In [1]:

```python
tree={
    1:[2,9,10],
    2:[3,4],
    3:[],
    4:[5,6,7],
    5:[8],
    6:[],
    7:[],
    8:[],
    9:[],
    10:[]
}
def breadth_first_search(tree,start):
    q=[start]
    visited=[]

    while q:
        print("before",q)
        node=q.pop(0)
        visited.append(node)
        for child in reversed(tree[node]):
            if child not in visited and child not in q:
                q.append(child)
                print("after",q)
    return visited
result=breadth_first_search(tree,1)
print(result)
```

```
before [1]
after [10]
after [10, 9]
after [10, 9, 2]
before [10, 9, 2]
before [9, 2]
before [2]
after [4]
after [4, 3]
before [4, 3]
after [3, 7]
after [3, 7, 6]
after [3, 7, 6, 5]
before [3, 7, 6, 5]
before [7, 6, 5]
before [6, 5]
before [5]
after [8]
before [8]
[1, 10, 9, 2, 4, 3, 7, 6, 5, 8]
```

In [2]:

```
def water_jug_dfs(capacity_x,capacity_y,target):
    stack=[(0,0,[])]
    visited_states=set()
    while stack:
        x,y,path=stack.pop()
        if(x,y)in visited_states:
            continue
        visited_states.add((x,y))
        if x==target or y==target:
            return path+[(x,y)]
        #define possible jug operations
        operations=[
            ("fill_x",capacity_x,y),
            ("fill_y",x,capacity_y),
            ("empty_x",0,y),
            ("empty_y",x,0),
            ("pour_x_to_y",max(0,x-(capacity_y-y)),min(capacity_y,y+x)),
            ("pour_y_to_x",min(capacity_x,x+y),max(0,y-(capacity_x-x))),
        ]
        print( operations)
        for  operation,new_x,new_y in  operations:
            if 0<=new_x <=capacity_x and 0 <=new_y<=capacity_y:
                stack.append((new_x,new_y,path +[(x,y, operation)]))
    return None
#example usage:
capacity_x=4
capacity_y=3
target=2
solution_path=water_jug_dfs(capacity_x,capacity_y,target)
if solution_path:
    print("solution found:")
    for state in solution_path:
        print(f"({state[0]},{state[1]})")
else:
    print("no solution founded.")
```

```
[('fill_x', 4, 0), ('fill_y', 0, 3), ('empty_x', 0, 0), ('empty_y', 0, 0),
('pour_x_to_y', 0, 0), ('pour_y_to_x', 0, 0)]
[('fill_x', 4, 3), ('fill_y', 0, 3), ('empty_x', 0, 3), ('empty_y', 0, 0),
('pour_x_to_y', 0, 3), ('pour_y_to_x', 3, 0)]
[('fill_x', 4, 0), ('fill_y', 3, 3), ('empty_x', 0, 0), ('empty_y', 3, 0),
('pour_x_to_y', 0, 3), ('pour_y_to_x', 3, 0)]
[('fill_x', 4, 3), ('fill_y', 3, 3), ('empty_x', 0, 3), ('empty_y', 3, 0),
('pour_x_to_y', 3, 3), ('pour_y_to_x', 4, 2)]
solution found:
(0,0)
(0,3)
(3,0)
(3,3)
(4,2)
```

In [3]:

```
#TSF
from itertools import permutations

def calculate_total_distance(tour,distances):
    total_distance=0
    for i in range(len(tour)-1):
        total_distance += distances[tour[i]][tour[i+1]]
    total_distance += distances[tour[-1]][tour[0]] # Return to the startir
          #print(total_distance)
    return total_distance

def traveling_salesman_bruteforce(distances):
    cities = range(len(distances))
    min_distance = float('inf')
    optimal_tour = None

    for tour in permutations (cities):
        #print(tour)
        distance = calculate_total_distance(tour, distances)
        if distance < min_distance:
            min_distance = distance+
            optimal_tour = tour
    return optimal_tour, min_distance

distances_matrix=[
    [0,10,15,20],
    [10,0,35,25],
    [15,35,0,30],
    [20,25,30,0]
]
optimal_tour, min_distance = traveling_salesman_bruteforce(distances_matri

print("optimal tour:", optimal_tour)
print("Minimum Distance :", min_distance)
```

```
optimal tour: (0, 1, 3, 2)
Minimum Distance : 80
```

