BFS 1

```
import itertools


def bfs(graph, start):
    visited = set()
    queue = [start]

    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            print(vertex)  # Replace with desired operation on the vertex

            neighbors = graph[vertex]
            unvisited_neighbors = itertools.filterfalse(visited.__contains__, neighbors)
            queue.extend(unvisited_neighbors)


# User input for constructing the graph
graph = {}
n = int(input("Enter the number of vertices in the graph: "))
for i in range(n):
    vertex = input(f"Enter vertex {i + 1}: ")
    neighbors = input(f"Enter neighbors of vertex {i + 1} (space-separated): ").split()
    graph[vertex] = neighbors

start_vertex = input("Enter the starting vertex for BFS: ")

# Calling the BFS function with user-provided inputs
bfs(graph, start_vertex)
```

BFS 2

```
import collections


# Function to perform Breadth First Search
def bfs(graph, start, goal):
    visited = set()  # Set to keep track of visited vertices
    queue = collections.deque([start])  # Queue for BFS traversal
    visited.add(start)  # Mark the start vertex as visited

    while queue:
        vertex = queue.popleft()
        print(vertex)

        # Check if the current vertex is the goal
        if vertex == goal:
            print("GOAL FOUND")
            return

        # Visit all the adjacent vertices of the current vertex
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

# Main program
if __name__ == "__main__":
    graph = collections.defaultdict(list)

    # Get the number of nodes from the user
    num_nodes = int(input("Enter the number of nodes: "))
```

```python
# Construct the graph
for i in range(1, num_nodes + 1):
    node = input("Enter node {}: ".format(i))
    adj_nodes = int(input("Enter the number of adjacent nodes: "))
    for j in range(adj_nodes):
        adj_node = input("Enter adjacent node: ")
        graph[node].append(adj_node)

start_node = input("Enter the starting node: ")
goal_node = input("Enter the goal node: ")

# Perform BFS
print("BFS traversal:")
bfs(graph, start_node, goal_node)
```

DFS 1

```
import itertools


def dfs(graph, start):
    visited = set()
    stack = [start]

    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            print(vertex)  # Replace with desired operation on the vertex

            neighbors = graph[vertex]
            unvisited_neighbors = itertools.filterfalse(visited.__contains__, neighbors)
            stack.extend(reversed(list(unvisited_neighbors)))

# User input for constructing the graph
graph = {}
n = int(input("Enter the number of vertices in the graph : "))
for i in range(n):
    vertex = input(f"Enter vertex {i + 1} : ")
    neighbors = input(f"Enter neighbors of vertex {i + 1} (space-separated) : ").split()
    graph[vertex] = neighbors

start_vertex = input("Enter the starting vertex for DFS: ")

# Calling the DFS function with user-provided inputs
dfs(graph, start_vertex)
```

DFS 2

```
class Graph():
    def __init__(self):
        self.graph = {}

    def dfs(self, v, visited=None):
        if visited is None:
            visited = set()

        visited.add(v)
        print(v, end=" ")

        for n in self.graph.get(v, []):
            if n not in visited:
                self.dfs(n, visited)

graph = Graph()

num_node = int(input("Enter number of nodes: "))
for i in range(num_node):
    node = int(input(f"Enter the {i+1} node: "))
    has_children = input(f"Does the node {node} have any children? (y/n): ")

    if has_children.lower() == 'y':
        children = []

        while True:
            print(f"Menu for node {node}")
            print("1. Add child")
            print("2. Finish adding children")
```

```
            choice = int(input("Enter your choice: "))
            if choice == 1:
                child = int(input(f"Enter child for node {node}: "))
                children.append(child)
            elif choice == 2:
                break
            else:
                print("Invalid choice!")

        graph.graph[node] = children

start_node = int(input("Start node: "))
print("DFS traversal:")
graph.dfs(start_node)
```

