1) TOY PROBLEM: (TOWER OF HANOI):

```python
def TowerOfHanoi(n , source, destination, auxiliary):
    if n==1:
        print ("Move disk 1 from source",source,"to destination",destination)
        return
    TowerOfHanoi(n-1, source, auxiliary, destination)
    print ("Move disk",n,"from source",source,"to destination",destination)
    TowerOfHanoi(n-1, auxiliary, destination, source)

# Driver code
n = 4
TowerOfHanoi(n,'A','B','C')
```

```
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from source A to destination C
Move disk 4 from source A to destination B
Move disk 1 from source C to destination B
Move disk 2 from source C to destination A
Move disk 1 from source B to destination A
Move disk 3 from source C to destination B
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
```

2) REAL WORLD AGENT PROBLEM

In [8]:

```python
'Agent problem Table method'
door_status=int(input("Enter the status of door(0/1):"))
person=int(input("Enter if there is person standing(0/1):"))
if door_status==1 and person==1:
    print(" stays openes")
elif door_status==1 and person==0:
    print("Close")
elif door_status==0 and person==1:
    print("Door openes")
elif door_status==0 and person==0:
    print("Stays closed ")
```

```
Enter the status of door(0/1):1
Enter if there is person standing(0/1):0
Close
```

```python
colors = ['red','blue','green','orange','yellow','violet']

states = ['MP','New Delhi','Haryana','Rajasthan','Gujarat']

neighbours = {
    'MP':['New Delhi','Rajasthan','Gujarat'],
    'New Delhi':['MP','Rajasthan','Haryana'],
    'Haryana':['New Delhi'],
    'Rajasthan':['MP','Gujarat','New Delhi'],
    'Gujarat':['Rajasthan','MP']
}

state_colors = {}
def promising(state, color):
    for neighbour in neighbours.get(state):
        color_of_neighbor = state_colors.get(neighbour)
        if color_of_neighbor == color:
            return False

    return True

for state in states:
    for color in colors:
        if promising(state, color):
            state_colors[state] = color

print (state_colors)
```

```
{'MP': 'violet', 'New Delhi': 'yellow', 'Haryana': 'violet', 'Rajasthan': 'oran
ge', 'Gujarat': 'yellow'}
```

3) CONSTRAINT SATISFACTION PROBLEM:

```
import itertools
def solve():
    letter=('b','a','s','e','l','g','m')
    digit=range(10)
    for perm in itertools.permutations(digit,len(letter)):
        sol=dict(zip(letter,perm))
        if sol['b']==0 or sol['g']==0:
            continue
        base=1000*sol['b']+100*sol['a']+10*sol['s']+sol['e']
        ball=1000*sol['b']+100*sol['a']+10*sol['l']+sol['l']
        games=10000*sol['g']+1000*sol['a']+100*sol['m']+10*sol['e']+sol['s']
        if base+ball==games:
            print("base","ball","games")
            return base,ball,games
print(solve())
```

```
base ball games
(7483, 7455, 14938)
```

4) DFS AND BFS

```
In [2]: graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F', 'G'],
    'F': [],
    'G': ['F']
}

visited_bfs = []
queue = []

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

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

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

visited_dfs = set()

def dfs(visited, graph, node):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)

        for neighbor in graph[node]:
            dfs(visited, graph, neighbor)

print("BFS:", end=" ")
bfs(visited_bfs, graph, 'A')
print("\nDFS:", end=" ")
dfs(visited_dfs, graph, 'A')
```

```
BFS: A B C D E F G
DFS: A B D E F G C
```

5) BEST FIRST SEARCH AND A*:

```
In [16]: from queue import PriorityQueue
class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list
    def get_neighbors(self, v):
        return self.adjacency_list[v]
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }
        return H[n]
    def best_first_search(self, start, goal):
        explored = []
        pq = PriorityQueue()
        pq.put((0, start))
        parents = {start: None}
        while not pq.empty():
            current = pq.get()[1]            if current == goal:
                path = []
                while current is not None:
                    path.append(current)
                    current = parents[current]
                path.reverse()
                print(f"Best-First Search path: {path}")
                return path
            explored.append(current)
            for neighbor, weight in self.get_neighbors(current):
                if neighbor not in explored and neighbor not in [i[1] for i in pq.queue]:
                    parents[neighbor] = current
                    pq.put((self.h(neighbor), neighbor))
        print("Path not found!")
        return None
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.best_first_search('A', 'D')
```

```
Best-First Search path: ['A', 'D']

Out[16]: ['A', 'D']
```

```
'A*'
from queue import PriorityQueue
def best(source,target,n,graph):
    visted=[0]*n
    visted[source]=True
    pq=PriorityQueue()
    pq.put((0,source))
    while pq.empty()==False:
        u=pq.get()[1]
        print(u,end=" ")
        if u==target:
            break
        for v,c in graph[u]:
            if visted[v]==False:
                visted[v]=True
                pq.put((c,v))
    print()
graph={
    0:[(1,5),(2,3)],
    1:[(3,2)],
    2:[(4,1)],
    3:[(4,6)],
    4:[]
}
source=0
target=4
n=5
best(source,target,n,graph)
```

```
0 2 4
```

6) UNCERTAIN METHOD(FUZZY METHOD):

```
# Difference Between Two Fuzzy Sets for A_key in A:  X[A_key]= 1-A[A_key] print('Fuzzy Set Complement is :', X)
A = dict()
B = dict()
Y = dict()
X = dict()
A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}
print('The First Fuzzy Set is :', A)
print('The Second Fuzzy Set is :', B)
for A_key, B_key in zip(A, B):
    A_value = A[A_key]
    B_value = B[B_key]
    if A_value > B_value:
        Y[A_key] = A_value
    else:
            Y[B_key] = B_value

print('Fuzzy Set Union is :', Y)
```

```
The First Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
The Second Fuzzy Set is : {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}
Fuzzy Set Union is : {'a': 0.9, 'b': 0.9, 'c': 0.6, 'd': 0.6}
```

7) UNCERTAIN METHOD (MONTY HALL):

In [12]:
```
# Monty Hall Game in Python
import random

def play_monty_hall(choice):
    prizes = ['goat', 'car', 'goat']
    random.shuffle(prizes)
    while True:
        opening_door = random.randrange(len(prizes))
        if prizes[opening_door] != 'car' and choice-1 != opening_door:
            break
    opening_door = opening_door + 1
    print('We are opening the door number-%d' % (opening_door))
    options = [1,2,3]
    options.remove(choice)
    options.remove(opening_door)
    switching_door = options[0]
    print('Now, do you want to switch to door number-%d? (yes/no)' %(switching_door))
    answer = input()
    if answer == 'yes':
        result = switching_door - 1
    else:
        result = choice - 1
    print('And your prize is ....', prizes[result].upper())
choice = int(input('Which door do you want to choose? (1,2,3): '))
play_monty_hall(choice)
```

```
Which door do you want to choose? (1,2,3): 2
We are opening the door number-1
Now, do you want to switch to door number-3? (yes/no)
y
And your prize is .... CAR
```

8) LEARNING ALGORITHM:

9) NLP PROGRAM: