N QUEENS TOY PROBLEM  
  
*#Number of queens*

print ("Enter the number of queens")

N = int(input())

*#chessboard*

*#NxN matrix with all elements 0*

board = [[0]\*N **for** \_ **in** range(N)]

**def** is\_attack(i, j):

*#checking if there is a queen in row or column*

**for** k **in** range(0,N):

**if** board[i][k]==1 **or** board[k][j]==1:

**return** **True**

*#checking diagonals*

**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\_queen(n):

*#if n is 0, solution found*

**if** n==0:

**return** **True**

**for** i **in** range(0,N):

**for** j **in** range(0,N):

'''checking if we can place a queen here or not

queen will not be placed if the place is being attacked

or already occupied'''

**if** (**not**(is\_attack(i,j))) **and** (board[i][j]!=1):

board[i][j] = 1

*#recursion*

*#wether we can put the next queen with this arrangment or not*

**if** N\_queen(n-1)==**True**:

**return** **True**

board[i][j] = 0

**return** **False**

N\_queen(N)

**for** i **in** board:

print (i)

SIMPLE REFLEX AGENT

print('All rooms are now clean.'

num\_rooms = int(input('Enter the number of rooms: '))

rooms = []

for i in range(num\_rooms):

status = input(f'Enter the status of room {i+1} (clean/dirty): ')

rooms.append(status)

# clean the rooms

for i, room in enumerate(rooms):

if room== 'clean':

print(f'the room {i+1} is clean moving to the next room')

if room == 'dirty':

print(f'Cleaning room {i+1}... and moving to the next room')

rooms[i] = 'clean'

print('All rooms are now clean.')

CSP PROBLEM  
def solutions():

# letters = ('s', 'e', 'n', 'd', 'm', 'o', 'r', 'y')

all\_solutions = list()

for s in range(9, -1, -1):

for e in range(9, -1, -1):

for n in range(9, -1, -1):

for d in range(9, -1, -1):

for m in range(9, 0, -1):

for o in range(9, -1, -1):

for r in range(9, -1, -1):

for y in range(9, -1, -1):

if len(set([s, e, n, d, m, o, r, y])) == 8:

send = 1000 \* s + 100 \* e + 10 \* n + d

more = 1000 \* m + 100 \* o + 10 \* r + e

money = 10000 \* m + 1000 \* o + 100 \* n + 10 \* e + y

if send + more == money:

all\_solutions.append(

(send, more, money))

return all\_solutions

print(solutions())

from queue import PriorityQueue

graph = {

'A': {'B': 4, 'C': 8},

'B': {'D': 5, 'E': 6},

'C': {'F': 12},

'D': {'G': 10},

'E': {'H': 8},

'F': {'I': 7},

'G': {'J': 14},

'H': {'K': 13},

'I': {'L': 11},

'J': {},

'K': {},

'L': {}

}

def best\_first\_search(graph, start, end):

visited = []

queue = PriorityQueue()

queue.put((0, start))

while not queue.empty():

node = queue.get()[1]

visited.append(node)

if node == end:

return visited

for neighbor, cost in graph[node].items():

if neighbor not in visited:

queue.put((cost, neighbor))

return None

start\_node = 'A'

end\_node = 'L'

path = best\_first\_search(graph, start\_node, end\_node)

if path:

print("Best First Search Path:", "->".join(path))

else:

print("No path found between", start\_node, "and", end\_node)

A \*

from queue import PriorityQueue

# define the graph with node positions and edge weights

graph = {

'A': {'B': 4, 'C': 8},

'B': {'D': 5, 'E': 6},

'C': {'F': 12},

'D': {'G': 10},

'E': {'H': 8},

'F': {'I': 7},

'G': {'J': 14},

'H': {'K': 13},

'I': {'L': 11},

'J': {},

'K': {},

'L': {}

}

# define the heuristic function (in this case, using straight-line distance)

heuristic = {

'A': 23,

'B': 20,

'C': 16,

'D': 13,

'E': 10,

'F': 7,

'G': 12,

'H': 9,

'I': 4,

'J': 8,

'K': 5,

'L': 0

}

def astar(graph, start, end, heuristic):

# initialize the visited set, the priority queue, and the path and cost dictionaries

visited = set()

queue = PriorityQueue()

path = {}

cost = {}

path[start] = [start]

cost[start] = 0

# add the start node to the queue with a priority based on the sum of the cost and heuristic

queue.put((heuristic[start], start))

while not queue.empty():

# get the node with the lowest priority (sum of cost and heuristic)

node = queue.get()[1]

if node == end:

# if we've reached the end node, return the path and cost

return path[end], cost[end]

if node not in visited:

# mark the node as visited

visited.add(node)

for neighbor, weight in graph[node].items():

# calculate the cost of getting to the neighbor node from the current node

new\_cost = cost[node] + weight

if neighbor not in cost or new\_cost < cost[neighbor]:

# if this is the first time we've visited the neighbor node or if the new cost is lower than the previous cost, update the cost and path dictionaries

cost[neighbor] = new\_cost

priority = new\_cost + heuristic[neighbor]

queue.put((priority, neighbor))

path[neighbor] = path[node] + [neighbor]

return None

# run the algorithm and print the path and cost

start\_node = 'A'

end\_node = 'L'

path, cost = astar(graph, start\_node, end\_node, heuristic)

if path:

print("A\* Path:", path)

print("Vertices:", "->".join(path))

print("A\* Cost:", cost)

else:

print("No path found between", start\_node, "and", end\_node)