MST

```
graph = {}


num_vertices = int(input("Enter the number of vertices: "))

num_edges = int(input("Enter the number of edges: "))


for i in range(num_edges):

    while True:

        edge = input(f"Enter edge {i+1} in the format 'vertex1 vertex2 weight': ")

        edge = edge.split()

        if len(edge) == 3:

            break

        else:

            print("Invalid input, please try again.")

            continue


    vertex1 = edge[0]

    vertex2 = edge[1]

    weight = int(edge[2])


    if vertex1 not in graph:

        graph [vertex1] = {}

    if vertex2 not in graph:

        graph[vertex2] = {}

    graph[vertex1][vertex2] = weight

    graph[vertex2][vertex1] = weight


mst =[]

visited = set()
```

```
start_vertex = list(graph.keys())[0]
visited.add(start_vertex)

while len(visited) < num_vertices:
    min_edge = None
    for vertex in visited:
        for neighbor in graph [vertex]:
            if neighbor not in visited:
                if min_edge is None or graph [vertex][neighbor] < min_edge[2]:
                    min_edge = (vertex, neighbor, graph [vertex][neighbor])

    mst.append(min_edge)
    visited.add(min_edge[1])

print("Minimum Spanning Tree:")

for edge in mst:
    print(f"{edge[0]} - {edge[1]}; weight: {edge[2]}")
```

BACKTRACING

```
def is_safe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    for i, j in zip(range(row, len(board)), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def nqueens(n):
    board = [[0]*n for _ in range(n)]
    solutions = []  # empty list of solutions

    def backtrack(col):   #when the no. of rows and no. of columns are equal it appends everything
        if col == n:
            solutions.append([list(row) for row in board])
            return

        for row in range(n):
            if is_safe(board, row, col):
                board[row][col] = 1
                backtrack(col+1)
                board[row][col] = 0
```

```
    backtrack(0)
    return solutions

n = int(input("Enter the board (size) : "))

solutions = nqueens(n)

print(f"Number of solutions {len(solutions)}")

for i, solutions in enumerate(solutions):
    print(f"\nSolution {i+1}:")

    for row in solutions:
        print(" ".join(["Q" if cell== 1 else "-" for cell in row]))
```

BRANCH AND BOUND

```
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=" ")
        print()


def isSafe(row, col, slashCode, backslashCode, rowLookup, slashCodeLookup, backslashCodeLookup):
    if (slashCodeLookup[slashCode[row][col]] or
        backslashCodeLookup[backslashCode[row][col]] or
        rowLookup[row]):
        return False
    return True


def solveNQueensUtil(board, col, slashCode, backslashCode, rowLookup, slashCodeLookup, backslashCodeLookup):
    if col >= N:
        return True


    for i in range(N):
        if isSafe(i, col, slashCode, backslashCode, rowLookup, slashCodeLookup, backslashCodeLookup):
            board[i][col] = "Q"
            rowLookup[i] = True
            slashCodeLookup[slashCode[i][col]] = True
            backslashCodeLookup[backslashCode[i][col]] = True


            if solveNQueensUtil(board, col + 1, slashCode, backslashCode, rowLookup, slashCodeLookup,
backslashCodeLookup):
                return True


            board[i][col] = "-"
            rowLookup[i] = False
```

```
            slashCodeLookup[slashCode[i][col]] = False
            backslashCodeLookup[backslashCode[i][col]] = False

    return False

def solveNQueens():
    board = [["-" for _ in range(N)] for _ in range(N)]
    slashCode = [["-" for _ in range(N)] for _ in range(N)]
    backslashCode = [["-" for _ in range(N)] for _ in range(N)]
    rowLookup = [False] * N
    x = 2 * N - 1
    slashCodeLookup = [False] * x
    backslashCodeLookup = [False] * x

    for rr in range(N):
        for cc in range(N):
            slashCode[rr][cc] = rr + cc
            backslashCode[rr][cc] = rr - cc + N - 1

    if not solveNQueensUtil(board, 0, slashCode, backslashCode, rowLookup, slashCodeLookup,
backslashCodeLookup):
        print("Solution does not"-" exist")
        return False

    printSolution(board)
    return True

# Prompt the user to enter the board size
N = int(input("Enter the board size: "))
solveNQueens()
```