DAA

**# Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and #bound strategy.**

def knapsack(values, weights, capacity):

    n = len(values)  # Number of items

    # Create a 2D array to store maximum value at each n and capacity

    dp = [[0 for \_ in range(capacity + 1)] for \_ in range(n + 1)]

    # Build the table dp[][] in bottom-up fashion

    for i in range(1, n + 1):

        for w in range(1, capacity + 1):

            # Check if the weight of the current item i can be included

            if weights[i - 1] <= w:

                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])

            else:

                dp[i][w] = dp[i - 1][w]

    # The last cell of the dp table contains the maximum value

    return dp[n][capacity]

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

    # Input for number of items

    n = int(input("Enter the number of items: "))

    # Input for values and weights

    values = []

    weights = []

    for i in range(n):

        value = int(input(f"Enter the value for item {i + 1}: "))

        weight = int(input(f"Enter the weight for item {i + 1}: "))

        values.append(value)

        weights.append(weight)

    # Input for the maximum weight capacity of the knapsack

    capacity = int(input("Enter the maximum weight capacity of the knapsack: "))

    # Function call

    max\_val = knapsack(values, weights, capacity)

    print(f"The maximum value in the knapsack is: {max\_val}")

**#N-Queens -**

class NQueens:

    def \_\_init\_\_(self) -> None:

        self.size = int(input("Enter size of chessboard: "))

        self.board = [[False]\*self.size for \_ in range(self.size)]

        self.count = 0

    def printBoard(self):

        for row in self.board:

            for ele in row:

                if ele == True:

                    print("Q",end=" ")

                else:

                    print("X",end=" ")

            print()

        print()

    def isSafe(self,row:int,col:int) -> bool:

        # Check Column(above and below of the (row,col))

        for i in self.board:

            if i[col] == True:

                return False

        # Check backward slash(\) diagonal only in above direction

        i = row

        j = col

        while i >= 0 and j >= 0:

            if self.board[i][j] == True:

                return False

            i -= 1

            j -= 1

        # Check backward slash(\) diagonal only in below direction

        i = row

        j = col

        while i < self.size and j < self.size:

            if self.board[i][j] == True:

                return False

            i += 1

            j += 1

        # Check forward slash diagonal(/) only in above direction

        i = row

        j = col

        while i >= 0 and j < self.size:

            if self.board[i][j] == True:

                return False

            i -= 1

            j += 1

         # Check forward slash diagonal(/) only in below direction

        i = row

        j = col

        while i < self.size and j >= 0:

            if self.board[i][j] == True:

                return False

            i += 1

            j -= 1

        return True

    def set\_position\_first\_queen(self):

        print("Enter coordinates of first queen: ")

        row = int(input(f"Enter row (1-{self.size}): "))

        col = int(input(f"Enter column (1-{self.size}): "))

        self.board[row-1][col-1] = True

        self.printBoard()

    def solve(self,row:int):

        if row == self.size:

            self.count += 1

            self.printBoard()

            return

        if any(self.board[row]) is True:

            self.solve(row+1)

            return

        for col in range(self.size):

            if self.isSafe(row,col) == True:

                self.board[row][col] = True

                self.solve(row+1)

                self.board[row][col] = False

    def displayMessage(self):

        if self.count > 0:

            print("Solution exists for the given position of the queen.")

        else:

            print("Solution doesn't exist for the given position of the queen.")

solver = NQueens()

solver.set\_position\_first\_queen()

solver.solve(0)

solver.displayMessage()

**# knapsack\_greedy**

class Item:

    def \_\_init\_\_(self, profit, weight):

        self.profit = profit

        self.weight = weight

# Main greedy function to solve the fractional knapsack problem

def fractionalKnapsack(W, arr):

    # Sorting items based on the value-to-weight ratio in descending order

    arr.sort(key=lambda x: (x.profit / x.weight), reverse=True)

    # Result (value in the knapsack)

    finalvalue = 0.0

    # Looping through all items

    for item in arr:

        # If adding the item won't overflow, add it completely

        if item.weight <= W:

            W -= item.weight

            finalvalue += item.profit

        else:

            # If we can't add the current item, add a fractional part of it

            finalvalue += item.profit \* (W / item.weight)

            break  # Knapsack is full

    # Returning the final value

    return finalvalue

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

    # Input for the maximum weight capacity of the knapsack

    W = float(input("Enter the maximum weight capacity of the knapsack: "))

    # Input for number of items

    n = int(input("Enter the number of items: "))

    arr = []

    # Input for each item's profit and weight

    for i in range(n):

        profit = float(input(f"Enter the profit for item {i + 1}: "))

        weight = float(input(f"Enter the weight for item {i + 1}: "))

        arr.append(Item(profit, weight))  # Create an Item and add to the list

    # Function call

    max\_val = fractionalKnapsack(W, arr)

    print(f"The maximum value in the knapsack is: {max\_val:.2f}")

**# Fibonacci with step count**

**# Recursive Fibonacci with Step Count**

step\_count = 0

def fibonacci\_recursive(n):

global step\_count

step\_count += 1

if n <= 1:

return n

return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2)

# Input from the user

n = int(input("Enter the value of n: "))

result = fibonacci\_recursive(n)

print(f"Fibonacci({n}) = {result}")

print(f"Step Count: {step\_count}")

**# Iterative Fibonacci with Step Count**

def fibonacci\_iterative(n):

step\_count = 0 # Initialize step count

a, b = 0, 1

for i in range(n):

step\_count += 1 # Counting each loop iteration

a, b = b, a + b # Updating values of a and b

return a, step\_count

# Input from the user

n = int(input("Enter the value of n: "))

result, step\_count = fibonacci\_iterative(n)

print(f"Fibonacci({n}) = {result}")

print(f"Step Count (Iterative): {step\_count}")

BT

**# Student data – Structure, arrays, fallback**

pragma solidity ^0.5.0;

contract Crud {

struct User {

uint id;

string name;

}

User[] public users;

uint public nextId = 0;

function Create(string memory name) public {

users.push(User(nextId, name));

nextId++;

}

function Read(uint id) view public returns(uint, string memory) {

for(uint i=0; i<users.length; i++) {

if(users[i].id == id) {

return(users[i].id, users[i].name);

}

}

}

function Update(uint id, string memory name) public {

for(uint i=0; i<users.length; i++) {

if(users[i].id == id) {

users[i].name =name;

}

}

}

function Delete(uint id) public {

delete users[id];

}

function find(uint id) view internal returns(uint) {

for(uint i=0; i< users.length; i++) {

if(users[i].id == id) {

return i;

}

}

// if user does not exist then revert back

revert("User does not exist");

}

}

**# Smart contract for Bank account**

// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.7;

contract Bank {

address public owner;

uint public balance;

constructor() {

owner = msg.sender;

}

function deposit() external payable {

balance += msg.value;

}

function withdraw(uint \_amount) external {

require(balance >= \_amount, "Insufficient balance!");

payable(owner).transfer(\_amount);

balance -= \_amount;

}

}