# EXP 2 and 6

```
# Taking number of queens as input from user
N = int(input("Enter the number of queens: "))

# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]

def attack(i, j):
   #checking vertically and horizontally if there are any queen placed
   for k in range(0,N):
       if board[i][k]==1 or board[k][j]==1:
           return True
   #checking diagonally if there are any queen placed
   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
   # here we are checking whether we can place queen at ith row and jth
column
   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)
```

```
mitanshudodia@mitanshu-pc:~/College/iis/IIS Codes$ /bin/python3 "/home
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]
```

# EXP3 A

```
def bfs(graph, vertex, goal):    #function which takes graph source and
goal as input
   visited = [vertex]           #to check a node is visited or not
   queue = [vertex]             #queue with source in it
   while queue:
       deVertex = queue.pop(0)  #poping the first element of queue
       if deVertex == goal:
           print(f'The goal is found :- {goal}')  #if goal is found then
stop the code
           break
       print(deVertex)
       for adjacentVertex in graph[deVertex]:    #inserting neighbour of
current node in queue
           if adjacentVertex not in visited:
               visited.append(adjacentVertex)
               queue.append(adjacentVertex)

graph = { "a" : ["b","c"],
          "b" : ["a", "d", "e"],
          "c" : ["a", "e"],
          "d" : ["b", "e", "f"],
          "e" : ["d", "f", "c"],
          "f" : ["d", "e"]
             }

bfs(graph, 'a', 'f')
```

```
mitanshudodia@mitanshu-pc:~/College/iis/IIS Codes$ /bin/python3
a
b
c
d
e
The goal is found :- f
```

# EXP 3B

```python
from collections import deque

class Graph:
    # example of adjacency list (or rather map)
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],
    # 'C': [('D', 12)]
    # }

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

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

    # heuristic function with 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_node, stop_node):
       # open_list is a list of nodes which have been visited, but who's
neighbors
       # haven't all been inspected, starts off with the start node
       # closed_list is a list of nodes which have been visited
       # and who's neighbors have been inspected
       open_list = set([start_node])
       closed_list = set([])

       # g contains current distances from start_node to all other nodes
       # the default value (if it's not found in the map) is +infinity
       g = {}

       g[start_node] = 0

       # parents contains an adjacency map of all nodes
       parents = {}
       parents[start_node] = start_node

       while len(open_list) > 0:
           n = None

           # find a node with the lowest value of f() - evaluation
function
           for v in open_list:
               if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                   n = v;

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

           # if the current node is the stop_node
           # then we begin reconstructin the path from it to the
start_node
           if n == stop_node:
               reconst_path = []

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

```
                n = parents[n]

            reconst_path.append(start_node)

            reconst_path.reverse()

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

        # for all neighbors of the current node do
        for (m, weight) in self.get_neighbors(n):
            # if the current node isn't in both open_list and
closed_list
            # add it to open_list and note n as it's parent
            if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight

            # otherwise, check if it's quicker to first visit n, then m
            # and if it is, update parent data and g data
            # and if the node was in the closed_list, move it to
open_list
            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n

                    if m in closed_list:
                        closed_list.remove(m)
                        open_list.add(m)

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

    print('Path does not exist!')
    return None
adjacency_list = {
```

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

```
mitanshudodia@mitanshu-pc:~/College/iis/IIS Codes$ /bin/python3 "/home/mitanshudodia/College/iis/IIS Codes/A_star.py"
Path found: ['A', 'B', 'D']
```

# EXP 4

```
import random
def score(parent1, parent2):
    # doing crossover
    for i in range(len(parent1)-1, len(parent1)-4, -1):
        parent1[i], parent2[i] = parent2[i], parent1[i]
    #doint mutation by randomly selecting the genes
    mutation_index = [random.randint(0, len(parent1)-1) for i in
range(len(parent1)//2)]

    for i in mutation_index:
        if parent1[i] == '0':
            parent1[i] = '1'
        else:
            parent1[i] = '0'

        if parent2[i] == '0':
            parent2[i] = '1'
        else:
            parent2[i] = '0'

    score1 = parent1.count('1')
    score2 = parent2.count('1')
    #checking which child is better with more gene of type1
    if score1 > score2:
        return [''.join(parent1), score1]
    else:
```

```
        return [''.join(parent2), score2]

def genetic_algo():
    # Taking input as no. of parents
    n = int(input('Enter the number of parents: '))

    parents = []
    #taking parents genes as input 1 by 1

    for i in range(n):
        parents.append(list(input(f'Enter the parent{i+1}: ')))
    results = []

    #finding the score and storing it in results
    for i in range(len(parents)):
        for j in range(i+1, len(parents)):
            arr = [parents[i].copy(), parents[j].copy()]
            scores = score(parents[i], parents[j])
            results.append(scores + arr)

    # finding the best score among all combination of parents
    results.sort(key=lambda x: x[1], reverse=True)
    print(f'The best offspring among the parents is : {results[0][0]} and
the parents are {"".join(results[0][2])} and {"".join(results[0][3])}')

genetic_algo()
```

```
mitanshudodia@mitanshu-pc:~/College/iis/IIS Codes$ /bin/python3 "/home/mitanshudodia/College/iis/IIS Coc
Enter the numer of parents: 4
Enter the parent1: 110101
Enter the parent2: 011011
Enter the parent3: 100001
Enter the parent4: 010011
The best offspring among the parents is : 111111 and the parets are 111100 and 010011
```