In [7]:

```
#tic-tac-toe
a = [['-','-','-'],['-','-','-'],['-','-','-']]
flag = 0
ex = 1
win = 0
print("Enter the values space separated")
while(ex):
    for i in range(3):
        for j in range(3):
            print(a[i][j],end = ' ')
        print(end='\n')
    if flag == 0:
        row,col = input("Player1 'X':").split()
        if a[int(row)][int(col)] == '-':
            a[int(row)][int(col)] = 'X'
            flag = 1
        else:
            print("already taken")
    else:
        row,col = input("Player2 'O':").split()
        if a[int(row)][int(col)] == '-':
            a[int(row)][int(col)] = 'O'
            flag = 0
        else:
            print("already taken")
    for i in range(3):
        if (a[i][0] == a[i][1] == a[i][2]) and (a[i][1] == "X" or a[i][1] == 'O'):
            if flag == 1:
                print("player1 wins")
                win = 1
                ex = 0
            else:
                print("player2 wins")
                win = 1
                ex = 0
    for i in range(3):
        if (a[0][i] == a[1][i] == a[2][i]) and (a[0][i] == "X" or a[0][i] == 'O'):
            if flag == 1:
                print("player1 wins")
                win = 1
                ex = 0
            else:
                print("player2 wins")
                win = 1
                ex = 0
    if (a[0][0] == a[1][1] == a[2][2]) and (a[i][1] == "X" or a[i][1] == 'O'):
        if flag == 1:
            print("player1 wins")
            win = 1
            ex = 0
        else:
            print("player2 wins")
            win = 1
            ex = 0
    if (a[0][2] == a[1][1] == a[2][0]) and (a[i][1] == "X" or a[i][1] == 'O'):
        if flag == 1:
            print("player1 wins")
            win = 1
            ex = 0
        else:
            print("player2 wins")
            win = 1
            ex = 0
    if a[0].count('-') == 0 and a[1].count('-') == 0 and a[2].count('-') == 0:
        ex = 0
for i in range(3):
        for j in range(3):
```

```
            print(a[i][j],end = ' ')
        print(end='\n')
if win == 0:
    print("Draw")
```

```
Enter the values space separated
- - -
- - -
- - -
Player1 'X':1 2
- - -
- - X
- - -
Player2 'O':0 0
O - -
- - X
- - -
Player1 'X':0 2
O - X
- - X
- - -
Player2 'O':2 2
O - X
- - X
- - O
Player1 'X':1 1
O - X
- X X
- - O
Player2 'O':0 2
already taken
O - X
- X X
- - O
Player2 'O':2 0
O - X
- X X
O - O
Player1 'X':2 1
O - X
- X X
O X O
Player2 'O':1 0
player2 wins
O - X
O X X
O X O
```

In [8]:

```
#waterjug
def pour(jug1, jug2,max1,max2,fill):
    #max1, max2, fill = 4, 3,2
    print("%d\t%d" % (jug1, jug2))
    if jug1 is fill:
        return
    elif jug1 is max1:
        pour(0, jug1,max1,max2,fill)
    elif jug1 != 0 and jug1 is 0:
        pour(0, jug1,max1,max2,fill)
    elif jug1 is fill:
        pour(jug1, 0,max1,max2,fill)
    elif jug2 < max2:
        pour(max2, jug1,max1,max2,fill)
    elif jug1 < (max1-jug1):
        pour(0, (jug1+jug2),max1,max2,fill)
    else:
        pour(jug2-(max1-jug1), (max1-jug1)+jug1,max1,max2,fill)

max1 = int(input("Enter the jug1 capacity"))
max2 = int(input("Enter the jug2 capacity"))
fill = int(input("Final capacity"))
```

```
if max1 > max2:
    pour(0,0,max1,max2,fill)
else:
    pour(0,0,max2,max1,fill)
```

```
Enter the jug1 capacity4
Enter the jug2 capacity3
Final capacity2
0 0
3 0
3 3
2 4
```

