DFS algorithm in Python

```
# DFS algorithm
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    print(start)

    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited


graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}

dfs(graph, '0')
```

BFS algorithm in Python

```
import collections
```

```
# BFS algorithm
def bfs(graph, root):

    visited, queue = set(), collections.deque([root])
    visited.add(root)

    while queue:

        # Dequeue a vertex from queue
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")

        # If not visited, mark it as visited, and
        # enqueue it
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)


if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

 # Selection sort in Python

```
def selectionSort(array, size):
```

```
    for step in range(size):
        min_idx = step

        for i in range(step + 1, size):

            # to sort in descending order, change > to < in this line
            # select the minimum element in each loop
            if array[i] < array[min_idx]:
                min_idx = i

        # put min at the correct position
        (array[step], array[min_idx]) = (array[min_idx], array[step])



data = [2, 1, 3, 5, 4]
size = len(data)
selectionSort(data, size)
print('Sorted Array in Ascending Order:')
print(data)

# Python3 program to solve N Queen
# Problem using backtracking

global N
N = 4


def printSolution(board):
            for i in range(N):
                        for j in range(N):
                                    if board[i][j] == 1:
```

```
                print("Q",end=" ")
            else:
                print(".",end=" ")
        print()


# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
def isSafe(board, row, col):

    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False
```

```
        return True


def solveNQUtil(board, col):

        # Base case: If all queens are placed
        # then return true
        if col >= N:
                return True

        # Consider this column and try placing
        # this queen in all rows one by one
        for i in range(N):

                if isSafe(board, i, col):

                        # Place this queen in board[i][col]
                        board[i][col] = 1

                        # Recur to place rest of the queens
                        if solveNQUtil(board, col + 1) == True:
                                return True

                        # If placing queen in board[i][col
                        # doesn't lead to a solution, then
                        # queen from board[i][col]
                        board[i][col] = 0

        # If the queen can not be placed in any row in
        # this column col then return false
        return False
```

```
# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
def solveNQ():
	board = [[0, 0, 0, 0],
			[0, 0, 0, 0],
			[0, 0, 0, 0],
			[0, 0, 0, 0]]

	if solveNQUtil(board, 0) == False:
		print("Solution does not exist")
		return False

	printSolution(board)
	return True



# Driver Code
if __name__ == '__main__':
	solveNQ()
```

Chatbot

```
# importing the required modules
```

```
from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer
# creating a chatbot
myBot = ChatBot(
 name = 'GS',
 read_only = True,
 logic_adapters = [
 'chatterbot.logic.MathematicalEvaluation',
 'chatterbot.logic.BestMatch'
 ]
 )
# training the chatbot
small_convo = [
 'Hi there!',
 'Hi',
 'How do you do?',
 'How are you?',
 'I\'m fine',
 'I feel awesome',
 'Excellent, glad to hear that.',
 'Not so good',
 'Sorry to hear that.',
 'What\'s your name?',
 ' I\'m GS. Ask me a math question, please.'
 ]
math_convo_1 = [
 'Pythagorean theorem',
 'a squared plus b squared equals c squared.'
 ]
math_convo_2 = [
 'Law of Cosines',
```

```
 'c**2 = a**2 + b**2 - 2*a*b*cos(gamma)'
 ]
# using the ListTrainer class
list_trainee = ListTrainer(myBot)
for i in (small_convo, math_convo_1, math_convo_2):
 list_trainee.train(i)
```