P1 non

def fibonacci(n):

num1, num2 = 0, 1

for \_ in range(n):

print(num1, end=" ")

num1, num2 = num2, num1 + num2

# Main Function

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

N = 10

fibonacci(N)

p1.2 rec

def fib(n):

# Function

if n <= 1:

return n

return fib(n - 1) + fib(n - 2)

# Main function

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

N = 10

for i in range(N):

print(fib(i), end=" ")

Pra2 . huffman

import heapq

class HuffmanNode:

def \_\_init\_\_(self, data, char):

self.data = data

self.char = char

self.left = None

self.right = None

def \_\_lt\_\_(self, other):

return self.data < other.data

def print\_code(root, code):

if root is None:

return

if root.left is None and root.right is None and root.char.isalpha():

print(f"{root.char}: {code}")

return

print\_code(root.left, code + "0")

print\_code(root.right, code + "1")

def huffman\_coding(charArray, charfreq):

heap = [HuffmanNode(charfreq[i], charArray[i]) for i in range(len(charArray))]

heapq.heapify(heap)

while len(heap) > 1:

x = heapq.heappop(heap)

y = heapq.heappop(heap)

f = HuffmanNode(x.data + y.data, '-')

f.left = x

f.right = y

heapq.heappush(heap, f)

root = heap[0]

print\_code(root, "")

charArray = ['a', 'b', 'c', 'd', 'e', 'f']

charfreq = [5, 9, 12, 13, 16, 45]

huffman\_coding(charArray, charfreq)

Prac3. Knapsack using frac

class Item:

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

# Initialize the Item with weight and value

self.weight = weight

self.value = value

# Calculate the value-to-weight ratio for the item

self.ratio = value / weight

def get\_max\_value(items, capacity):

# Sort the items by their value-to-weight ratio in descending order

items.sort(key=lambda item: item.ratio, reverse=True)

total\_value = 0 # Initialize total value of the knapsack

# Iterate over the sorted items

for item in items:

# If the capacity is full, exit the loop

if capacity == 0:

break

# If the item can fit in the remaining capacity

if capacity >= item.weight:

# Reduce the capacity by the item's weight

capacity -= item.weight

# Add the item's full value to the total value

total\_value += item.value

else:

# If the item cannot fit, take the fraction that fits

total\_value += item.value \* (capacity / item.weight)

capacity = 0 # The knapsack is now full

return total\_value # Return the maximum value that can be achieved

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

# Create a list of items with their weights and values

items = [

Item(10, 60), # Item 1: weight 10, value 60

Item(20, 100), # Item 2: weight 20, value 100

Item(30, 120) # Item 3: weight 30, value 120

]

capacity = 50 # Define the maximum capacity of the knapsack

max\_value = get\_max\_value(items, capacity) # Calculate the maximum value

print("Maximum value in knapsack:", max\_value) # Output the result

Prac4. Knapsack using dp

def knapsack(weights, values, capacity):

n = len(weights) # Number of items

# Create a 2D list (dp) to store maximum profit for each subproblem

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

# Iterate over each item

for i in range(n + 1):

# Iterate over each weight capacity from 0 to the maximum capacity

for w in range(capacity + 1):

# If there are no items or the capacity is 0, the maximum profit is 0

if i == 0 or w == 0:

dp[i][w] = 0

# If the weight of the current item is less than or equal to the current capacity

elif weights[i - 1] <= w:

# Choose the maximum between including the current item or excluding it

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

else:

# If the current item's weight is more than the current capacity, exclude it

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

# The last cell of the dp table contains the maximum profit for the full capacity

return dp[n][capacity]

# Main execution

weights = [10, 20, 30] # Weights of the items

values = [60, 100, 120] # Values of the items

capacity = 50 # Maximum capacity of the knapsack

# Calculate the maximum profit that can be obtained with the given weights and values

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

print("Maximum profit:", max\_profit) # Output the result

prac5.rand

import random

import time

def deterministic\_quick\_sort(arr, low, high):

# Perform QuickSort using a deterministic pivot (last element)

if low < high:

pi = partition(arr, low, high) # Partition the array

deterministic\_quick\_sort(arr, low, pi - 1) # Sort the left part

deterministic\_quick\_sort(arr, pi + 1, high) # Sort the right part

def randomized\_quick\_sort(arr, low, high):

# Perform QuickSort using a randomized pivot

if low < high:

pi = randomized\_partition(arr, low, high) # Partition the array with a random pivot

randomized\_quick\_sort(arr, low, pi - 1) # Sort the left part

randomized\_quick\_sort(arr, pi + 1, high) # Sort the right part

def partition(arr, low, high):

# Partition the array around the pivot (last element)

pivot = arr[high] # Choose the pivot

i = low - 1 # Pointer for the smaller element

for j in range(low, high):

if arr[j] < pivot: # If current element is smaller than the pivot

i += 1 # Increment the index of the smaller element

swap(arr, i, j) # Swap the elements

swap(arr, i + 1, high) # Place the pivot in the correct position

return i + 1 # Return the partition index

def randomized\_partition(arr, low, high):

# Randomly choose a pivot and partition the array

random\_index = random.randint(low, high) # Random index for pivot

swap(arr, random\_index, high) # Swap the pivot with the last element

return partition(arr, low, high) # Partition the array

def swap(arr, i, j):

# Swap two elements in the array

arr[i], arr[j] = arr[j], arr[i]

def generate\_random\_array(size, range\_limit):

# Generate an array of random integers

return [random.randint(0, range\_limit - 1) for \_ in range(size)]

def copy\_array(arr):

# Return a copy of the array

return arr[:]

def main():

size = 10000 # Size of the random array

range\_limit = 10000 # Range limit for random integers

original\_array = generate\_random\_array(size, range\_limit) # Generate a random array

# Test deterministic QuickSort

arr1 = copy\_array(original\_array) # Copy the original array

start\_time = time.time\_ns() # Start time measurement

deterministic\_quick\_sort(arr1, 0, len(arr1) - 1) # Sort the array

end\_time = time.time\_ns() # End time measurement

print("Deterministic QuickSort time:", end\_time - start\_time, "ns") # Print execution time

# Test randomized QuickSort

arr2 = copy\_array(original\_array) # Copy the original array

start\_time = time.time\_ns() # Start time measurement

randomized\_quick\_sort(arr2, 0, len(arr2) - 1) # Sort the array

end\_time = time.time\_ns() # End time measurement

print("Randomized QuickSort time:", end\_time - start\_time, "ns") # Print execution time

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

main() # Run the main function

prac6.queen

def is\_safe(board, row, col):

    # Check the row on the left side

    for i in range(col):

        if board[row][i] == 1:

            return False

    # Check upper diagonal on the 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 the left side

    for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    return True

def solve\_n\_queens(board, col):

    if col == len(board):

        return True

    for i in range(len(board)):

        if is\_safe(board, i, col):

            board[i][col] = 1

            if solve\_n\_queens(board, col + 1):

                return True

            board[i][col] = 0  # Backtrack if placement leads to no solution

    return False

def print\_board(board):

    for row in board:

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

# Create an 8x8 chessboard with all cells initialized to 0

n = 8

chessboard = [[0 for \_ in range(n)] for \_ in range(n)]

# Place the first queen in the first row, first column

chessboard[0][0] = 1

# Try to solve the remaining queens

if solve\_n\_queens(chessboard, 1):

    print("Solution found:")

    print\_board(chessboard)

else:

    print("No solution exists.")