In [9]:

```
#bfs
def createGraph(graph):

          n = int(input("Enter the number of nodes in graph:-"))
          for _ in range(n):
           node = input("Enter nodes and connected nodes in following format \n node:con
nectedNode1,connectedNode2,...").split(":")
           graph[node[0]]=node[1].split(",")
          return graph

def bfs(graph,start,dest):
      result = ["Not reachable",list()]
      visited = list()
      queue=list()
      queue.append(start)
      visited.append(start)
      while queue:
        currentNode=queue.pop(0)
        if( currentNode not in graph.keys() ):
         continue

        for node in graph[currentNode]:
         if( node not in graph.keys() ):
          continue

         if( node==dest ):
          result[0]= "Reachable"
          break

         if( node not in visited ):
          visited.append(node)
          queue.append(node)

      result[1] = visited
      return result

graph = dict()
graph = createGraph(graph)
start = input("Enter the starting point of traversal:-")
end= input("Enter the ending point of traversal:-")

result = bfs(graph,start,end)
print( "Result:-",result[0] )

print( "path traversed:-",result[1] )
```

```
Enter the number of nodes in graph:-4
Enter nodes and connected nodes in following format
 node:connectedNode1,connectedNode2,...0:1,2
Enter nodes and connected nodes in following format
 node:connectedNode1,connectedNode2,...1:0,2
Enter nodes and connected nodes in following format
 node:connectedNode1,connectedNode2,...2:0,1,3
Enter nodes and connected nodes in following format
 node:connectedNode1,connectedNode2,...3:2
Enter the starting point of traversal:-2
Enter the ending point of traversal:-1
```

```
Result:- Reachable
path traversed:- ['2', '0']
```

In [10]:

```
#dfs
def dfs_iterative(graph, start):
    stack,visitedvertex=[start],[]
    while stack:
        current = stack.pop()
        if current in visitedvertex:
            continue
        visitedvertex.append(current)
        for neighbor in graph[current]:
            stack.append(neighbor)
    return visitedvertex


adjacency_matrix = {1: [2, 3], 2: [4, 5],
                    3: [5], 4: [6], 5: [6],
                    6: [7], 7: []}

print(dfs_iterative(adjacency_matrix, 1))
```

```
[1, 3, 5, 6, 7, 2, 4]
```

In [13]:

```
#8 queens
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
    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 [12]:

```
#TSP
from sys import maxsize
```

```python
from itertools import permutations
def travellingSalesmanProblem(graph, s,V):
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
        min_path = maxsize
        next_permutation=permutations(vertex)
        for i in next_permutation:
            current_pathweight = 0
            k = s
            for j in i:
                current_pathweight += graph[k][j]
                k = j
            current_pathweight += graph[k][s]
            min_path = min(min_path, current_pathweight)
    return min_path

if __name__ == "__main__":

        graph = []
        n = int(input("enter number of nodes"))
        for i in range(n):
            graph.append(list(map(int,input().split())))
        s = 0
        print(travellingSalesmanProblem(graph, s,n))
```

```
enter number of nodes4
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
80
```

In [14]:

```python
#A*
class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    # This is heuristic function which is having equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start, stop):
        # In this open_lst is a lisy of nodes which have been visited, but who's
        # neighbours haven't all been always inspected, It starts off with the start
  #node
        # And closed_lst is a list of nodes which have been visited
        # and who's neighbors have been always inspected
        open_lst = set([start])
        closed_lst = set([])

        # poo has present distances from start to all other nodes
        # the default value is +infinity
        poo = {}
        poo[start] = 0

        # par contains an adjac mapping of all nodes
        par = {}
        par[start] = start
```

```
        while len(open_lst) > 0:
            n = None

            # it will find a node with the lowest value of f() -
            for v in open_lst:
                if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop
            # then we start again from start
            if n == stop:
                reconst_path = []

                while par[n] != n:
                    reconst_path.append(n)
                    n = par[n]

                reconst_path.append(start)

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all the neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
              # if the current node is not presentin both open_lst and closed_lst
                # add it to open_lst and note n as it's par
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    par[m] = n
                    poo[m] = poo[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update par data and poo data
                # and if the node was in the closed_lst, move it to open_lst
                else:
                    if poo[m] > poo[n] + weight:
                        poo[m] = poo[n] + weight
                        par[m] = n

                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)

            # remove n from the open_lst, and add it to closed_lst
            # because all of his neighbors were inspected
            open_lst.remove(n)
            closed_lst.add(n)

        print('Path does not exist!')
        return None

adjac_lis = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('A', 'D')
```

```
Path found: ['A', 'B', 'D']
```

Out[14]:

```
['A', 'B', 'D']
```

In [15]:

