accuracy                           0.97        30
      macro avg       0.97      0.96      0.97        30
   weighted avg       0.97      0.97      0.97        30


SVM Confusion Matrix:
[[10  0  0]
 [ 0  8  1]
 [ 0  0 11]]
```

Conclusion: Hence, the Iris Classification Model is implemented successfully.