In [3]:

```
#TIC TAC TOE
board=[" "for x in range(9)]
def print_board():
    row1="| {} | {} | {} |".format(board[0],board[1],board[2])
    row2="| {} | {} | {} |".format(board[3],board[4],board[5])
    row3="| {} | {} | {} |".format(board[6],board[7],board[8])
    print()
    print(row1)
    print(row2)
    print(row3)
    print()
def player_move(icon):
    if icon=="X":
        number=1
    elif icon=="O":
        number=2
    print("your turn player {} ".format(number))
    choice=int(input("enter your move(1-9) ").strip())
    if board[choice-1]==" ":
        board[choice-1]=icon
    else:
        print()
        print("that space is taken ! ")
def is_victory(icon):
            if(board[0]==icon and board[1]==icon and board[2]==icon)or \
            (board[3]==icon and board[4]==icon and board[5]==icon)or \
            (board[6]==icon and board[7]==icon and board[8]==icon)or \
            (board[0]==icon and board[3]==icon and board[6]==icon)or \
            (board[1]==icon and board[4]==icon and board[7]==icon)or \
            (board[2]==icon and board[5]==icon and board[8]==icon)or \
            (board[0]==icon and board[4]==icon and board[8]==icon)or \
            (board[2]==icon and board[4]==icon and board[6]==icon):
                return True
            else:
                return False
def is_draw():
    if " " not in board:
        return True
    else:
        return False
while True:
    print_board()
    player_move("X")
    print_board()
    if is_victory("X"):
        print("X wins ! congratulations !!! ")
        break
    elif is_draw():
        print("it is a draw !!!")
        break
    player_move("O")
    if is_victory("O"):
        print_board()
        print("O wins! Congratulations !!! ")
        break
    elif is_draw():
        print("it is a draw !!!")
```

```
        break
```

```
|   |   |   |
|   |   |   |
|   |   |   |

your turn player 1
enter your move(1-9) 1

| X |   |   |
|   |   |   |
|   |   |   |

your turn player 2
enter your move(1-9) 2

| X | O |   |
|   |   |   |
|   |   |   |

your turn player 1
enter your move(1-9) 3

| X | O | X |
|   |   |   |
|   |   |   |

your turn player 2
enter your move(1-9) 4

| X | O | X |
| O |   |   |
|   |   |   |

your turn player 1
enter your move(1-9) 5

| X | O | X |
| O | X |   |
|   |   |   |

your turn player 2
enter your move(1-9) 6

| X | O | X |
| O | X | O |
|   |   |   |

your turn player 1
enter your move(1-9) 7

| X | O | X |
| O | X | O |
| X |   |   |

X wins ! congratulations !!!
```

In [2]:

```
#taking number of queens as input from user
print("Enter the number of queens:")
N=int(input())
board=[[0]*N for _ in range(N)]
def attack(i,j):
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if(k+l==i+j)or(k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if(not(attack(i,j))) and (board[i][j]!=1):
                board[i][j]=1
                if N_queens(n-1)==True:
                    return True
                board[i][j]=0
    return False
N_queens(N)
for i in board:
    print(i)
```

```
Enter the number of queens:
8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

In [1]:

```python
import heapq
road_graph = {
'Arad': {'Zerind': 75, 'Timisoara': 118,'Sibiu': 140},
'Zerind': {'Arad': 75, 'Oradea': 71},
'Timisoara': {'Arad': 118, 'Lugoj': 111},
'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80}
'Oradea':{'Zerind': 71, 'Sibiu': 151},
'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},
'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
'Drobeta': {'Mehadia': 75, 'Craiova': 120},
'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
'Bucharest': {'Fagaras': 211, 'Pitesti': 101}
}
heuristic_cost = {
"Arad": {"Bucharest": 366},
"Bucharest": {"Bucharest": 0},
"Craiova": {"Bucharest": 160},
"Dobreta": {"Bucharest": 242},
"Eforie": {"Bucharest": 161},
"Fagaras": {"Bucharest": 176},
"Giurgiu": {"Bucharest": 77},
"Hirsowa": {"Bucharest": 151},
"Lasi" : {"Bucharest": 226},
"Lugoj": {"Bucharest": 244},
"Mehadia": {"Bucharest": 241},
"Neamt": {"Bucharest": 234},
"Oradea": {"Bucharest": 380},
"Pitesti": {"Bucharest": 100},
"Rimnicu Vilcea": {"Bucharest": 193},
"Sibiu": {"Bucharest": 253},
"Timisoara": {"Bucharest": 329},
"Urziceni": {"Bucharest": 80},
"Vaslui": {"Bucharest": 199},
"Zerind": {"Bucharest": 374}
}
def heuristic_cost_estimate(node, goal):
    return heuristic_cost [node][goal]

def a_star(graph, start, goal):
    open_set = [(0, start)]
    came_from = {}
    g_score = {city: float('inf') for city in graph}
    g_score[start] = 0

    while open_set:
        current_cost, current_city = heapq.heappop(open_set)

        if current_city == goal:
            path = reconstruct_path(came_from, goal)
            return path

        for neighbor, cost in graph[current_city].items():
            tentative_g_score = g_score[current_city] + cost
            if tentative_g_score < g_score[neighbor]:
```

```
                g_score[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic_cost_estimate(neig
                heapq.heappush(open_set, (f_score, neighbor))
                came_from[neighbor] = current_city
    return None

def reconstruct_path(came_from, current_city):
    path = [current_city]
    while current_city in came_from:
        current_city = came_from[current_city]
        path.insert(0, current_city)
    return path
def calculate_distance(graph, path):
    total_distance = 0
    for i in range(len(path)-1):
        current_city = path[i]
        next_city = path[i+1]
        total_distance += graph[current_city][next_city]
    return total_distance
start_city = 'Arad'
goal_city = 'Bucharest'
path = a_star (road_graph, start_city, goal_city)
distance = calculate_distance(road_graph, path)

print ("Shortest Path from {} to {}: {}". format(start_city, goal_city, pa
print ("Total distance: {}".format(distance))
```

```
Shortest Path from Arad to Bucharest: ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pi
testi', 'Bucharest']
Total distance: 418
```

In [7]:

```
global facts
global rules
rules=True
facts=[["plant","mango"],["eating","mango"],["seed","sprouts"]]
def assert_fact(fact):
    global facts
    global rules
    if not fact in facts:
        facts +=[fact]
        rules=True
    while rules:
        rules=False
        for A1 in facts:
            if A1[0]=="seed":
                assert_fact(["plant",A1[1]])
            if A1[0]=="plant":
                assert_fact(["fruit",A1[1]])
            if A1[0]=="plant" and["eating",A1[1]] in facts:
                assert_fact(["human",A1[1]])
print(facts)
```

```
[['plant', 'mango'], ['eating', 'mango'], ['seed', 'sprouts']]
```

In [10]:

```
global facts
global rules
rules=True
facts=[["plant","mango"],["eating","mango"],["seed","sprouts"]]
def assert_fact(fact):
    global facts
    global rules
    if not fact in facts:
        facts +=[fact]
        rules=True
while rules:
    rules=False
    for A1 in facts:
        if A1[0]=="seed":
            assert_fact(["plant",A1[1]])
        if A1[0]=="plant":
            assert_fact(["fruit",A1[1]])
        if A1[0]=="plant" and["eating",A1[1]] in facts:
            assert_fact(["human",A1[1]])
print(facts)
```