```
#AO*
def recAOStar(n):
    global finalPath
    print("Expanding Node:",n)
    and_nodes = []
    or_nodes =[]
    if(n in allNodes):
        if 'AND' in allNodes[n]:
            and_nodes = allNodes[n]['AND']
        if 'OR' in allNodes[n]:
            or_nodes = allNodes[n]['OR']
    if len(and_nodes)==0 and len(or_nodes)==0:
        return

    solvable = False
    marked ={}

    while not solvable:
        if len(marked)==len(and_nodes)+len(or_nodes):
            min_cost_least,min_cost_group_least = least_cost_group(and_nodes,or_nodes,{}
)
            solvable = True
            change_heuristic(n,min_cost_least)
            optimal_child_group[n] = min_cost_group_least
            continue
        min_cost,min_cost_group = least_cost_group(and_nodes,or_nodes,marked)
        is_expanded = False
        if len(min_cost_group)>1:
            if(min_cost_group[0] in allNodes):
                is_expanded = True
                recAOStar(min_cost_group[0])
            if(min_cost_group[1] in allNodes):
                is_expanded = True
                recAOStar(min_cost_group[1])
        else:
            if(min_cost_group in allNodes):
                is_expanded = True
                recAOStar(min_cost_group)
        if is_expanded:
            min_cost_verify, min_cost_group_verify = least_cost_group(and_nodes, or_node
s, {})
            if min_cost_group == min_cost_group_verify:
                solvable = True
                change_heuristic(n, min_cost_verify)
                optimal_child_group[n] = min_cost_group
        else:
            solvable = True
            change_heuristic(n, min_cost)
            optimal_child_group[n] = min_cost_group
        marked[min_cost_group]=1
    return heuristic(n)

def least_cost_group(and_nodes, or_nodes, marked):
    node_wise_cost = {}
    for node_pair in and_nodes:
        if not node_pair[0] + node_pair[1] in marked:
            cost = 0
            cost = cost + heuristic(node_pair[0]) + heuristic(node_pair[1]) + 2
            node_wise_cost[node_pair[0] + node_pair[1]] = cost
    for node in or_nodes:
        if not node in marked:
            cost = 0
            cost = cost + heuristic(node) + 1
            node_wise_cost[node] = cost
    min_cost = 999999
    min_cost_group = None
    for costKey in node_wise_cost:
        if node_wise_cost[costKey] < min_cost:
            min_cost = node_wise_cost[costKey]
            min_cost_group = costKey
```

```
    return [min_cost, min_cost_group]

def heuristic(n):
    return H_dist[n]

def change_heuristic(n, cost):
    H_dist[n] = cost
    return

def print_path(node):
    print(optimal_child_group[node], end="")
    node = optimal_child_group[node]
    if len(node) > 1:
        if node[0] in optimal_child_group:
            print("->", end="")
            print_path(node[0])
        if node[1] in optimal_child_group:
            print("->", end="")
            print_path(node[1])
    else:
        if node in optimal_child_group:
            print("->", end="")
            print_path(node)
H_dist = {
 'A': -1,
 'B': 4,
 'C': 2,
 'D': 3,
 'E': 6,
 'F': 8,
 'G': 2,
 'H': 0,
 'I': 0,
 'J': 0
}
allNodes = {
 'A': {'AND': [('C', 'D')], 'OR': ['B']},
 'B': {'OR': ['E', 'F']},
 'C': {'OR': ['G'], 'AND': [('H', 'I')]},
 'D': {'OR': ['J']}
}
optimal_child_group = {}
optimal_cost = recAOStar('A')
print('Nodes which gives optimal cost are')
print_path('A')
print('\nOptimal Cost is :: ', optimal_cost)
```

```
Expanding Node: A
Expanding Node: B
Expanding Node: C
Expanding Node: D
Nodes which gives optimal cost are
CD->HI->J
Optimal Cost is ::  5
```

In [18]:

```
#MINIMAX
def ConstBoard(board):
    print("Current State Of Board : \n\n");
    for i in range (0,9):
        if((i>0) and (i%3)==0):
            print("\n");
        if(board[i]==0):
            print("- ",end=" ");
        if (board[i]==1):
            print("O ",end=" ");
        if(board[i]==-1):
            print("X ",end=" ");
    print("\n\n");

#This function takes the user move as input and make the required changes on the board.
```

```
def User1Turn(board):
    pos=input("Enter X's position from [1...9]: ");
    pos=int(pos);
    if(board[pos-1]!=0):
        print("Wrong Move!!!");
        exit(0) ;
    board[pos-1]=-1;

def User2Turn(board):
    pos=input("Enter O's position from [1...9]: ");
    pos=int(pos);
    if(board[pos-1]!=0):
        print("Wrong Move!!!");
        exit(0);
    board[pos-1]=1;

