1.a

```
graph1={
      'A': set(['B','C']),
        'B':set(['A','B','E']),
        'C': set(['A','F']),
        'D':set(['B']),
        'E':set(['B','F']),
        'F':set(['C','E']),
    }
def dfs(graph,node,visited):
    if node not in visited:
        visited.append(node)
        for n in graph[node]:
            dfs(graph,n,visited)
    return visited
visited=dfs(graph1,'A',[])
print(visited)
```

1.b

```
graph={'A':set(['B','C']),
       'B':set(['A','D','E']),
       'C':set(['A','F']),
       'D':set(['B']),
       'E':set(['B','F']),
       'F':set(['C','E']),
       }
def bfs(start):
    queue=[start]
    levels={}
    levels[start]=0
    visited = set(start)
    while queue:
        node=queue.pop(0)
        neighbours=graph[node]
        for neighbor in neighbours:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
                levels[neighbor]=levels[node]+1
    print(levels)
    return visited
print(str(bfs('A')))
```

2.a

```
global N
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=' ')
        print()

def isQSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solveNQUtil(board, col):
    if col >= N:
        return True
    for i in range(N):
        if isQSafe(board, i, col):
            board[i][col] = 1
            if solveNQUtil(board, col + 1):
                return True
            board[i][col] = 0
    return False


def solveNQ():
    board = [[0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0]]
    if not solveNQUtil(board, 0):
        print("Solution does not exist")
        return False
    printSolution(board)
    return True


solveNQ()
```

2.b

```
def t(h,s,aux,e):
    if h>=1:
        print("line1")
        t(h-1,s,e,aux)
        print("line2")
        print("moving disk from",h,s,"to",e)
        print("line3")
        t(h-1,aux,s,e)

t(3,"A","B","C")
```

3.a

3.b

```
j1 = int(input("Capacity of jug1: "))
j2 = int(input("Capacity of jug2: "))
q = int(input("Amount of water to be measured: "))

def apply_rule(ch, x, y):
    if ch == 1:  # Fill jug1
        if x < j1:
            return j1, y
        else:
            print("Rule cannot be applied")
            return x, y
    elif ch == 2:  # Fill jug2
        if y < j2:
            return x, j2
        else:
            print("Rule cannot be applied")
            return x, y
    elif ch == 3:  # Transfer all water from jug1 to jug2
        if x > 0 and x + y <= j2:
            return 0, x + y
        else:
            print("Rule cannot be applied")
            return x, y
    elif ch == 4:  # Transfer all water from jug2 to jug1
        if y > 0 and x + y <= j1:
            return x + y, 0
        else:
            print("Rule cannot be applied")
            return x, y
    elif ch == 5:  # Transfer water from jug1 to jug2 until jug2 is full
        if x > 0 and x + y > j2:
            return x - (j2 - y), j2
        else:
            print("Rule cannot be applied")
            return x, y
    elif ch == 6:  # Transfer water from jug2 to jug1 until jug1 is full
        if y > 0 and x + y > j1:
            return j1, y - (j1 - x)
        else:
            print("Rule cannot be applied")
            return x, y
    elif ch == 7:  # Empty jug1
        if x > 0:
            return 0, y
        else:
            print("Rule cannot be applied")
            return x, y
    elif ch == 8:  # Empty jug2
        if y > 0:
            return x, 0
        else:
            print("Rule cannot be applied")
            return x, y
    else:
        print("Invalid rule choice")
        return x, y


x = y = 0
while True:
    if x == q or y == q:
```

```
    print('Goal achieved!')
    break
else:
    print("\nAvailable rules:")
    print("Rule 1: Fill jug1")
    print("Rule 2: Fill jug2")
    print("Rule 3: Transfer all water from jug1 to jug2")
    print("Rule 4: Transfer all water from jug2 to jug1")
    print("Rule 5: Transfer water from jug1 to jug2 until jug2 is full")
    print("Rule 6: Transfer water from jug2 to jug1 until jug1 is full")
    print("Rule 7: Empty jug1")
    print("Rule 8: Empty jug2")
    ch = int(input("Enter rule to apply: "))
    x, y = apply_rule(ch, x, y)
    print("Current status:", x, y)
```

5.a

```
import os

import time


board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']

player = 1


# Game status flags

win = 1

draw = -1

running = 0

stop = 1


game = running

mark = 'X'


# Function to draw the game board

def draw_board():

    print("%s | %s | %s" % (board[1], board[2], board[3]))

    print("--|---|--")

    print("%s | %s | %s" % (board[4], board[5], board[6]))

    print("--|---|--")

    print("%s | %s | %s" % (board[7], board[8], board[9]))


# Function to check if a position is empty

def check_position(x):

    return board[x] == ' '


# Function to check if a player has won

def check_win():
    global game
    # Horizontal
    if (board[1] == board[2] == board[3] != ' ' or
        board[4] == board[5] == board[6] != ' ' or
        board[7] == board[8] == board[9] != ' '):
        game = win
    # Vertical
    elif (board[1] == board[4] == board[7] != ' ' or
          board[2] == board[5] == board[8] != ' ' or
          board[3] == board[6] == board[9] != ' '):
        game = win
    # Diagonal
    elif (board[1] == board[5] == board[9] != ' ' or
          board[3] == board[5] == board[7] != ' '):
        game = win
    # Check for draw
    elif all(space != ' ' for space in board[1:]):
        game = draw
    else:
        game = running

print("Tic Tac Toe GAME")
print("Player 1 [X] --- Player 2 [O]\n")
print("Please wait...")
time.sleep(1)

while game == running:
    os.system('cls' if os.name == 'nt' else 'clear')  # Clear the screen
    draw_board()

    if player % 2 != 0:
        print("Player 1's turn")
        mark = 'X'
    else:
        print("Player 2's turn")
        mark = 'O'

    choice = int(input("Enter the position (1-9) where you want to mark: "))

    if check_position(choice):
        board[choice] = mark
        player += 1
        check_win()
    else:
        print("Position is already taken! Try again.")
        time.sleep(1)

os.system('cls' if os.name == 'nt' else 'clear')
draw_board()

if game == draw:
    print("Game draw!")
elif game == win:
    player -= 1
    if player % 2 != 0:
        print("Player 1 won!")
    else:
        print("Player 2 won!")
```

5b

```
import random
import itertools


deck = list(itertools.product(range(1, 14), ["Heart", "Spade", "club", "diamond"]))


random.shuffle(deck)


print(deck)


for i in range(5):
   print(deck[i][0], "of", deck[i][1])
```

6.a

```
def print_in_format(matrix):
    for i in range(9):
        if i%3==0 and i>0:
            print("")
        print(str(matrix[i])+"",end ="")



def count(s):
    c=0
    ideal=[1,2,3,
           4,5,6,
           7,8,0]

    for i in range(9):
        if s[i]!=0 and s[i]!=ideal[i]:
            c+=1

    return c

def move(ar,p,st):
    store_at=st.copy()
    for i in range(len(ar)):

        dup1_st=st.copy()
        tmp=dup1_st[p]
        dup1_st[p]=dup1_st[ar[i]]
        dup1_st[ar[i]]=tmp

        trh=count(dup1_st)
        store_st=dup1_st.copy()
    return store_st,trh


state=[1,2,3,
       0,5,6,
       4,7,8]


h=count(state)
level=1


print("\n-------level "+str(level)+"-------")
print_in_format(state)
print("\nheuristic value(misplaced): "+str(h))



while h>0:
    pos=int(state.index(0))
    print('pos',pos)
    level+=1


    if pos==0:
        arr=[1,3]
        state,h=move(arr,pos,state)


    elif pos==1:
        arr=[0,2,4]
        state,h=move(arr,pos,state)


    elif pos==2:
        arr=[1,5]
        state,h=move(arr,pos,state)
```

```
elif pos==3:
    arr=[0,4,6]
    state,h=move(arr,pos,state)

elif pos==4:
    arr=[1,3,5,7]
    state,h=move(arr,pos,state)

elif pos==5:
    arr=[2,4,8]
    state,h=move(arr,pos,state)

elif pos==6:
    arr=[3,7]
    state,h=move(arr,pos,state)

elif pos==7:
    arr=[4,6,8]
    state,h=move(arr,pos,state)

elif pos==8:
    arr=[5,6]
    state,h=move(arr,pos,state)

print("\n------- level "+str(level)+"-----")
print_in_format(state)
print("\n heuristic value(misplaced): "+str(h))
```

7.a

```
from simpleai.search import CspProblem, backtrack
variables=('A','B','C','D')


domains={
   'A':['Red','Green','Blue'],
   'B':['Red','Green','Blue'],
   'C':['Red','Green','Blue'],
   'D':['Red','Green','Blue'],
}

def different_colors(variables,values):
   return values[0] != values[1]


constraints=[
   (('A','B'),different_colors),
   (('A','C'),different_colors),
   (('A','D'),different_colors),
   (('B','C'),different_colors),
   (('C','D'),different_colors),
]

problem=CspProblem(variables,domains,constraints)

solution=backtrack(problem)

print("Solution:")
print(solution)
```

8.a

8.b

9.a

```
% Facts
batsman(sachin).
wicketkeeper(dhoni).
footballer(ronaldo).

% Rules
cricketer(X) :- batsman(X).
cricketer(X) :- wicketkeeper(X).

% Footballers are not cricketers
not_cricketer(X) :- footballer(X).

% To determine if someone is a cricketer
is_cricketer(X) :- cricketer(X), \+ not_cricketer(X).
```

10.a

```
female(pam).
female(liz).
female(pat).
female(ann).
male(jim).
male(bob).
male(tom).
male(peter).
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(pat,jim).
parent(bob,peter).
parent(peter,jim).
mother(X,Y):-parent(X, Y),female(X).
father(X, Y):-parent(X,Y),male(X).
sister(X,Y):-parent(Z,X),parent(Z,Y),female(X),X\==Y.
brother(X,Y):-parent(Z,X),parent(Z,Y),male(X),X\==Y.
grandparent(X,Y):-parent(X,Z),parent(Z, Y).
grandmother(X,Z):-mother(X,Y),parent(Y,Z).
grandfather(X,Z):-father(X, Y),parent(Y,Z).
wife(X,Y):-parent(X,Z),parent(Y,Z),female(X),male(Y).
uncle(X,Z):-brother(X,Y),parent(Y,Z).
```