```
[['plant', 'mango'], ['eating', 'mango'], ['seed', 'sprouts'], ['fruit', 'man
go'], ['human', 'mango'], ['plant', 'sprouts'], ['fruit', 'sprouts']]
```

In [1]:

```python
set1={1,2,3,4,5}
set2={3,4,5,6,7}

#union
union_set=set1.union(set2)
print(union_set,"union set")

#intersection
intersection_set=set1.intersection(set2)
print(intersection_set,"intersection set")

#difference set1
difference_set1=set1.difference(set2)
print(difference_set1,"set1-set2")

#symmetric difference
symmetric_difference_set=set1.symmetric_difference(set2)
print(symmetric_difference_set,"symmetric difference set")

#have common element in set 1 and set 2
have_common_elements=set1.isdisjoint(set2)
print("do set1 and set2 have common elements? ",not have_common_elements)

#adding
set1.add(6)
print("list after adding in set1",set1)

#removing from set 2
set2.remove(3)
print("list after removing in set2",set2)

```

```
{1, 2, 3, 4, 5, 6, 7} union set
{3, 4, 5} intersection set
{1, 2} set1-set2
{1, 2, 6, 7} symmetric difference set
do set1 and set2 have common elements?  True
list after adding in set1 {1, 2, 3, 4, 5, 6}
list after removing in set2 {4, 5, 6, 7}
```

In [2]:

```python
def count_occurence(main_string,substring):
    count=0
    start_index=0
    while start_index < len(main_string):
        index=main_string.find(substring,start_index)
        if index==-1:
            break
        count +=1
        start_index=index=index +1
    return count
main_string="ababab ab abbbbb ab ab bbbbb ab"
substring="ab"
result=count_occurence(main_string,substring)
print(f"substring'{substring}'occurs {result} time in the mainstring")
print(main_string.count(substring))
```

```
substring'ab'occurs 8 time in the mainstring
8
```

In [5]:

```
from sympy import symbols,Not,Or,simplify
def resolve(clause1,clause2):
    resolvent=[]
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1==Not(literal2) or literal2==Not(literal1):
                resolvent.extend([l for l in(clause1 + clause2) if l !=lit
    return list(set(resolvent))
def resolution(clauses):
    new_clauses=list(clauses)
    while True:
        n=len(new_clauses)
        print(new_clauses)
        print("---------------------")
        pairs=[(new_clauses[i],new_clauses[j]) for i in range(n) for j in

        for (clause1,clause2) in pairs:
            print(clause1)
            print(clause2)
            resolvent=resolve(clause1,clause2)
            print(resolvent)
            print("---------------------")
            if not resolvent:
                return True
            if resolvent not in new_clauses:
                new_clauses.append(resolvent)
    if n== len(new_clauses):
        return False
if __name__ == "__main__":
    clause1=[symbols('P'),Not(symbols('Q'))]
    clause2=[Not(symbols('P')),symbols('Q')]
    clause3=[Not(symbols('P')),Not(symbols('Q'))]
    clauses=[clause1,clause2,clause3]
    result=resolution(clauses)
    if result:
        print("The set of clause is unsatisified(found)")
    else:
         print("The set of clause is satisified")
```

```
[[P, ~Q], [~P, Q], [~P, ~Q]]
---------------------
[P, ~Q]
[~P, Q]
[~Q, Q, ~P, P]
---------------------
[P, ~Q]
[~P, ~Q]
[~Q]
---------------------
[~P, Q]
[~P, ~Q]
[~P]
---------------------
[[P, ~Q], [~P, Q], [~P, ~Q], [~Q, Q, ~P, P], [~Q], [~P]]
---------------------
[P, ~Q]
[~P, Q]
[~Q, Q, ~P, P]
---------------------
[P, ~Q]
[~P, ~Q]
[~Q]
---------------------
[P, ~Q]
[~Q, Q, ~P, P]
[~Q, Q, ~P, P]
---------------------
[P, ~Q]
[~Q]
[]
---------------------
The set of clause is unsatisified(found)
```

In [ ]:

```
1
```