#MinMax function.
def minimax(board,player):
    x=analyzeboard(board);
    if(x!=0):
        return (x*player);
    pos=-1;
    value=-2;
    for i in range(0,9):
        if(board[i]==0):
            board[i]=player;
            score=-minimax(board,(player*-1));
            if(score>value):
                value=score;
                pos=i;
            board[i]=0;

    if(pos==-1):
        return 0;
    return value;

#This function makes the computer's move using minmax algorithm.
def CompTurn(board):
    pos=-1;
    value=-2;
    for i in range(0,9):
        if(board[i]==0):
            board[i]=1;
            score=-minimax(board, -1);
            board[i]=0;
            if(score>value):
                value=score;
                pos=i;

    board[pos]=1;


#This function is used to analyze a game.
def analyzeboard(board):
    cb=[[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8],[0,4,8],[2,4,6]];

    for i in range(0,8):
        if(board[cb[i][0]] != 0 and
           board[cb[i][0]] == board[cb[i][1]] and
           board[cb[i][0]] == board[cb[i][2]]):
            return board[cb[i][2]];
    return 0;

#Main Function.
def main():
    choice=input("Enter 1 for single player, 2 for multiplayer: ");
    choice=int(choice);
    #The broad is considered in the form of a single dimentional array.
    #One player moves 1 and other move -1.
    board=[0,0,0,0,0,0,0,0,0];
    if(choice==1):
        print("Computer : O Vs. You : X");
```

```
        player= input("Enter to play 1(st) or 2(nd) :");
        player = int(player);
        for i in range (0,9):
            if(analyzeboard(board)!=0):
                break;
            if((i+player)%2==0):
                CompTurn(board);
            else:
                ConstBoard(board);
                User1Turn(board);
    else:
        for i in range (0,9):
            if(analyzeboard(board)!=0):
                break;
            if((i)%2==0):
                ConstBoard(board);
                User1Turn(board);
            else:
                ConstBoard(board);
                User2Turn(board);


    x=analyzeboard(board);
    if(x==0):
         ConstBoard(board);
         print("Draw!!!")
    if(x==-1):
         ConstBoard(board);
         print("X Wins!!! Y Loose !!!")
    if(x==1):
         ConstBoard(board);
         print("X Loose!!! O Wins !!!!")

#---------------#
main()
#---------------#
```

```
Enter 1 for single player, 2 for multiplayer: 1
Computer : O Vs. You : X
Enter to play 1(st) or 2(nd) :1
Current State Of Board :


-  -  -

-  -  -

-  -  -


Enter X's position from [1...9]: 1
Current State Of Board :


X  -  -

-  O  -

-  -  -


Enter X's position from [1...9]: 4
Current State Of Board :


X  -  -

X  O  -

O  -  -
```

```
Enter X's position from [1...9]: 3
Current State Of Board :


X  O  X

X  O  -

O  -  -


Enter X's position from [1...9]: 8
Current State Of Board :


X  O  X

X  O  O

O  X  -


Enter X's position from [1...9]: 9
Current State Of Board :


X  O  X

X  O  O

O  X  X


Draw!!!
```

In [19]:

```
#Alpha-Beta Pruning
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,values, alpha, beta):

    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                          False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

    else:
        best = MAX

        # Recur for left and
        # right children
```

```
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                            True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

# Driver Code
if __name__ == "__main__":

    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

```
The optimal value is : 5
```

In [20]:

```
#logic
from kanren import isvar,run,membero
from kanren.core import success,fail,goaleval,condeseq,eq,var
from sympy.ntheory.generate import prime,isprime
import itertools as it
def prime_test(n): #Function to test for prime
    if isvar(n):
        return condeseq([(eq,n,p)] for p in map(prime,it.count(1)))
    else:
        return success if isprime(n) else fail
n=var() #Variable to use
set(run(0,n,(membero,n,(12,14,15,19,21,20,22,29,23,30,41,44,62,52,65,85)),(prime_test,n
)))
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-20-5a44f8c26595> in <module>()
      1 #logic
----> 2 from kanren import isvar,run,membero
      3 from kanren.core import success,fail,goaleval,condeseq,eq,var
      4 from sympy.ntheory.generate import prime,isprime
      5 import itertools as it

ModuleNotFoundError: No module named 'kanren'

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

In [ ]: