# Experiment 1: 8 Queens Problem

## Aim

To write a program to solve the 8 Queens problem using backtracking.

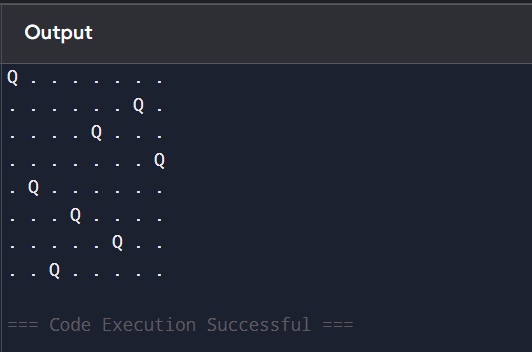
## Procedure

1. Place queens one by one in different columns, starting from the leftmost column.  
2. When placing a queen in a column, check for row and diagonal conflicts.  
3. If a conflict occurs, backtrack and try placing the queen in the next row.  
4. Repeat the process until all queens are placed without conflict.

## Code

def print\_solution(board):  
 for row in board:  
 print(" ".join("Q" if col else "." for col in row))  
  
def is\_safe(board, row, col, n):  
 for i in range(col):  
 if board[row][i]:  
 return False  
 for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  
 if board[i][j]:  
 return False  
 for i, j in zip(range(row, n, 1), range(col, -1, -1)):  
 if board[i][j]:  
 return False  
 return True  
  
def solve\_nqueens(board, col, n):  
 if col >= n:  
 return True  
 for i in range(n):  
 if is\_safe(board, i, col, n):  
 board[i][col] = 1  
 if solve\_nqueens(board, col + 1, n):  
 return True  
 board[i][col] = 0  
 return False  
  
def solve():  
 n = 8  
 board = [[0 for \_ in range(n)] for \_ in range(n)]  
 if solve\_nqueens(board, 0, n):  
 print\_solution(board)  
 else:  
 print("Solution does not exist")  
  
solve()

## Output



# Experiment 2: Depth First Search (DFS)

## Aim

To solve a problem using the Depth First Search algorithm.

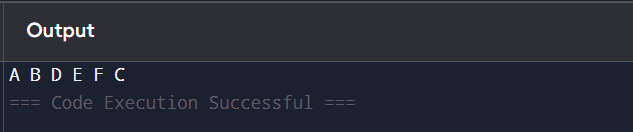
## Procedure

1. Start from the root node and push it to the stack.  
2. Pop the top item from the stack and mark it as visited.  
3. Push all adjacent unvisited nodes to the stack.  
4. Repeat until the stack is empty.

## Code

def dfs(graph, start, visited=None):  
 if visited is None:  
 visited = set()  
 visited.add(start)  
 print(start, end=' ')  
 for neighbor in graph[start]:  
 if neighbor not in visited:  
 dfs(graph, neighbor, visited)  
  
graph = {  
 'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': ['F'],  
 'F': []  
}  
  
dfs(graph, 'A')

## Output



# Experiment 3: MINIMAX Algorithm

## Aim

To implement the MINIMAX algorithm for decision making in game trees.

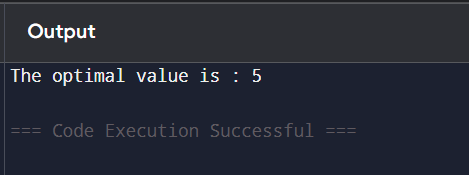
## Procedure

1. Generate the game tree.  
2. Assign a score to each leaf node.  
3. Propagate scores back using MIN and MAX at alternate levels.  
4. Choose the move with the optimal score.

## Code

def minimax(depth, nodeIndex, isMax, scores, h):  
 if depth == h:  
 return scores[nodeIndex]  
 if isMax:  
 return max(minimax(depth+1, nodeIndex\*2, False, scores, h),  
 minimax(depth+1, nodeIndex\*2 + 1, False, scores, h))  
 else:  
 return min(minimax(depth+1, nodeIndex\*2, True, scores, h),  
 minimax(depth+1, nodeIndex\*2 + 1, True, scores, h))  
  
scores = [3, 5, 6, 9, 1, 2, 0, -1]  
h = 3  
print("The optimal value is :", minimax(0, 0, True, scores, h))

## Output



# Experiment 4: A\* Algorithm

## Aim

To implement the A\* algorithm for shortest path finding.

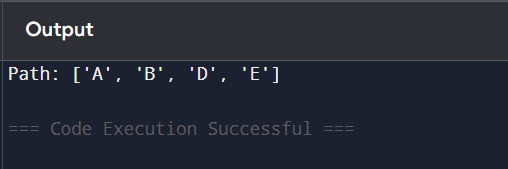
## Procedure

1. Maintain open and closed lists.  
2. Select the node with lowest f = g + h.  
3. Move it to closed list, and update neighbors.  
4. Repeat until goal is found.

## Code

from queue import PriorityQueue  
  
def a\_star(start, goal, graph, heuristic):  
 open\_list = PriorityQueue()  
 open\_list.put((0, start))  
 came\_from = {}  
 g\_score = {node: float('inf') for node in graph}  
 g\_score[start] = 0  
  
 while not open\_list.empty():  
 \_, current = open\_list.get()  
  
 if current == goal:  
 path = []  
 while current in came\_from:  
 path.append(current)  
 current = came\_from[current]  
 path.append(start)  
 return path[::-1]  
  
 for neighbor in graph[current]:  
 temp\_g = g\_score[current] + graph[current][neighbor]  
 if temp\_g < g\_score[neighbor]:  
 came\_from[neighbor] = current  
 g\_score[neighbor] = temp\_g  
 f = temp\_g + heuristic[neighbor]  
 open\_list.put((f, neighbor))  
 return None  
  
graph = {  
 'A': {'B': 1, 'C': 4},  
 'B': {'D': 1},  
 'C': {'D': 1},  
 'D': {'E': 3},  
 'E': {}  
}  
heuristic = {'A': 7, 'B': 6, 'C': 2, 'D': 1, 'E': 0}  
  
print("Path:", a\_star('A', 'E', graph, heuristic))

## Output



# Experiment 5: Backward Chaining

## Aim

To implement backward chaining to prove a hypothesis.

## Procedure

1. Start with the goal.  
2. Search for rules that conclude the goal.  
3. Recursively prove all conditions of those rules.  
4. If all subgoals are proven, the goal is proven.

## Code

class BC:

def \_\_init\_\_(self, rules, facts):

self.rules = rules # {conclusion: [premises]}

self.facts = set(facts)

def prove(self, goal):

if goal in self.facts:

return True

for conc, prem in self.rules.items():

if conc == goal:

if all(self.prove(p) for p in prem):

self.facts.add(goal)

return True

return False

rules = {

'flies': ['has\_wings', 'is\_bird'],

'is\_bird': ['has\_feathers']

}

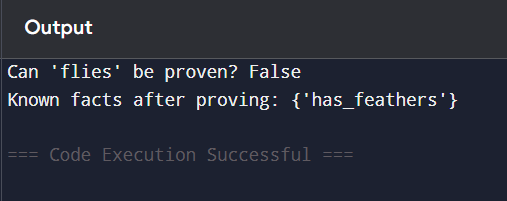
facts = ['has\_feathers']

bc = BC(rules, facts)

print(f"Can 'flies' be proven? {bc.prove('flies')}")

print(f"Known facts after proving: {bc.facts}")

## Output



# Experiment 6: Forward Chaining

## Aim

To implement forward chaining to infer new facts.

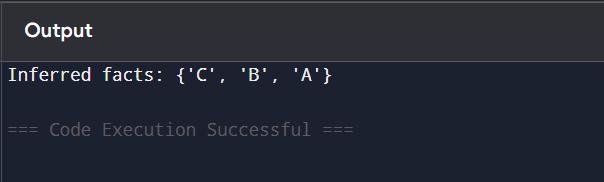
## Procedure

1. Start with known facts.  
2. Apply rules to infer new facts iteratively.  
3. Stop when no more facts can be inferred or the goal is achieved.

## Code

facts = {'A'}  
rules = {  
 'A': ['B'],  
 'B': ['C']  
}  
inferred = set(facts)  
while True:  
 added = False  
 for key, values in rules.items():  
 if key in inferred:  
 for value in values:  
 if value not in inferred:  
 inferred.add(value)  
 added = True  
 if not added:  
 break  
print("Inferred facts:", inferred)

## Output



# Experiment 7: Decision Tree

## Aim

To implement a decision tree classifier.

## Procedure

1. Import dataset and split into training/testing.  
2. Train decision tree classifier.  
3. Visualize the tree and evaluate accuracy.

## Code

from sklearn import tree  
from sklearn.datasets import load\_iris  
from sklearn.model\_selection import train\_test\_split  
  
iris = load\_iris()  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(iris.data, iris.target)  
clf = tree.DecisionTreeClassifier()  
clf.fit(X\_train, y\_train)  
print("Accuracy:", clf.score(X\_test, y\_test))

## Output



# Experiment 8: K-means Algorithm

## Aim

To implement K-means clustering algorithm.

## Procedure

1. Initialize k centroids.  
2. Assign points to the nearest centroid.  
3. Update centroids based on mean of points.  
4. Repeat until convergence.

## Code

from sklearn.cluster import KMeans  
import numpy as np  
  
X = np.array([[1,2],[1,4],[1,0],  
 [10,2],[10,4],[10,0]])  
kmeans = KMeans(n\_clusters=2)  
kmeans.fit(X)  
print("Centroids:", kmeans.cluster\_centers\_)  
print("Labels:", kmeans.labels\_)

## Output

